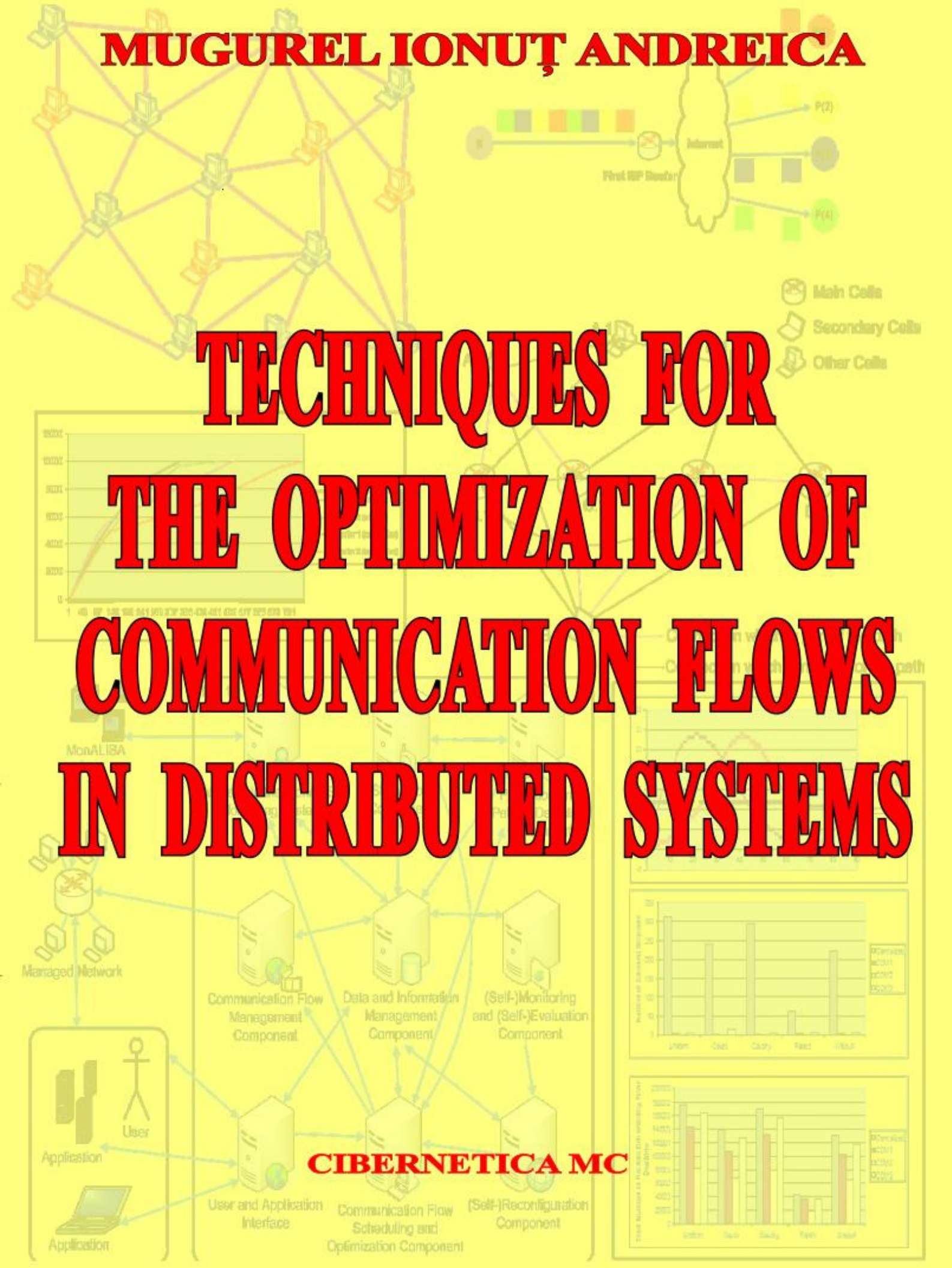
Techniques for the Optimization of Communication Flows in Distributed Systems

Mugurel Ionut Andreica

MUGUREL IONUȚ ANDREICA



TECHNIQUES FOR THE OPTIMIZATION OF COMMUNICATION FLOWS IN DISTRIBUTED SYSTEMS

CIBERNETICA MC

MUGUREL IONUŢ ANDREICA

TECHNIQUES FOR THE OPTIMIZATION OF COMMUNICATION FLOWS IN DISTRIBUTED SYSTEMS

CIBERNETICA MC Publishing House

Bucharest, 2010

COLECŢIA INFORMATICĂ, AUTOMATICĂ

Controlul ştiinţific: Prof. univ. dr. ing. NICOLAE ŢĂPUŞ

Tehnoredactare: Mădălina Ecaterina Andreica
Eliana-Dina Tîrşa

Coperta: Mihai Motea

Descrierea CIP a Bibliotecii Naţionale a României
ANDREICA, MUGUREL IONUŢ
Techniques for the optimization of communication flows in distributed systems / Mugurel Ionuţ Andreica. - Bucureşti : Cibernetica, 2010
Bibliogr.
ISBN 978-973-88451-4-5

316.77:336

EDITURĂ ACREDITATĂ DE CONSILIUL NAŢIONAL AL CERCETĂRII ŞTIINŢIFICE DIN ÎNVĂŢĂMÂNTUL SUPERIOR (C.N.C.S.I.S.).

Table of Contents

List of Figures

List of Tables

Pseudocode Listings

Abstract

A large variety of distributed systems have been developed and deployed throughout the world in order to solve various real-world problems or in order to meet specific demands and provide novel types of services. Communication is an essential property of every distributed system. The components of the distributed system must communicate with each other in order to meet the system's goals (e.g. solve a problem, or provide a service). As the attributes of a distributed system may vary considerably, so can their communication requirements. Some types of requirements can be easily met, but most of them pose challenges which have been insufficiently addressed so far. This book focuses on the broad topic of communication flow optimization in distributed systems and proposes two types of solutions: algorithmic and architectural.

The algorithmic solutions consist of online and offline methods and techniques, which are applicable in multiple types of communication optimization problems. The architectural solutions consist of centralized, decentralized and hybrid architectures for optimizing several communication metrics, which are relevant in many classes of distributed systems.

The book considers both point-to-point and multicast data transfers, as well as a wide range of communication parameters (e.g. bandwidth, latency, cost, reliability). The addressed communication flow optimization problems are modeled based on the communication requirements of a subset of the most important types of distributed systems. These requirements and the main communication parameters are identified in Chapter 2, where a thorough analysis of the current state-of-the-art in the field is presented.

Chapter 3 presents several decentralized peer-to-peer architectures for optimizing point-to-point communication flows, as well as flows generated by content search and retrieval. Many new types of techniques applicable in these architectures are also presented.

Chapter 4 proposes a centralized real-time data transfer scheduling architecture, together with many algorithmic techniques, for solving the problem of providing end-to-end Quality-of-Service (QoS) guarantees to the communication flows occurring in a distributed system.

Chapter 5 presents new algorithms for several offline point-to-point communication flow scheduling problems.

In Chapter 6 we present solutions for optimizing multicast data transfers. We present a peer-to-peer multicast tree architecture with bounded node degrees and small diameter, as well as offline algorithms for new models of multicast and broadcast in tree networks.

Chapter 7 considers several replica placement problems in tree-like networks and focuses on solving the associated communication optimization problems. We present new algorithms for optimally placing replicas such that the communication cost associated to accessing the replicas is minimized.

Chapter 8 concludes the book and presents an overview of the original contributions of this book.

The results presented in this book are based upon the author's Ph.D. thesis, which was written under the guidance of Prof. Nicolae Ţăpuş, Ph.D.

Acknowledgements

Writing this book was not an easy task and most certainly it wasn't a solitary task. I have quite a lot of people to thank to who, one way or another, shaped the direction, focus and intensity of my research and, thus, influenced the contents of this book. I would like to start by thanking my Ph.D. advisor, Prof. Nicolae Ţăpuş, who has been my mentor during all the years while I have been a Ph.D. student. I have learnt a lot from him and he has always guided my research in the most appropriate directions (I really don't know how he always managed to be right).

I would also like to thank my current and former colleagues from the Computer Science & Engineering Department, with whom I worked on many projects, from whom I learnt how to write papers or how to perform research activities, and who motivated me to work harder for my Ph.D.: Eliana-Dina Tîrşa, Florin Pop, Corina Stratan, Alexandru Costan, Alexandru Herişanu, Ciprian Mihai Dobre, Emil Sluşanschi, Laura-Elena Duţă, Răzvan Adrian Deaconescu, George Milescu, Lucian Muşat, Adrian Muraru, Mihaela Dediu, Traian Rebedea, Andrei-Horia Mogoş and Cătălin Leordeanu. Florin has motivated me with his long list of Ph.D. publications and gave me the confidence to express my ideas and results in a coherent manner, according to rigorous scientific standards. Corina has been awarded an IBM Ph.D. Fellowship twice. These outstanding achievements of hers made me work harder with the hope of at least reaching the same level of research excellency.

I would particularly like to address an extra special thanks to Eliana-Dina Tîrşa, who has helped me in more ways than I can remember in order to perform the research work required to write this book. She has helped me from a scientific point of view (as we wrote several research papers together), and many times she has provided me with the motivational support required to perform much of the work presented in this book.

I also have to acknowledge the contributions of Prof. Valeriu Iorga, Prof. Nicolae Cupcea, Prof. Valentin Cristea and Prof. Cristian Giumale, who shaped my Ph.D. experience in subtle, but meaningful ways.

I must also thank Alexandra Carpen-Amarie, Irina Borozan, Lucian-Ionuţ Bălăceanu and Eduard-Marius Dragomir. They were undergraduate students with whom I had the pleasure to collaborate while they were working on their diploma thesis. The results of our collaborations were published in several papers.

I am also in debt to Prof. Theodor Borangiu and Anamaria Dogar, who supported me for obtaining a prestigious IBM Ph.D. Fellowship. On the same line, I have to thank Alain Ozan and Dan Gârlaşu from Oracle, who supported my Ph.D. research with an Oracle Ph.D. scholarship. My Ph.D. research has also been supported by a research grant for young Ph.D. students, funded by the National Council for Scientific Research in Higher Education (CNCSIS), for which I am very grateful. Managing this grant taught me how to handle deadlines, how to coordinate my research activities efficiently, how to write reports, and how to plan and handle budget expenses.

I would also like to thank my „external" collaborators, for without their help or persistence, the contents of this book would have been of a lower quality. I thank Alexandru Iosup and Cătălin Dumitrescu, as they were the first ones to introduce me in the world of scientific research, even before I started my Ph.D. Together we developed ServMark and we had long talks over late-night beers in Delft. I would also like to thank Prof. Johan Pouwelse for his enthusiastic and pushy attitude towards the development of peer-to-peer content delivery networks. He provided me with both the motivation and the pressure to perform high quality research work. I must also thank Mihai Pătraşcu, together with whom I participated at the national and international olympiads in Informatics when we were still in high school. In the mean time, Mihai earned a Ph.D. degree from MIT and has a good perspective on the directions in which research in theoretical computer science should go. I thank him for sharing some of his thoughts on this topic with me. I also have to thank

Florin Manea, whose dedication to research excellency inspired me. We discussed several times about research topics that we both found interesting. Last but not least, I would like to thank Iosif Charles Legrand, for inviting me to join the MonALISA project's team and for providing much of the initial motivation for this book.

In the end, I would like to thank my family: my mother, Cristina Teodora Andreica, my father, Marin Andreica, and my sister, Mădălina Ecaterina Andreica. Without their constant support, I would not have been able to follow my research interests and write this book. However, their help was not only „collateral". They actively helped me with my research and we even published several papers together. I am proud to be part of a family of scientists ! ☺

Chapter 1 – Introduction

Distributed systems are becoming more and more wide-spread nowadays, being used for numerous purposes and in many fields. From the client-server model of the web to instant messaging applications, video conference systems, peer-to-peer file sharing programs, Grids and massively multiplayer online games, distributed systems arise as natural solutions to many problems. However, due to their nature, distributed systems present many challenging issues which have not yet been handled successfully. Some of these important issues are concerned with the efficient communication between the various components of a distributed system.

A distributed system is a complex ensemble of several components. From a structural point of view, a distributed system is composed of geographically distributed nodes collaborating at solving a common task (e.g. offering several types of services or running computationally intensive scientific applications). Each node is an individual entity which communicates with a subset (possibly all) of the other nodes in order to solve the common task. The nodes may have several roles and the roles of two nodes need not be identical. New nodes may join the system and old nodes may leave (gracefully or due to failures) at any time.

From a functional point of view, a simplistic (but sufficient for the purpose of this book) scheme of a distributed system consists of 2 layers: the *application layer* and the *communication layer*. The application layer is concerned with all the details regarding the common task (the computations to perform, the algorithms to use, the actions to take) and the communication layer offers communication services to the application layer. We considered this simplistic 2-layered structure because the focus of this book is on the communication aspects of a distributed systems and not on the computations the system performs.

Any distributed system fits both the structural and the functional scheme we proposed in the previous two paragraphs. The web is composed of servers and clients as nodes and the common task is that of providing access to content stored at various web sites (text documents, images, multimedia streams and many others). The nodes of an instant messaging system are mainly the applications running on the users' machines. Except for these, dedicated servers which facilitate user login, discovery and inter-connection may exist. The common task is that of facilitating the interaction between users.

Audio and video conference systems, peer-to-peer file sharing applications and massively multiplayer online games are composed of user clients (which may be differentiated into normal clients and super-nodes, depending on their capabilities and resources) and dedicated servers which facilitate content search, discovery and distribution, state maintenance, and user login and discovery. Grids are composed of a large variety of heterogenous resources (e.g. processors, storage devices) and the common task of the Grid nodes is to be available when large-scale complex problems requiring large amounts of resources need to be solved.

Note how our definition of a distributed system includes both what are traditionally called *distributed applications*, and *service-oriented* systems. Note also that, according to the proposed two-layer scheme, it is quite possible that a distributed system providing communication services may form (part of) the communication layer of another distributed system. Thus, distributed systems may be layered one on top of another. What actually defines a distributed system is its *application layer* (the common goal aimed by all of the system's nodes), while its *communication layer* is just an *attribute* of the system.

Communication is one of the key issues in every distributed system, but although it is extremely important, efficient communication is often difficult to achieve. Every system has its own communication requirements and, thus, its own way of defining communication efficiency; communication needs vary broadly and depend on many parameters. For instance, some systems require reliable, possibly out of order, message transmission, others require a steady transfer rate,

with occasional packet losses being considered acceptable, while others are interested in achieving high bandwidth communication flows. Unfortunately, for many real-life systems, no efficient communication techniques are known, and, in practice, suboptimal greedy or heuristic algorithms are used. This is caused by several factors, like:

- the complexity of the behavior patterns of the system (synchronization elements, system state information dissemination, cooperative coordinated solving of complex tasks and so on)
- the dynamic nature of the system (components/nodes may join or leave the system at any time)
- the scale of the system (it may be composed of thousands of nodes distributed throughout the world)

The topic of *communication optimization* in a distributed system is very broad and is related to many other topics, like network protocols, scheduling, resource allocation and reservation, approximation and online algorithms, data structures, and many others. Furthermore, within the topic of communication optimization there are many problems and challenges, some of which are solved or partially solved, others are open and many more will soon be posed, due to the rapid development of the communication and information technology.

In every setting where communication optimization is needed, there exists a set of internal requirements which must be fulfilled and a set of external factors which cannot be overcome. This is the set of constraints for the communication optimization problem. Moreover, there also exists an objective function whose value must be optimized. This function is defined as a combination of several measurable parameters. We classify the optimization objective as either hard or soft.

A hard objective means that the value of the objective function should reach a global optimum (minimum or maximum) or should be within some controllable threshold from this value (this is the domain of exact, approximation and competitive, online algorithms).

A soft objective is not as strict, i.e. the values of the objective function should be optimized as much as possible, but in some cases we do not know how to verify if a value is optimal or not. A good example of a soft objective is to minimize the response time of a scheduling algorithm which handles many scheduling or reservation requests. However, even considering everything else fixed, the response times depend on the algorithm chosen. Although it is obvious that some algorithms are better than others, we cannot tell which one is the optimal one, in the exact sense.

This book addresses a series of practical and theoretical communication optimization problems and proposes multiple novel solutions and approaches. The proposed solutions are of two types:

- algorithmic
    - online / offline algorithms
    - centralized / decentralized algorithms
    - data structures
- architectural
    - decentralized architectures (e.g. peer-to-peer)
    - centralized architectures (e.g. for data transfer scheduling)
    - hybrid architectures

Offline algorithms assume that complete information is available for performing their optimization decisions. Although having all the data is definitely much easier than having to deal with uncertainty, there are many situations in which this fortunate situation does not help much. Offline problems can be classified as belonging either to the class *P* or to the class *NP*, according to their computational complexity.

Problems in *P* have a polynomial time algorithm which solves them optimally for any input instance. Problems belonging to *NP* (most likely) do not have such an algorithm – only exponential algorithms can solve them to optimality. Although at the moment of writing this book the fact that

P≠NP has not been proved, it is largely believed to be so. It is clear that the problems belonging to *NP* cannot take advantage of the full information being available. Hopefully, there are approximation algorithms which can find solutions to some *NP* problems which are only a number of times worse than the corresponding optimal solutions. When they are only a constant number of times worse, we may decide that the solution produced by the approximation algorithm is acceptable. However, there are many *NP* problems for which no constant factor approximation algorithm exists.

Online problems are more difficult to tackle than offline problems, because of the incomplete information. A decision must be made in real-time, using only past and present information, without any knowledge of the future. Although knowledge of the future is unavailable, there are many situations in which the data forms patterns, which can be estimated and predicted based on the knowledge available so far. In these cases, a more or less accurate estimation of future information is possible. Most real-life communication optimization problems belong to this class.

A centralized algorithm is run on a single machine (or processor), while a decentralized algorithm is run on multiple machines (or processors) which communicate between them. A centralized algorithm is (most of the times) easier to implement and has the advantage that all the required information is gathered in a single place. Decentralized algorithms have to deal with state gathering, consistency and synchronization issues. Moreover, the information is not fully available in a single place or, if it is, it may be not be recent. The advantage of a decentralized algorithm over a centralized one is its increased fault-tolerance and its potential for performing more computational operations during the same period of time (because it may use multiple processors).

Centralized and decentralized architectures share the properties of centralized and decentralized algorithms. A centralized architecture stores all of its information and makes all of its decisions at a central server. In a decentralized architecture, data is stored over multiple nodes and decisions are made locally (each node makes its own independent decisions). Centralized architectures are easier to design and implement, but are less fault-tolerant. Decentralized architectures are more reliable and may use more computing power (and possess more storage space), but efficient coordination between the nodes is more difficult. A hybrid architecture contains both centralized and decentralized components.

The structure of this book consists of 8 chapters. The rest of Chapter 1 (Introduction) provides a high level overview of the contents of all the other chapters and should offer the reader a clear picture of the types of problems and solutions which are addressed and proposed in this book.

Chapter 2 – Related Work and Relevant Communication Parameters starts by identifying and classifying the main communication parameters and the main communication requirements which are relevant in many (if not most) of the existing distributed systems. The communication parameters are classified into two categories:

- category 1: communication parameters which are „visible” at the application layer of a distributed system
- category 2: communication layer parameters

Among the parameters belonging to category 1 are: bandwidth, time-related parameters (latency, latency jitter, earliest start time, latest finish time), or communication reliability. Category 2 contains parameters like communication topology, scalability, packet reordering, fault tolerance and security. As communication requirements, we identify:

- type of communication based on the number of receivers
    - unicast
    - multicast
    - broadcast
- communication paradigm
    - remote procedure call
    - message passing

- distributed objects
- connection-oriented

The communication paradigm can be considered both as a requirement (imposed by the application to the communication layer), or as a parameter (an API provided by the communication layer to the application).

The second part of Chapter 2 surveys and analyzes related work on the topics addressed by this book. The survey is focused on presenting the kinds of problems that have beedn addressed and the types of solutions that have been proposed so far in the literature, while the analysis is concerned with determining how efficient the solutions are, and with the identification of their advantages and disadvantages. This analytical survey provides the context into which the results presented in this book can be placed, as well as part of the motivation for the research work carried out and presented in this book.

In Chapter 2 we also argue for the need of application-level routing overlays, by presenting five convincing motivating scenarios. These scenarios, together with the current packet routing characteristics of the Internet (described in the following two paragraphs), motivated the work on peer-to-peer overlay networks and algorithms, which is presented in Chapter 3.

The current Internet architecture provides only best-effort data transfer services, without any kind of Quality of Service (QoS) guarantees. Under these conditions, providing a sufficient data transfer quality to many types of communication flows (like real-time or on-demand multimedia streams) is mostly a game of chance. Even when QoS guarantees are not required, the best effort model may still not be suitable because it may provide only a poor data transfer performance (e.g. low data transfer speed).

On the other hand, in some situations, the best effort attempts of the Internet are too much for the application's requirements. For instance, content sharing applications would like to upload files to other users without interferring too much with the traffic of the other applications running on the machine. Such low priority traffic would be better served by a less-than-best-effort data transfer service. Content sharing applications also have other specific requirements, like system-level data transfer fairness and user-perceived high data transfer and retrieval quality. Many times, these two objectives are in conflict with one another, no matter how we define data transfer (and retrieval) fairness and quality.

Chapter 3 – Peer-to-Peer Architectures and Techniques for Data Transfer and Retrieval Optimization addresses all the problems mentioned above (except for providing QoS data transfer guarantees) by proposing a family of scalable, collaborative peer-to-peer architectures. We first design a generic, fully distributed, peer-to-peer architectural model. This model defines the main properties that a peer-to-peer architectures should have:

- every peer must be aware of only a small number of other peers
- a message must be routed within the peer-to-peer overlay topology from any peer *X* to any (other) peer *Y* using local decisions only
- the local message routing decisions of a peer *X* must be based on local information only (i.e. information which can be obtained from the peer's neighbors or from itself)

The model proposes that every peer should have an identifier which is a point in a multi-dimensional space. Then, peers will form the overlay topology based on the distances between their corresponding identifiers. The model defines several neighbor selection methods based on which the peer-to-peer topology is created (and modified). These methods are named based on the geometric concepts they use:

- the *Independent Dimensions* method
- the *Local Voronoi* method
- the *Hyperplanes* method

Each neighbor selection method is defined as generic as possible (e.g. they allow for the use

of many distance functions, or for many types of hyper-planes). The described methods are convergent, i.e. in the absence of new peers joining the system or old peers leaving the system, the peer-to-peer topology reaches an equilibrium. This equilibrium is reached in a fully distributed manner (i.e. no central coordination is required).

The model also describes the procedures taken when a new peer joins the system or an old peer leaves it. These procedures are very light-weight and make use of the properties of the neighbor selection methods.

After defining the generic peer-to-peer architectural model, we present two peer-to-peer architectures for data transfer optimization. The first one uses the presented model as a core and, based on it, defines its own specific functions (e.g. message routing functions). This architecture uses an instance of the *Independent Dimensions* neighbor selection method, where each peers has a 1D identifier (actually, the identifiers are strings of characters, which, when sorted, form a 1D space). This peer-to-peer system has two goals:

- increasing the overall throughput of the point-to-point data transfers taking place between peers within the system
- balancing the traffic load in the system

The main method used for increasing the throughput is that of routing the data over multiple paths within the peer-to-peer overlay topology. This way, more data is transferred in parallel towards the destination, using different peers in the system. The routing process consists of two stages and makes use of the concept of *zone*. In the first stage, a peer within the zone is chosen as the next peer to make routing decisions. Then, a path towards this peer is chosen, based on a maximum flow algorithm and the data is *source-routed* towards the peer (i.e. the peers on the path towards the chosen peer are stored within each message).

The traffic load is balanced over multiple paths as a result of the routing process. Peers and paths which are known to be congested (i.e. have a low available bandwidth) are used for transferring data with a lower probability than peers and paths with higher amounts of available bandwidth.

The second data transfer optimization peer-to-peer system is a hybrid architecture and makes use of the concepts of the generic peer-to-peer model only partially. Each peer chooses a 1D identifier based on its ping times to K selected servers (landmarks) in the Internet. A central tracker stores information about all the peers existing in the system and the content they store. When a peer wants to transfer a content unit, it queries the tracker and obtains the list of peers currently storing the content. Then, the content is transferred over multiple paths from all the peers in the list. The peer-to-peer topology is not fixed, but rather created on-demand (e.g. TCP connections are opened when required and closed when they are no longer used). Like in the case of the first peer-to-peer system, mechanisms for distributing the data transfer over multiple paths are used.

In Chapter 3 we also present a fault-tolerant object storage peer-to-peer architecture with multidimensional range search capabilities. This architecture uses the generic peer-to-peer model defined in the first part of the chapter as its core. The architecture uses a special type of the *Hyperplanes* neighbor selection method, based on the L_1 metric. This special method has been chosen because it has two important properties:

1. it allows the system to perform geometric routing with a high probability (i.e. starting from any peer, we can reach the peer whose identifier is closest to a given point)
2. it is less computationally intensive than other methods which also have the first property (e.g. the *Local Voronoi* method)

On top of the core based on the generic peer-to-peer architectural model, we define a layer which implements the specific functions of the system (e.g. object management functions, like insertion and retrieval). An important property of this architecture is that it support efficient multidimensional range queries: the number of peers visited for answering a range query is proportional to the volume of the queried range. Moreover, the architecture does not have a

hierarchical structure like many other similar architectures, meaning that it is free of *hot spots*.

The second type of contributions presented in Chapter 3 consists of techniques applicable within peer-to-peer systems (irrespective of their topology). We address the system-level fairness and user-perceived quality problems by introducing novel techniques for handling data transfers at the communication layer of a (peer-to-peer) content sharing system. The data transfer quality is improved by establishing path reservations within the peer-to-peer topology. One of the novelties of this approach is that it is not specific to a certain type of topology. The system-level fairness problem is addressed by assigning a priority to each reservation. Then, when multiple reservations share an overlay link, they are assigned bandwidth shares proportional to their priority. This is achieved by limiting the bandwidth of incoming reservations at each peer to an appropriate fraction.

Novel mechanisms for estimating the upload capacity of a machine are also presented in Chapter 3. These mechanisms make use of the concept of *helper peers* and they are the first of their kind to be proposed in the literature. Using some of the techniques developed for estimating the upload capacity, we propose a novel upload congestion control mechanism which acts as a less-than-best-effort data transfer service.

In the final part of Chapter 3 we introduce ServMark, ar architecture for automated testing of distributed systems. It is well known that relevant testing of distributed systems is a very difficult or tedious task. ServMark can be used for automatically deploying the nodes of a distributed system and for generating workloads according to a wide range of user-specified parameters. Although ServMark was validated by testing web servers, it can also be used for testing peer-to-peer applications and protocols.

None of the techniques developed in Chapter 3 provide QoS data transfer guarantees. They are either *more-than-best-effort* or *less-than-best-effort* techniques. They are suitable for providing real-time optimizations, but cannot be used for planning data transfers in advance. This problem is addressed in **Chapter 4 - Real-Time Centralized Scheduling of Data Transfer Requests** by proposing a centralized data transfer scheduling framework and by developing several centralized data transfer scheduling techniques (based on novel algorithms and data structures). We begin by defining multiple types of data transfer requests and provide guidelines regarding their pricing methods, risk assessment and economic feasibility:

- fixed-bandwidth fixed-duration requests
- fixed-data fixed-duration requests
- fixed-bandwidth variable-duration requests

Multiple types of request processing modes are also considered (e.g. online and batched). The underlying assumption in Chapter 4 is that the owner of the network infrastructure is also the data transfer service provider. In order for this to happen, the data transfer provider must benefint from economic incentives. We discuss several principles for pricing such data transfer services and for managing the risks associated to such a service.

After defining and discussing the three types of data transfer requests, we focus on developing efficient online algorithmic techniques for the fixed-bandwidth fixed-duration category. First, we consider the single network link, for which we developed a new data structure, Time Slot Groups. For some particular situations of this case we developed novel extensions and frameworks for the segment tree and block partitioning data structures, focused on *range update* operations. Then, we considered the single network path case, for which multidimensional extensions of the frameworks developed for the segment tree and block partitioning data structures can be used efficiently.

The next type of network topology which naturally extends the single link and single path case is the tree topology. The tree topology is not just a particular case of a more generic „real” topology, because the structure of the Internet interconnections is hierarchical and, thus, in a sense, tree-like. Moreover, many solutions for general topologies can be approximated by a solution for a spanning tree of the topology. For the tree topology case we considered both the online and batched data transfer scheduling modes. The time was divided into equally-sized time slots. Based on their

duration (expressed as a number of time slots) and on the amount of bandwidth requested, four types of non-preemptive data transfer requests were considered:

- unit duration data transfer requests with full link usage
- unit duration data transfer requests with partial link usage
- multi-unit duration data transfer requests with full link usage
- multi-unit duration data transfer requests with partial link usage

Algorithmic techniques specific to each of the four types and each of the two scheduling modes are presented in Chapter 4. These techniques can be classified as having based on several distinct *algorithmic cores*, like:

- (multidimensional) data structures like the segment tree or the block partitioning method
- modelling the conflicts between the requests by using a conflict graph
- computing optimal matchings in bipartite graphs consisting of the data transfer requests (on one side) and the time slots (on the other side)

Finally, after considering several types of particular topologies (single link, path, tree), for which efficient (and topology-specific) algorithms and data structures were developed, we consider the case of a general network topology. We present a technique which handles the requests online and which makes its decisions based on computing a maximum flow in a time-expanded graph (a graph which models the traffic in the network over time). This technique is evaluated against several decentralized strategies and under multiple workloads (i.e. request submission patterns). The results show that it outperforms by a large margin any decentralized algorithm, suggesting that, at least for the time being, providing hard QoS guarantees can only be performed efficiently in a centralized manner.

All the algorithmic techniques presented for the single link, path, tree, and general topology case were based on dividing the time into equally-sized time slots. In Chapter 4 we also consider the other perspective, in which time is seen as being *event-based*. Each network link has several events assigned to it, representing the beginning of a data transfer (decrease of its available bandwidth) or the end of a data transfer (increase of the bandwidth). Using this event-based paradigm, we present several heuristic algorithmic methods for scheduling fixed-bandwidth fixed-duration data transfer requests in networks with arbitrary topology.

The last part of Chapter 4 presents novel, efficient, algorithmic solutions for the case when the conflicts between the data transfer requests are modelled using conflict graphs. Several particular cases for the structure of the mutual exclusion graph (e.g. tree, intersecting cliques) are considered, together with several possibilities regarding the timing constraints of the request (e.g. common deadline, separate earliest start time and latest finish time).

Offline optimization of multiple communication flows is considered in **Chapter 5 - Offline Optimal Scheduling of Constrained Point-to-Point Communication Flows**. In this chapter, a communication flow is analyzed at the packet level and decisions are made for individual packets. We consider several particular cases, like data items with divisible sizes, or communication flows with ordering constraints for their packets. We focus on objectives like minimizing the makespan of the (weighted) sum of completion times. For each case, we present new, polynomial-time algorithms. A particularity of the problems considered in Chapter 5 is that in most cases the sender and the receiver are connected by multiple (classes of) disjoint paths. From this perspective, these scheduling problems are related to the problems of scheduling jobs on multiple (classes of) processors.

In the last part of Chapter 5 we introduce a model for representing TCP conversations, based on which we consider the problem of minimizing the total TCP sender processing time. All the results presented in Chapter 5 are exclusively of a theoretical nature, and are based on a solid theoretical, efficiency and optimality analysis.

Another problem with the current Internet (which has not been mentioned so far) is that it can only provide point-to-point data transfers, although there are many situations in which multicast

communication is required (e.g. for sending notifications or for live streaming). Of course, multicast communication can be realized through multiple point-to-point data transfers. However, this presents two disadvantages:

- the consumed bandwidth would be much greater than in the case of using, for instance, a multicast tree
- it may happen that not all the destinations of the data are known by the source (e.g. when there are many such destinations)

Multicast has been recognized as being an important communication mechanism by the development of IP multicast. However, IP multicast is not widely deployed at the Internet level and, thus, it does not provide a good solution. Application-layer multicast routing has emerged as an efficient alternative to IP multicast. Application-layer routing has the advantage that multicast can be performed using any topology we consider fit and that we can emphasize any metric we want (e.g. bandwidth, delay, traffic load).

In **Chapter 6 - Multicast Communication Optimization Techniques** we present a novel peer-to-peer application-layer multicast routing architecture. The system maintains a multicast tree with bounded node degrees and small node diameter. Each peer maintains information regarding the number of nodes and the longest path in the part of the tree defined by each of its neighbors. This information is updated through gossiping. A convergence and correctness theorem is proved, which shows that after every peer insertion or departure, these values converge to their correct values after a number of gossiping rounds which is proportional to the diameter of the tree. A particular advantage of the presented architecture is that it is the first fully-decentralized and non-hierarchical system which can maintain a perfectly balanced tree (under certain limits regarding node arrivals and departures).

In Chapter 6 we also present novel offline multicast routing strategies for optimizing the reliability of the transmission or its duration. We consider a problem called maximum reliability k-hop multicast strategy, in which the root of a tree has to send a message to all the leaves of the tree. The message must reach the leaves after passing through an upper-bounded number of *relay nodes*. The problem is a re-interpretation of another problem, for which the best time complexity results from two papers published in 2001 and 2006. The solution presented in this book improves the previous best known solution by a logarithmic factor.

The last part of Chapter 6 presents new extensions of the well-studied single-port broadcast model in tree networks. We introduce sending and receiving constraints for the tree vertices, under several scenarios (the constraints are periodical, or they disappear after a fixed period of time). For each scenario we present new algorithmic solutions, based on dynamic programming or greedy techniques.

In content sharing systems, content (and content replica) placement is of paramount importance, because good placements can drastically reduce access times and improve user satisfaction. In **Chapter 7 – Optimal Replica Placement in Tree-Like Content Delivery Networks** we consider several such (offline) data placement problems in tree-like graphs. We consider multiple placement cost functions and we present new algorithmic techniques for optimizing the considered cost functions.

Many replica placement problems can be modelled as a k-center or k-median problem in the graph representing the network topology. Thus, we consider the 1-center problem in cactus graphs, for which we provide a new linear time solution. The developed algorithmic techniques can also be easily extended to solving the longest path or diameter problem on the same class of graphs. Then, we consider (k+p)-center and –median problem in trees (a (k+p)-center/median problem means that p replicas are already placed and we need to place at most k additional replicas such that the objective function is minimized).

Moreover, we also consider the additional constraint that the (at most) k vertices of the graph at which replicas are stored should form a connected subgraph, both in the center and in the median case. This constraint is meaningful when the replicas need to be updated frequently. In

order to solve these problems we introduce a novel algorithmic framework for many types of combinatorial optimization problems in trees. Then, we use this framework in order to develop efficient solutions for the considered problems.

We also discuss a balanced content replication problem in trees, which we model as a k-equitable coloring problem for which we present a new greedy solution based on a hierarchy of (re-)coloring permutations. The algorithm is evaluated in the context of a new tree reliability metric based on a solution to the *unrestricted vertex multicut (UVMC)* problem in trees. As a side result, we also present the first linear time solution to the *UVMC* problem (although a polynomial-time solution was previously known).

Another particular case that we consider is that of path graphs. We optimally solve a constrained k-center problem in these graphs by using a new linear time algorithm based on a geometric interpretation of the problem.

Finally, at the end of Chapter 7 we consider one of the most general classes of tree-like graphs: graphs with bounded pathwidth. We define a new algorithmic framework for solving combinatorial optimization problems on this class of graphs. Then, using this framework, we optimally solve a replica placement problem in these graphs.

Chapter 8 - Conclusions is the final chapter of this book, in which we conclude. We present an overview of the results, as well as a summary of the novel contributions of this book.

In summary, the addressed problems and proposed solutions presented in this book can be classified in one of the following categories:

1. peer-to-peer architectures for data transfer and content retrieval optimization, and algorithmic techniques applicable in peer-to-peer systems (Chapter 3)
2. real-time centralized scheduling of QoS-constrained data transfer requests (Chapter 4)
3. offline constrained communication flow scheduling (Chapter 5)
4. multicast peer-to-peer architectures and offline constrained multicast data transfer models and strategies (Chapter 6)
5. optimal offline replica placement in tree-like networks (Chapter 7)

Chapter 2 – Related Work and Relevant Communication Parameters

As presented in the previous chapter, this book addresses multiple communication optimization problems and proposes many types of solutions for these problems, ranging from distributed system architectures to centralized and decentralzied, online and offline algorithms. However, the purpose and validity of the presented techniques would be at least questionable without placing them in the broader context of the state-of-the-art communication optimization techniques developed by the scientific community. The purpose of this chapter is to survey the relevant (and related) existing techniques from the scientific literature, to analyze them from a critical point of view, and to provide the necessary context for the contributions presented in the rest of this book.

We begin by identifying the main (types of) communication requirements and parameters of a distributed system. Then, we argue for the need for application-layer routing peer-to-peer overlays using a series of 5 scenarios as motivational examples. Afterwards, we survey the relevant and most important types of existing peer-to-peer architectures, explictly stating what the original conribution of our novel proposed architectures is. We then continue on the same trend, by critically analyzing existing system-level fairness enforcing techniques and user satisfaction enhancement methods in peer-to-peer content sharing systems.

Afterwards, we analyze network link capacity estimation techniques and congestion control methods. We also survey and analyze some of the most important testing infrastructures of distributed systems. In Sections 2.8 and 2.9 we focus on data structures and algorithms for real-time scheduling of data transfers. Sections 2.10-2.12 are dedicated to offline problems: offline constrained communication flow scheduling, offline multicast strategies and offline replica placement in tree-like networks. Every time we survey existing results, we point out some of their drawbacks and show what is the novelty of our approaches. At the end of this chapter, the reader should have a clear picture of the context in which the techniques described in the rest of this book are placed, as well as a general idea regarding the description of these techniques.

2.1. Relevant Communication Requirements and Parameters

2.1.1. Application Layer Communication Requirements and Parameters

The communication requirements of a distributed system are dictated by its application layer, which uses the services provided by the communication layer. We will identify the most important requirements in this part of the report and introduce a set of communication parameters which are „visible” at the application layer of a distributed system. The communication requirements will be expressed in terms of these parameters. The communication requirements of a distributed system can be viewed as a set of rules specifying the acceptable levels of communication quality, in terms of a set of quantifyable parameters. The parameters will be either numeric or boolean and their values can be observed at the application layer. Thus, distributed applications can verify by themselves if the communication requirements are met by the communication layer.

2.1.1.1. Communication Paradigm

From the point of view of the communication paradigm, we will consider the following cases which present practical interest:

- the *remote procedure call* paradigm
- the *distributed objects* paradigm
- the *message passing* paradigm

- the *connection-based* paradigm

The communication paradigm is visible at the application layer mostly due to the provided communication API used by the application. For some applications, some paradigms are better suited, because they match the internal structure of the application better.

2.1.1.1.1. Remote Procedure Call

This model is similar to the procedural programming paradigm, in which one node may call some functions implemented in a different module. The difference between the usual procedural programming consists in the fact that the two modules are not necessarily located on the same computer. Applications structured on function and procedure calls benefit from this communication paradigm more than other applications (e.g. object-oriented applications).

2.1.1.1.2. Distributed Objects

Another communication model makes use of distributed objects. These objects contain data and methods, and some of these methods are public. The methods of a public-server object can be called by its clients. This model corresponds to the object-oriented programming paradigm, in which objects are located on different computers.
The model offers all the facilities of object-oriented programming, including separating the methods and data in an object into security access classes (public, protected, private). This model can easily be fit into modular and/or object-oriented applications, because it provides a seamless way of accessing remote objects as if they were local objects. Thus, it can be integrated without much effort into such an application. The model is less suited for applications whose internal structure is not focused on objects and/or classes.

2.1.1.1.3. The Message Passing Communication Paradigm

This model differs significantly from those presented above, where the communication is achieved through function and procedure calls. A source node sends a message to a destination node, containing some important information. The destination receives the message, decodes it, performs the necessary processing and then transmits back a response message (if necessary).

The main difference between this model and the remote procedure calls is given by the fact that RPC communication is synchronous (blocking), while message passing communication is (usually) asynchronous (non-blocking).

In the message passing paradigm, the transmission medium is shared by all the nodes and the nodes communicate among themselves by sending messages. A message is the smallest individual entity of information. Every message has one source node and one or more destination nodes. The application layer of the source nodes produces the content of the message and passes the content to the communication layer, which is responsible with its delivery to the destination(s).

This model is somewhat more difficult to integrate into applications, because we need to take care explicitly of sending and receiving messages. In the previous two models, messages were sent implicitly, within the function call (for the RPC model) or within a call of an object's method (for the distributed objects model). However, it offers the largest amount of flexibility to the application developer and, thus, can be adjusted to a wide variety of communication requirements, although at a cost of increased effort.

2.1.1.1.4. The Connection-Oriented Paradigm

Connection-oriented communication is based on establishing connections between a source node and one or more destinations. After a connection is established, the source node may send information to the other nodes to which the source node connected. Although the information may be split into messages when being sent along a connection, this communication paradigm is different from the message passing paradigm, because of its focus. The connections may be unidirectional or bidirectional (information may be sent both from and towards the initiator of the connection). The most common case consists of connections between two nodes, but, at least theoretically, connections between multiple nodes may be established (e.g. when the source node needs to send the same content to a subset of other nodes).

A connection is a logical concept, which does not need to have a direct correspondent at the physical level (for instance, a connection between two nodes does not need to be mapped to a

physical network path between those nodes, although this may be the case in some situations). As a result, information sent along a connection may follow different paths at different times (or may even be sent on multiple parallel paths).

A sigificant difference between this model and the previous one is the fact that information is seen as a continuous flow of data, not necessarily split into arbitrary chunks (messages). It is the application's responsibility to split the data flow into meaningful pieces.

2.1.1.2. Unicast, Multicast and Broadcast Communication

As the nodes of a distributed system work together towards a common goal, they need to exchange information about the state of the realization of the goal. From the point of view of the number of senders and receivers of the information, there are four communication types of practical interest:

- 1 sender, 1 receiver
- 1 sender, more than 1 receiver
- more than 1 sender, 1 receiver
- more than one sender, more than one receiver

The 1 sender – 1 receiver case is the most common and describes the *unicast* type of information distribution. When one sender needs to send information to several receivers, we distinguish two cases:

- *broadcast*: send the information to ***all*** the nodes in the distributed system
- *multicast*: send the information to a subset of the nodes of the distributed system

Obviously, broadcast is a special kind of multicast, where the subset of receivers consists of all the nodes.

The case when there are multiple senders and only one receiver is the inverse situation of the one sender - multiple receivers situation and we will consider it as a special type of multicast. This is because when such a situation occurs, it is usually as a reply (or acknowledgement) to a message previously sent by a node to a subset of other nodes.

The multiple senders - multiple receivers case is mostly of theoretical interest, because it is difficult to handle in practice. Although some optimizations may be possible by considering this case, it is difficult in practice to obtain and make use of information regarding this situation; thus, it is usually interpreted simply as multiple independent multicast transmissions.

From the application layer perspective, a protocol verifying whether unicast, multicast and broadcast transmissions are correctly supported by the communication layer can be easily implemented, if we consider some supplementary assumptions. The verification would consist in checking if all the destinations received the information correctly. We can make every destination send a reply message to the source node (or a confirmation message to a third party) as soon as it receives the incoming message from the source. However, at any moment in time, we cannot know if the original message was not delivered to the destination(s) or is still in transit somewhere inside the system. Thus, we need to introduce a time limit for receiving the reply message.

Unicast communication capabilities are required in every type of distributed system. Multicast and broadcast capabilities used to be less needed, but are becoming more prevalent nowadays. A few examples of systems requiring multicast and/or broadcast capabilities are the following: *audio and video conference systems*, *multimedia streaming farms* and applications (e.g. YouTube), *scientific applications* which generate important data which needs to be analyzed at a later time (see, for instance, the recent experiments at CERN), *massively multiplayer online games* which need to transmit the state of the game to the other players.

2.1.1.3. Bandwidth

Bandwidth represents the speed of the data transmission and is measured in bits (bytes) per time unit (usually Megabits or Gigabits per second). When a source node needs to send some data continuously (over a non-negligible time interval) to a destination node, the bandwidth of a data

transfer can be computed by the receiver (and, possibly, by the sender, too) and becomes an important factor in characterizing the communication between the nodes of a distributed system. When the nodes need to transfer large files or multimedia streams between each other, it is sometimes necessary that the transfer speed be larger than a threshold value (otherwise, the transfer would not finish on time or the quality of the multimedia stream would be poor). As bandwidth is relatively easy to measure in the case of end-to-end transfers (and only slightly more complicated in the case of multicast transfers), it is one of the communication parameters best describing the quality of the communication.

Sometimes the bandwidth achieved by a data transfer is also called throughput or goodput (meaning the transfer speed of the useful data, ignoring the lower level headers and trailers). Bandwidth is an important parameter in data-intensive applications and systems (e.g. applications which need to transmit large files by some specified deadline; audio/video conference and multimedia streaming systems which need to transmit data at a constant, specified bandwidth; file sharing applications, where the file download speed is the most appreciated parameter).

2.1.1.4. Time Parameters

Time is a very important parameter when characterizing the communication between the nodes of a distributed system. There are several types of relevant time constraints and we will discuss most of them in this section.

2.1.1.4.1. Latency

The latency is the time duration between the time moment when a message is sent by the sender and the time moment when the message is received by the receiver. It measures the amount of time during which the message is in transit. End-to-end latency depends on the latencies of the network links composing the network path on which the message is routed and on the delays encountered at the intermediate nodes and network devices (routers, switches). It is usually desirable that latency is as low as possible, but there are situations when it is more important for the latency to be relatively constant (not have large variations), in order to obtain a predictable steady transfer rate.

Having a small latency is particularly important in real-time distributed systems, where data has to arrive to the receiver(s) as soon as possible (e.g. nuclear power plants; systems allowing to perform medical operations on the Internet). Having a constant latency is important in multimedia streaming applications, where the user-perceived quality would be degraded by large variations in latency.

2.1.1.4.2. Latency Jitter

Latency jitter represents the variation of the latency over time. As mentioned in the previous subsection, there are situations when it is more important for the latency to be relatively constant rather than low (e.g. multimedia streaming systems). In these situations, a high amount of jitter is undesirable. Both latency and jitter are measurable at the receiver. The jitter can be estimated from the interarrival times between consecutive messages received from the same source. Latency can be computed if the message is time-tagged at the source and the deviation between the clocks of the sender and the receiver is either known or negligible.

2.1.1.4.3. Earliest Start Time

The communication layer does not necessarily offer only immediate communication services (i.e. sending out the information right away or as soon as possible). In some situations, in order to improve the overall communication quality (or simply in order to ensure the required quality levels of other nodes), the communication layer may employ scheduling techniques. Most commonly, these techniques schedule the data transfers in time and make use of efficient data structures. In these cases, the application layer may submit a data transfer request to the communication layer in advance. The request may have an associated earliest start time *t*, meaning that the data transfer should not start before time t (possibly because the data is not ready before that time). The communication layer must consider the earliest start time constraints when scheduling the data transfers.

Earliest start time constraints arise, for instance, in multimedia streaming applications, when the user requests for the multimedia stream in advance, but specifies that it will only be available after a certain time and in file transfer scheduling with dependencies (we cannot start a file transfer before another activity finishes).

2.1.1.4.4. Latest Finish Time

Just like the earliest start time, a data transfer request may have an associated latest finish time, i.e a time moment *t* before which the data transfer should be completed (deadline). The same types of distributed systems mentioned for the *Ealiest Start Time* parameter can also make use of a *Latest Finish Time* parameter. However, this parameter is more common in practices. Usually, users and/or applications request some data and specify some deadline before they need to receive that data (e.g.). Using deadlines is common to our society and, thus, it is also prevalent in the use of distributed systems.

2.1.1.4.5. Lower and Upper Bounds for the Transmission Duration

Still considering the data transfer requests model, other time constraints which may be specified are a lower and upper bound for the data transfer duration. This allows the communication layer to have more flexibility in creating the data transfer schedules. In some situations, the total transmission duration (makespan) needs to be minimized (i.e. send the data as quickly as possible).

Other time parameters can be defined based on those already mentioned. For instance, in the case of multiple communication flows, we may be interested in minimizing the sum of (weighted) completion times.

2.1.1.5. Communication Reliability

Communication reliability is a statistical measure regarding the good functionality of the communication layer. Communication reliability can be mathematically defined in many ways, but the important thing is that it captures the „trust" that the communication layer will provide the required communication quality levels. A reliability metric and value (between 0 and 1) can be associated to every (other) communication parameter P, signifying the „trust" that the values of the parameter P will be of „good" quality (above and/or below a pre-specified threshold).

Reliability is defined as the probability that the communication layer will **not** *fail*, where a failure can be defined in many ways. If we consider the reliability of delivering messages to the destination(s) as 0.9, this tells us that there is a 0.1 probability that a message will not reach its destination (will be *lost* somewhere inside the system). If the reliability of a message reaching its destination(s) after a time duration dt in [lbt, ubt] is 0.75, this tells us that there is a 25% chance that the message will reach a destination after a time duration either smaller than lbt or larger than ubt.

Some distributed systems can handle lower levels of reliability, by retransmitting data. However, critical and real-time systems (like those supervising nuclear power plant processes or medical operations over the Internet) required a very high reliability.

2.1.2. Communication Layer Parameters

There exists a subset of parameters which are not exposed to the application layer or their impact upon the communication quality cannot be clearly quantified at that layer. We will discuss these parameters in this section.

2.1.2.1. Communication Topology

The communication topology of a distributed system refers to the communication interactions between the system nodes. The topology is very important especially for the task of message routing. In order for a message to reach its destination(s), it will pass through several (possibly zero) intermediate nodes of the distributed system (as well as some network devices, like switches or routers).

The communication layer of a distributed system is usually built on top of a communication layer available from the operating system or on top of some communication libraries. As such, the

communication layer of a distributed system does not have much (any) control on the network (physical) path a message takes from a source node to the destination(s). The communication layer may estimate some parameters of the OS/library communication layer (the same parameters the application layer can estimate regarding the communication layer- latency, jitter, bandwidth) and make decisions based on these parameters, but it cannot control the network path.

Under these circumstances, the only way the communication layer can impose some control is by scheduling data transfers in time and by establishing a virtual communication topology, overlaid on the physical, network topology. Such a topology is described as a graph, in which the vertices are the nodes of a distributed system and the edges correspond to point-to-point connections between the nodes. The graph may be directed or undirected (i.e. the point-to-point connections may be uni- or bidirectional).

The easiest way for a communication layer is to not establish any virtual communication topology and rely fully on the OS/library communication layer. However, in order to achieve increased communication performance, many communication layers choose to establish their own communication topology, which brings many new challenges.

We will use the classification employed by peer-to-peer systems in order to characterize the communication topology of a distributed system. There are two types of topologies: structured and unstructured. In the case of *structured* topologies, the nodes have identifiers which are mapped into a metric space. The connections between nodes are related to the distance measure of the metric space, in the sense that nodes whose identifiers are close in the metric space are connected together (through a bidirectional connection).

Routing a message from a source node to a destination is performed through intermediate nodes such that the distance between the identifier of the current (intermediate) node and the destination node constantly decreases. Structured communication topologies have the advantage that a node may only have partial knowledge of the topology (at the very least, he only needs to know its own direct neighbors). However, structured topologies are very sensitive to churn. Large churn values imply that the topology is not stable and, thus, the message routing process may not converge to the destination properly. *Unstructured* topologies are very simple to implement, yet this is their only advantage. As the system grows larger, the nodes either need to store more and more information about the newly joined nodes or they need to exchange more and more node discovery messages (thus flooding the system).

The communication topology is also related to the way the actions of the distributed system are coordinated. A centralized coordination ensures a consistent view, but limits the scalability (mentioned below). Decentralized decision making increases the scalability, but introduces problems related to consistency and consensus (which are harder to achieve than in the case of the centralized coordination). The two types of coordinating actions lead to significantly different communication patterns. In between them there are many types of hybrid coordination strategies, which make use of both centralized and decentralized decision making strategies.

2.1.2.2. Scalability

Scalability is a property of the communication layer of a distributed system which ensures that the communication performance degrades gracefully as the number of nodes increases. First of all, it is normal that some communication parameters degrade as the number of nodes increases, because extra effort is required to reach the new nodes as destinations of messages. On the other hand, an increased number of nodes may improve other parameters (e.g. throughput, because there will be more paths on which messages can be sent).

Scalability is strongly connected to the communication topology. Structured communication topologies are very scalable (but sensitive to churn), while unstructured communication topologies usually do not scale well. Scalability is a desirable feature in every distributed system, but it is particularly important in those systems which are likely to contain a large amount of nodes/users (e.g. peer-to-peer file sharing, multimedia streaming).

2.1.2.3. Fault Tolerance

The fault tolerance of the communication layer is tightly connected to the reliability measures visible at the application layer. The fault tolerance is indirectly visible to the application layer through the reliability metrics, but, otherwise, it is an „internal" parameter of the communication layer. Unlike reliability, fault tolerance is not easily quantified. We might be tempted to define it through a probability measure (just like reliability), but, in fact, fault tolerance has a more qualitative nature.

We propose in this section a novel description of the fault tolerance of the communication layer based on scenarios. Each scenario describes an „unexpected" event (fault) and the fault tolerance measure consists of a description of the system's response to the event. For instance, a possible scenario may consists of the failure of a node in the distributed system through which a large portion of the messages in the system were routed. The response to such an event may consist of the system adapting itself and rerouting the messages through other nodes or the system may take no immediate action (and, thus, some messages may be lost or delayed).

In the context of the communication topology, unstructured topologies are more fault tolerant than structured ones, because they are less complex (decreased complexity implies that there are fewer things that can go wrong when a fault occurs).

2.1.2.4. Packet Reordering

Packet reordering is discussed in the context of a communication flow between a source and one or more destinations. The packets (messages) composing a flow can be ordered sequentially. Normally, these packets must be sent and received in the logical sequential order. However, if the receiver has the ability to properly reorder the packets, then we can achieve some improvements, by sending multiple packets in parallel or out of order.

2.1.2.5. Security

Security is an important communication parameter in many situations. In some distributed systems, critical applications exchange confidential data, whose security is paramount. Because the transmission medium is usually shared, cyptographic techniques need to be used in order to ensure the four standard security requirements: authentication, authorization, confidentiality and non-repudiation.

2.2. The Need for Application-Layer Routing Peer-to-Peer Overlays

The main property of a peer-to-peer architecture is that it connects all the components of a distributed system into a live overlay topology, which consists of point-to-point logical connections or transport-level (e.g. TCP) connections. A live overlay topology, coupled with an application-level routing mechanism presents several advantages for optimizing the communication flows (and access to data), which will be illustrated by presenting 5 scenarios.

Scenario #1 – There is no appropriate option to communicate to a large number of peers

Suppose an application (from now on, we will use the term „peer") needs to communicate to many other peers (tens of thousands of them). The message delivery needs to be reliable, so the most appropriate solution would be to use TCP. What choices does a peer have in order to do that ?

One possibility would be to establish a TCP connection for each message it needs to send to some other peer. Then, after the messages is sent, the connection to the destination peer would be closed. If messages need to be sent frequently, the TCP connection establishment time would introduce a potentially unacceptable delay.

Another possibility, which avoids opening a new TCP connection for each message would be to open a TCP connection to any peer it might ever need to communicate with and keep the connection open. This way, when a message needs to be sent, the connection is already established,

so there is no overhead. However, on a computer there are at most 2^{16} ports, so only that many connections could be open at any time. If that peer needed to communicate to more peers than this, it couldn't. Anyway, operating system have a much lower limit on the number of open files and a TCP connection is treated like a file, so the actual number of open connections would be much less (around a few thousands).

Scenario #2 – Network-layer routing algorithms were not designed to match the communicating peers' goals

The path on which a message is delivered is entirely entrusted to the routers. This implies that the routing protocols running on those routers are the most appropriate for the needs of the communicating peers. However, this is rarely the case. The routing algorithms were designed having goals different than those of the communicating peers.

For instance, suppose that the purpose of the peers is to be able to deliver as many messages as possible to one another, making full use of the available bandwidth. This would require, among other things, an appropriate routing metric and load balancing. There is no guarantee that the routing algorithms use the same metric the communicating peers are interested in. Also, there is no guarantee that the routing algorithms might use any load-balancing. They might route all the incoming messages on the same path, which might have been the most appropriate path at the time the first message was delivered, but which might have become congested due to the large flow of messages routed on it.

Another possible problem with the routing algorithms at the network layer is that some of them are prone to errors. There have been remarkable cases when slow convergence or routing loops have caused significant problems. In a large scale network, no routing algorithm guarantees a 100% rate of success (not any algorithm that we know of), but new more reliable algorithms might have been discovered in the mean time. However, although new, more intelligent algorithms might exist, it is unreasonable to expect a change in the routing algorithms used by every router in the world. Routing algorithms used by routers are resistant to improvement. Once they were set loose, it is very difficult to replace them.

Scenario #3 – Multicasting is a big problem

Multicasting turns out to be a big problem at the network layer. Although solutions like IP multicast do exist, they are not implemented in many places around the world. So, some peers might take advantage of multicast technologies, while others might not.

Efficient multicast algorithms are a current research topic. However, even if some very efficient multicast algorithms are found, it would still be very difficult to implement on every router in the world. Basically, just like in the previous scenario, it is very difficult to change world-wide the algorithms used by routers.

Scenario #4 – One long-distance TCP connection could be slower than a chain of TCP connections connecting the same pairs of peers through other intermediate peers

Suppose that one peer established a TCP connection to a distant peer and the connection has a high error rate and packets are lost and need to be redelivered frequently. Because TCP uses adaptive timers, the time it waits for an ACK is proportional to the time needed for the packet to travel from source to destination. However, the longer the distance the packet needs to travel, he higher the probability that some error might occur (or the packet might be lost entirely). By using such a connection, the transmission speed would be quite low.

Perhaps surprisingly, using more intermediary peers to route the packet might provide a better transmission time. Suppose that, instead of directly connecting to the far-away destination peer, the source peer connects to a peer close-by. This peer to which the source peer connects to

will connect to another peer close-by and so on, until you get to the actual destination. By using a chain of TCP connections, packet losses are detected more quickly. It is like having checkpoints along the initial long distance connection, as opposed to only checking for packet delivery at the ends of the connection.

Scenario #5 – It is impossible to initiate communication with peers located behind firewalls or which use Network Address Translation (NAT)

Suppose that a peer needs to send a message to another peer and in order to do this, it tries to establish a TCP connection to that peer. But that peer might be located behind a firewall which rejects incoming TCP connections. So, the only way for the two peers to communicate by using a TCP connections would be if the destination peer established a TCP connection to the source peer. However, there is no way to tell the destination peer to do that.

A similar situation arises when the destination peer is using NAT (Network Address Translation) in order to communicate with the "outside world". The source peer cannot establish a TCP connection to the destination peer, therefore the sending of messages becomes impossible.

The 5 scenarios presented above should be sufficient to motivate the development of application layer routing algorithms. From the point of view of the OSI reference model, it would be like subdividing the application layer. We would have a communication framework which uses the overlay network (application-level network, as defined before) and provides a communication interface to other applications. The communication framework would be the lower half of the Application layer and the applications using it would reside on the upper half.

2.3. Relevant Existing Peer-to-Peer Architectures

One of the original purposes for developing peer-to-peer architectures was file sharing. Systems like Napster [Napster] or Gnutella [Lua, Crowcroft, Pias, Sharma and Lim, 2005] formed either fully centralized or fully decentralized (and unstructured) peer-to-peer overlays which faciltated the search and retrieval of content. Later, systems like Bittorrent [Bittorrent] and Kazaa [Kazaa] emerged, which focused on improving both the search process and the content delivery. Bittorent uses swarming content delivery. All the peers trying to download the same file are part of the same swarm. In order to download a file, a peer has to contact a central tracker. After that, the peer connects to several randomly chosen peers in the swarm. Files are decomposed into smaller pieces and a client may start sharing the downloaded pieces before downloading the entire file. Bittorent uses a tit for tat scheme which encourages peers to upload files, not just download them.

Kazaa's peers are differentiated into clients and super-peers. Super-peers are chosen automatically based on their computational power, storage capacity and bandwidth. Clients connect to the closest super-peer, using it for file search and download. All of these systems allow keyword-based searches only and their architecture combines centralized and decentralized components.

Content search is performed either by inspecting a centralized directory (e.g. the central bittorrent tracker or Napster's central server) or by flooding the search query in the system (e.g. Gnutella). The first search method is not scalable because of the single point(s) of failure represented by the central component; moreover, the central component may be a bottleneck when there are many peers in the system. The second approach is not scalable because it may generate a lot of undesirable traffic. The flood search may be limited to a fixed hop count, but in this case we have no guarantees that the searched content is actually found.

The first generation of peer-to-peer systems which were used for a somewhat different purpose than content sharing and which had a fully decentralized, yet very structured, architecture, were the distributed hash tables (DHTs). DHTs are decentralized distributed systems that provide a simple API consisting mainly of two functions: *put(key, value)* and *get(key)*. They partition the keys inserted into the hash table among the participating nodes and usually form a structured overlay network in which each communicating node is connected to a small number of other nodes.

CAN (Content Addressable Network) [Ratmasamy, Francis, Handley, Karp and Shenker, 2001] is a DHT which uses a virtual coordinate space represented by a d-dimensional torus. Each node has a unique ID in this space. Considering that there are N nodes in the system, a message reaches its destination after $O(d \cdot N^{1/d})$ hops. The amount of routing information stored by each node is $O(2 \cdot d)$.

In [Stoica, Morris, Karger, Kaashoek and Balakrishnan, 2001], the authors introduce Chord. The identifiers of the participating nodes are 160-bit numbers ordered on a circle modulo 2^{160}. Each node X is connected to its numerical successor and stores a list of $O(log_2(N))$ "fingers" to other nodes. The i^{th} element of this list contains the identifier of the peer which succeeds X by at least 2^{i-1}. A node can reach any other node in $O(log_2(N))$ hops using this list.

Pastry, presented in [Rowstron and P. Druschel, 2001], uses prefix-based routing in order to build an overlay network and route messages within this network. Each node has a 128-bit identifier in a circular space. The identifiers and the keys are interpreted as base B numbers. Any message reaches its destination after $O(log_B(N))$ hops. The amount of routing information stored by each node is $O((B-1) \cdot log_B(N))$.

CAN, Chord and Pastry were the initial DHTs which started a whole new research area. After them, many other DHTs were developed, like Kelips [Gupta, Birman, Linga, Demers and van Renesse, 2003], Kademlia [Maymounkov and Mazieres, 2002] or Viceroy [Malkhi, Naor and Ratajczak, 2002]. A very good survey on peer-to-peer overlay schemes, of which a large part is dedicated to DHTs, is [Lua, Crowcroft, Pias, Sharma and Lim, 2005]. DHTs are very appealling for distributed data storage and retrieval, because every operation requires the traversal of only a small number of hops, with small amounts of routing information. Their drawbacks are the following:

- they require connectivity capabilities between any pair of nodes, so they are less likely to deal appropriately with machines located behind NATs and firewalls
- peers which are close in the overlay network may be very far geographically; thus, latencies may be quite high, although the number of hops is small
- they only support key-based searches

DHTs provided the inspiration and also the building blocks for many of the future advances in peer-to-peer architectures. On one hand, DHTs were used for providing multicast communication services. Scribe [Rowstron, Kermarrec, Castro and Druschel, 2001] is based on Pastry and constructs a single multicast tree for data delivery. Splitstream [Castro et al., 2003] uses multiple Scribe trees.

Multicast communication services based on multicast trees and which do not use DHTs at their core have also been proposed. Such a multicast data delivery system, based on multiple, concurrently used, balanced trees, is presented in [den Burger, Kielmann and Bal, 2005]. A system called ZIGZAG, maintaining a multicast balanced tree, is presented in [Tran, Hua and Do, 2003]. Every peer has $O(K^2)$ degree and the diameter of the tree is $O(log_K(N))$. ZIGZAG has a hierarchical structure and whenever a new peer joins the tree, it contacts the root first, which redirects it to another peer, and so on. Collaborative multiple multicast tree approaches were presented in [Padmanabhan, Wang, Chou and Sripanijkulchai, 2002] and [Venkataraman, Yoshida and Francis, 2006]. In [Cohen and Kaempfer, 2001], the authors consider several optimization objectives for constructing a multicast tree (e.g. a maximum bottleneck multicast tree). Other concerns regarding multicast trees are anonymity [Xiao, Liu, Gu, Xuan, Liu, 2006] or transfer reliability.

On the other hand, the drawbacks of the DHTs motivated researchers to find novel solutions for distributed data storage and retrieval systems. Some form of locality awareness was introduced in DHTs [Wu et al., 2008], in order to reduce the latencies. Key-based searches were extended to range queries by introducing tree-based indexing techniques over DHTs [Zheng, Shen, Li and Shenker, 2006], [Lopes and Baquero, 2007], [Gao and Steenkiste, 2004]. However, in tree-based schemes, the peer storing the tree root is accessed too often (during every operation). Even with replication, the system peers storing the replicas of the tree nodes located closer to the root will be accessed more frequently. An extension of Chord to support range queries was presented in

[Abdallah and Buyukkaya, 2006]. All of these DHT extensions to range queries only support 1D range queries, as DHTs are essentially using only one coordinate value (with the possible exception of CAN). To overcome this limitation, systems supporting scalable multidimensional range queries were proposed in [Hauswirth and Schmidt, 2005], [Gupta, Agrawal and Abbadi, 2005], [Bharambe, Agrawal and Seshan, 2004] and [March and Teo, 2006].

Distributed systems supporting distributed data storage and multidimensional (i.e. multi-attribute) range queries are useful, for instance, in order to develop efficient resource discovery mechanisms. For example, the jobs which are executed in Grids have specific requirements regarding the resources they need (e.g. minimum amounts of available memory and storage space, minimum and/or maximum CPU frequency, minimum number of processors or cores, maximum machine load, and possibly many others). When making a job scheduling decision, a Grid scheduler needs to consider only those resources (e.g. machines) whose characteristics satisfy the requirements of the job. In many scheduling systems this part is trivially addressed by assuming that the scheduler is aware of all the available resources within the system and their up-to-date characteristics, either by subscribing to a notification (and/or monitoring) service or by inspecting a (centralized) catalogue.

These simple approaches can be successfully adopted in distributed systems of small and medium sizes. However, in large Grids, where thousands or tens of thousands of resources are dynamically available and whose characteristics change frequently, it is unfeasible to assume that any single entity (or fixed number of entities) can be aware of all the existing resources within the system. Thus, scalable, decentralized, distributed resource discovery techniques are required (as was noticed in [Hauswirth and Schmidt, 2005] and [Gupta, Agrawal and Abbadi, 2005]). For a critical analysis of middleware for Grids and other distributed systems see [Pop et al., 2009].

All of the currently existing systems which support multidimensional range searches suffer from one or more of the following drawbacks:

- they lack load balancing
- they do not consider data replication (i.e. some data may be lost even if only a single node fails)
- they support only read-only objects
- the peer-to-peer topology is structured in such a way that some queries may be forwarded to too many peers

The peer-to-peer data storage and retrieval system presented in [Andreica, Tîrşa and Ţăpuş, 2009a] (and part of which is described in Chapter 3) overcomes most of these limitations: it uses explicit load balancing (although there are situations in which the technique we chose is not too efficient), it replicates objects on multiple peers, it support both reads and writes (although we only present the read-only case in Chapter 3) and the peer-to-peer topology is naturally mapped upon the multidimensional geometric attributed space.

As we have seen, most peer-to-peer systems that have been developed were focused on data storage, data access, data search and retrieval, or all of these. There have been only few attempts to develop peer-to-peer systems for optimizing (point-to-point or multipoint-to-point) data transfers. Let's note first that any DHT could be used for providing fully decentralized point-to-point communication services. When we want to transfer data from a peer *X* to a peer *Y*, we just transfer it along one of the paths within the DHT (using other DHT nodes as intermediate nodes). The DHT guarantees that the data will eventually reach the peer Y, even though peer X does not know anything about peer Y. However, using the DHTs in this way does not help optimize any communication metric, except for the number of hops. Latency may be arbitrarily high (except, perhaps, in the case of locality-aware DHTs) and the transfer speed may be even lower than in the case of using the best effort transfer service of the Internet.

Perhaps surprinsingly, the Bittorrent peer-to-peer system may be a better starting point for developing data transfer optimization peer-to-peer architectures. Bittorrent is concerned with optimizing the upload and download speeds of the participating peers. Thus, there is no surprise in

the fact that the Bittorrent protocol was used for developing a data transfer optimization service for Grids [Zissimos, Doka, Chazapis and Koziris, 2007]. An agent-based peer-to-peer architecture for data transfers (but for wireless sensor networks) was proposed in [Shakshuki, Hussain, Matin and Matin, 2006]. Optimization of media flows in peer-to-peer overlays using a distributed approach was considered in [Argyriou and Chakareski, 2008].

In the context presented so far, the family of peer-to-peer architectures presented in Chapter 3 and the multicast tree architecture described in Chapter 6 bring multiple original contributions. At first, in Chapter 3, we present a generic peer-to-peer architectural model, which is the first of its kind. Then, we implement this model in a communication framework focused on providing high throughput data transfers and we evaluate its performance in practical settings. The communication framework routes the data over multiple paths in order to improve the throughput, a feature which has not been encountered in any of the peer-to-peer architectures developed so far. Based on the same peer-to-peer architectural model, we develop a fault tolerant peer-to-peer architecture for storing data objects, which efficiently supports multidimensional range queries. This architecture is the first of its kind which supports both object replication and multidimensional range queries.

2.4. System Level Fairness and User Satisfaction in Peer-to-Peer Content Sharing Systems

Most of the peer-to-peer file sharing systems use some kind of mechanism for imposing system-level fairness (i.e. in order to avoid freeriding). Bittorrent is the most popular such system nowadays and it uses the tit for tat mechanism [Zghaibeh and Harmantzis, 2008]. In essence, this mechanism should ensure that the download speed of a user is proportional to its upload speed. Thus, if a user does not upload data, then it shouldn't be able to download it, either. Based on the original Bittorrent protocol many peer-to-peer file sharing applications have been developed. Tribler [Pouwelse et al., 2008] is one such application which emphasizes the social aspect more than other systems of its type.

A distributed mechanism called Bartercast has been developed for Tribler in order to prevent lazy freeriding [Meulpolder, Pouwelse, Epema and Sips, 2009]. This mechanism essentially computes maximum data flows between pairs of peers and is reliable even in the presence of lies (i.e. peers which report that they uploaded more data than they actually did). Bittorrent clients focused on free riding (like Bitthief) have been developed [Locher, Moor, Schmid, Wattenhofer, 2006] and they showed that the tit for tat mechanism can be fooled (when most of the other peers are cooperative, normal, Bittorrent clients). An analysis of several fairness issues in peer-to-peer networks has been performed in [Wong, 2002].

In Chapter 3 we present several techniques which impose system-level fairness at the communication layer. These techniques are complemented by a set of other techniques whose purpose is to enhance the user satisfaction (i.e. to improve the user's perception regarding the quality provided by the system), like the use of distributed path reservations as a means of increasing the data transfer quality. We introduce a novel technique for enforcing fairness at the communication layer by sharing the available incoming bandwidth of each peer proportionally among multiple path reservations. The concept of using path reservations is not new. It has been mentioned in many papers, like [Carbunar, Ioannidis and Nita-Rotaru, 2004], [Jurca and Frossard, 2008], [Sahni et al., 2007] and [Wu and Katzela, 2000]. Moreover, the idea of sharing bandwidth among multiple data transfers under given proportions has also been used before. However, the method we use for sharing the incoming bandwidth among reservations is completely new.

2.5. Network Link and Path Capacity Estimations

The capacity of a network link is the largest bandwidth at which an application may send data over that link. The capacity of a network path is the minimum of the capacities of the network links composing the path. Many end-to-end network path estimation techniques have been presented in the literature, like packet pair/train dispersion, variable packet size, or self-loading periodic streams (see [Prasad, Dovrolis, Murray and Caffy, 2003] for a short survey on this, and

[Lai and Baker, 1999]), each of them having varying degrees of intrusiveness (i.e. generating varying degrees of extra traffic required for computing the estimation).

However, no technique has been proposed for estimating the capacity of the upload link of a machine. Information regarding the upload capacity of a link is useful in many peer-to-peer content sharing systems, for instance, for selecting super-peers (i.e. peers which large upload capcities which could serve other less endowed peers). In Chapter 3 we present the first technique for estimating the upload capacity of a network link. The technique uses the concept of helper peers, which send upload bandwidth estimations based on packet interarrival times back to the source peer.

2.6. Congestion Control Algorithms

Congestion control is an important issue in many data transfer applications. Congestion control is usually implemented at the transport protocol level (e.g. TCP) and its purpose is to avoid the congestion of the network. It is well-known that TCP's congestion control algorithm is too conservative and is unable to fill well network links with large latencies and high bandwidths. Because of this, many alternatives to the TCP protocol have been proposed, most of them modifying only the congestion control algorithm. Scalable TCP [Kelly, 2003], HighSpeed TCP [Floyd, 2003], FAST TCP [Jin, Wei and Low, 2004], TCP Cubic [Rhee and Xu, 2005] are just some of the novel proposed protocols. Application-level protocols based on UDP have also been proposed, e.g. RBUDP [He, Leigh, Yu and DeFanti, 2002], SABUL [Gu, Hong, Mazzucco, Grossman] and UDT [Gu and Grossman, 2007].

The use of multiple parallel TCP sockets has also been considered as possibly one of the simplest alternatives which could be deployed (due to the fact that no support for any new protocol is required) [Sivakumar, Bailey and Grossman, 2000]. A good survey on these alternatives to TCP is [He, Primet and Welzl, 2005].

Most of the efforts in developing these protocols were on overcoming TCP's limitations in the case of high bandwidth – high latency network links and on fairness in regard to standard TCP flows (i.e. the new protocols should not behave too aggressively to standard TCP flows; instead, they should behave as if they had equal priority to standard TCP flows). A direction which has been explored to a less extent is that of developing less-than-best-effort data transfer protocols. Such a protocol assumes that it has the lowest priority and tries not to influence the speed of any other competing communication flow. However, at the same time, such a protocol should try to fill as much of the available upload bandwidth as possible. We can say that we would like for such a protocol to always use only that part of the bandwidth which is not used by the other communication flows.

2.7. Distributed Testing Infrastructures

There are many projects which have tried to tackle the distributed systems performance assessment problem from different perspectives: modeling workloads and simulating their run under various environment assumptions [Bucur and Epema, 2003], [Ernemann, Hamscher, Schwiegelshohn, Yahyapour and Streit, 2002], [Weil and Feitelson, 2001], and creating tools for launching benchmarks/application-specific functionality tests and reporting results like the GridBench project [Tsouloupas and Dikaiakos, 2005] and the NMI [Pavlo et al., 2006] projects.

ServMark (presented in Chapter 3) is the natural complement to these approaches, by offering a much larger application base, more advanced workload modeling features, and the ability to replay existing workload traces. In addition, ServMark can be used for much more than just Grid performance evaluation. ServMark is based on the already existing tools DiPerF [Dumitrescu, Raicu, Ripeanu and Foster, 2004] and GrenchMark [Iosup and Epema, 2006].

2.8. Data Structures for Online Resource Reservations

Many resource reservation and scheduling techniques [Marchal, Primet, Robert, and Zeng, 2005] make use of efficient data structures capable of improving the response time. The simplest one is an array storing the available bandwidth for each time slot, but this takes $O(T)$ time per

operation. The segment tree [Andreica and Ţăpuş, 2008f] and the bandwidth tree [Wang and Chen, 2002] provide a time complexity of $O(log(T))$ per operation, but only for simple requests: for $D=1$ ($D=s_2-s_1+1$) we need a range maximum (minimum) query operation, together with a range addition update. A dynamic version of an augmented segment tree is proposed in [Brodnik and Nilsson, 2002] and a linked-list data structure is presented in [Xiong, Wu, Xing, Wu and Zhang, 2005].

The segment tree and the principles behind the block partitioning technique have been known for a long time. However, no general purpose algorithmic framework (like the one presented in Chapter 4) has been published so far. Moreover, our framework is equally focused on supporting range updates and range queries, while most previous papers seem to have ignored the range update capability.

None of the data structures we mentioned (or that we are aware of) can efficiently support the complex requests that the *Time Slot Group* data structure (presented in Chapter 4) does. All of them exhibit linear ($O(T)$) or superlinear ($O(T \cdot log(T))$) time for at least one of the two operations, while our data structure takes sublinear time for both.

2.9. Real-Time Centralized Scheduling of Data Transfers

A scheduling framework for real-time data transfers has been proposed in [Cîrstoiu, 2008] and [Cîrstoiu, Voicu and Ţăpuş, 2008]. The framework considers that it has full control over the underlying network infrastructure. Bandwidth reservations are placed across all the links of a path along which a data transfer occurs. The actual data transfer is performed using the FDT software [FDT]. The architecture of the data transfer scheduling framework presented in Chapter 4 is an extension of the one presented in [Cîrstoiu, 2008] and [Cîrstoiu, Voicu and Ţăpuş, 2008]. We added new components, like prediction and simulation components, and we also developed new techniques for these components (e.g. using neural networks for pattern detection and forecast, like we did in [Andreica, Cătăniciu and Andreica, 2009]). Moreover, all the scheduling algorithms and data structures introduced in Chapter 4 are new – they are more complex than the data transfer scheduling methods used in [Cîrstoiu, Voicu and Ţăpuş, 2008].

A problem which is similar to our centralized deadline-constrained data delivery optimization problem (presented in Chapter 4) was considered in [Chen and Primet, 2007]. The main difference is that the data is not split into individual packets, but rather viewed as being continuous. For their problem, the authors of [Chen and Primet, 2007] provide a polynomial-time solution. An online deadline-constrained data transfer scheduling problem similar to ours and to the offline problem studied in [Chen and Primet, 2007] was addressed in [Eltayeb, Dogan and Ozguner, 2004], where several heuristics were proposed. [Soldati, Zhang and Johansson, 2009] discusses a deadline-constrained data transfer scheduling problem in wireless networks, in which a solution based on network flows in a time-expanded graph is presented. We also use a time-expanded graph for our centralized, real-time deadline-constrained data transfer scheduling problem from Chapter 4.

Mutual exclusion scheduling problems with different constraint graphs (trees, permutation graphs, comparability graphs) were considered in [Jansen, 2003] and [Baker and Coffman, Jr., 1996]. These problems are similar to the ones we considered in Chapter 4 for scheduling the data transfer from the same batch. However, the existing solutions only considered requests with unit duration and a common deadline. The algorithms we presented introduce variable durations and start and finish time constraints into mutual exclusion scheduling problems.

2.10. Offline Optimal Scheduling of Constrained Point-to-Point Communication Flows

The high multiplicity scheduling problem for files with divisible sizes studied in the section 5.1 is related to the problems of job scheduling and resource allocation [Pruhs, Sgall and Torng, 2004]. In [McCormick, Smallwood and Spieksma, 2001], an $O(P \cdot log(T))$ high multiplicity multiprocessor scheduling algorithm is presented for $C=2$, given the maximum value for the makespan T, but the sizes are not considered to be divisible. Another related problem is bin-packing with variable bin sizes and divisible item sizes [Coffman, Jr., Garey and Johnson, 1987] or the

multiple knapsack problem [Chekuri and Khanna, 2000], for which several objectives have been considered (minimizing the number of bins, minimizing costs). Related to the bin packing problem is the knapsack problem with divisible item sizes, for which a polynomial time algorithm was presented in [Verhaegh and Aarts, 1997].

The problem discussed in section 5.2, regading the (high multiplicity) scheduling of two communication flows, is related to the flexible job shop problem [Jansen, Mastrolilli and Solis-Oba, 2000], in which there are several jobs, each of which composed of a number of operations. The operations of a single job must be executed sequentially on any of the m machines given. There are important differences, however. The problem we considered is a high multiplicity scheduling problem and all the operations (packets) that compose a job (flow) are identical.

TCP buffer management strategies have been proposed in many papers, for optimizing different performance metrics [Ocher and Cohen, 1998], [Goldenberg, Kagan, Ravid and Tsorkin, 2005], [Afsahi and Dimopoulos, 2002]. As far as we know, TCP sender buffer management has not been addressed from the perspective discussed in section 5.5. However, by reinterpreting the parameters of that perspective, we obtain the (uncapacitated) economic lot sizing problem, which has been studied extensively (under different variations) in many papers [Bitran and Yanasse, 1982], [Federgruen and Tzur, 1991], [Wagelmans, van Hoesel and Kolen, 1992], [Wagner and Whitin, 1958]. Optimal $O(n \cdot log(n))$ algorithms were proposed in the literature. Although we only developed an $O(n \cdot log^2 n)$ algorithm, this algorithm is simpler to implement than other approaches. This can be noticed by the fact that the pseudocode presented can be immediately turned into a program in many programming langiuages. Besides, the techniques employed by the algorithm are of interest by themselves, as they can be used for solving other problems, too.

2.11. Offline Optimal Multicast Strategies

Communication scheduling in networks with tree topologies was considered in many papers (e.g. [Erlebach and Jansen, 1997], [Henzinger and Leonardi, 2003]) and the optimization of content delivery trees (multicast trees) was studied in [Cui, Xue and Nahrstedt, 2004]. Optimal broadcast strategies in trees in the single-port model have been studied in [Slater, Cockayne and Hedetniemi, 1981] and [Cohen, Fraginaud and Mitjana, 2002]. In [Koh and Tcha, 1991], the problem was enhanced with non uniform edge transmission times and an $O(n \cdot log(n))$ algorithm was proposed. The single-port model refers to the fact that at every time moment, a tree node can only send a message to (at most) one of its neighbors. We extended the single-port model with sending and receiving constraints in Chapter 6 and we developed a new set of algorithmic techniques for this case.

A dynamic programming algorithm was presented in [Birchler, Esfahanian and Torng, 1996] for the minimum time broadcast in directed trees, under the single port line model. Reliable multicast strategies and reliable multicast trees have been the object of many research papers, like [Miloucheva, Reyes, Mahnke and Jonas, 2006], [Kinoshita, Shiroshita and Nagata, 1998] and [Lai and Liao, 2001].

The k-station placement (k-SP) problem (equivalent to the k-hop multicasting problem discussed in section 6.2) was investigated in [Galdi, Kaklamanis, Montangero and Persiano, 2001] and was solved in polynomial time for directed trees. The time complexity of the algorithm presented in [Galdi, Kaklamanis, Montangero and Persiano, 2001] is $O(k \cdot n \cdot M(n))$, where $M(n)$ is the fastest min-cut algorithm on a graph with n vertices and $O(n)$ edges. $M(n)$ is $O(n \cdot log(n))$ [Borradaile and Klein, 2006] and the time complexity of the algorithm in [Galdi, Kaklamanis, Montangero and Persiano, 2001] is $O(k \cdot n^2 \cdot log(n))$, but the implementation is cumbersome. Our solution is quite easy to implement and has a better time complexity, of only $O(k \cdot n^2)$.

Some other types of (multicast) transportation strategies in trees were presented in [Andreica, Briciu and Andreica, 2009].

2.12. Offline Optimal Replica Placement in Tree-Like Networks

Center, diameter and longest path problems have been considered in [Ben-Moshe,

Bhattacharya, Shi, Tamir, 2007], [Das and Pas, 2008] and [Lan, Wang and Suzuki, 1999], where efficient algorithms were proposed for several weighted and unweighted center problems on cacti. Our algorithms from Chapter 7, however, are derived from the dynamic programming solutions on trees and may be more easily extended to other problems which have a dynamic programming solution on trees. In [Yen and Chen, 2006], a linear time algorithm for the unweighted connected k-center problem in trees was given. A framework for centrality problems on trees (which resembles the framework we proposed to some extent) was introduced in [Rosenthal and Pino]. Center problems on path networks were considered in [Halman, 2003].

Reliability metrics have been proposed before in many research papers (e.g. [Wood, 2001] and [Weichenberg, Chan and Medard, 2004]), but, as far as we are aware, none of them uses the unrestricted vertex multicut as a subproblem for computing the metric's values. The $O(V \cdot H)$ solution for the Unrestricted Vertex Multicut Problem on trees was presented in [Guo, Huffner, Kenar, Niedermeier and Uhlmann, 2006]. Other papers [Gottlob and Lee, 2007] studied the vertex multicut problem, but their focus was on making a distinction between classes of problems which are solvable in polynomial time, and not on developing efficient polynomial time algorithms.

Equitable colorings of trees have been studied either explicitly [Chen and Lih, 1994], [Jarvis and Zhou, 2001] or by solving scheduling problems [Baker and Coffman. Jr., 1996]. In [Chen and Lih, 1994], the equitable coloring of trees with a minimum number of colors was studied and a polynomial time algorithm was proposed. In [Jarvis and Zhou, 2001], the authors present an $O(n^3)$ algorithm for the equitable k-bounded vertex coloring of trees with n vertices. In [Baker and Coffman. Jr., 1996], the authors try to minimize the total number of colors used, subject to limitations like the maximum number of vertices colored with the same color and they present a linear time algorithm for trees, but are not necessarily interested in obtaining an equitable coloring. Equitable colorings on graphs with bounded treewidth have also been studied [Bodlaender, and Fomin, 2004], but so far the known polynomial algorithms are only of theoretical interest. Our equitable coloring algorithm from Chapter 7 brings a fresh perspective on the class of tree (equitable) coloring problems.

Data placement in networks should also consider the network's reliability, which is an important property of a distributed system. Failure detection techniques have been developed [Dobre, Pop, Costan, Andreica and Cristea, 2009], in order to decide when a component fails and when a replica should replace a failed component. In order to make informed decisions regarding network upgrades (with the purpose of optimizing either fault tolerance or the access time to the data), network analysis techniques are important. In [Andreica, Ungureanu, Andreica and Andreica, 2009], efficient algorithms for classifying critical edges relative to matchings and to minimum spanning trees are given.

Chapter 3 – Peer-to-Peer Architectures and Techniques for Data Transfer and Retrieval Optimization

This chapter presents a family of peer-to-peer architectures, models and techniques used for the online optimization of point-to-point communication flows, as well as for optimizing the storage and retrieval (search) of distributed data objects. The structure of the chapter is the following. At first, a generic peer-to-peer architectural model is introduced, after which two peer-to-peer communication frameworks based on this model are presented. One of them follows the architectural model closely, while the other one is based on a hybrid architecture (making use of both centralized and decentralized components). Afterwards, two objectives relevant in peer-to-peer content sharing systems, fairness and (user perceived) QoS, are considered, for which novel techniques, implementations and experimental results are presented.

Section 3.5 introduces another fully decentralized peer-to-peer architecture, whose purpose is to store data objects and allow efficient retrieval (multidimensional range search) of the stored objects. This architecture follows the architectural model from section 3.1 but differs in purpose from the other two peer-to-peer systems based on the same model: it does not optimize general point-to-point communication flows, but rather communication flows generated by data access requests. In the end of the chapter, a new upload bandwidth estimation technique is introduced, which is quite useful in peer-to-peer content sharing systems, as well as a distributed testing frameworks for distributed systems (peer-to-peer systems in particular).

The original contributions presented in this chapter were published in [Andreica, Dragomir and Ţăpuş, 2009], [Andreica, Borozan, Bălăceanu and Ţăpuş, 2009], [Andreica and Ţăpuş, 2009d], [Andreica and Ţăpuş, 2009f], [Andreica, Tîrşa and Ţăpuş, 2009a], [Andreica, Tîrşa and Ţăpuş, 2009c], [Andreica, Legrand and Ţăpuş, 2007] and [Andreica et al., 2006].

3.1. The Design of a Generic Peer-to-Peer Architecture

This section presents a generic peer-to-peer architectural model which will be used in future sections for developing peer-to-peer systems for optimizing (multi)point-to-point data transfers, as well as storage and retrieval of data objects.

3.1.1. Properties of the Peer-to-Peer Architecture

The peer-to-peer architecture must have the following properties:

- every peer must know only a small subset of other peers
- using local decisions only, we must be able to reach any peer Y, starting from any (other) peer X
- every peer must make message routing decisions based on local information only (i.e. information which can be obtained from its neighbors or from itself)

In order to provide all these properties, every peer will be assigned the coordinates of a point in a d-dimensional space. Every peer can generate its own coordinates, using d different hash functions. The arguments of the hash functions can be any combination of IP address, MAC address, time of day, application-specific information, randomly generated information, and so on. By using hash functions on a large enough number of bits, we can expect, with a high probability, that the set of coordinates of each peer will be unique.

Every peer X will periodically broadcast its existence to all the peers which are located at most $BR \geq 2$ hops away from X in the peer-to-peer topology (BR is the broadcast radius). Every peer X will maintain a set $I(X)$ containing the peers Y which broadcasted their existence to X during the past $T_{limit,1}$ seconds. Periodically, based on the peers in the set $I(X)$, peer X will choose its set $ND(X)$

of desired direct neighbors. For each peer Y in $ND(X)$, peer X will initiate a new connection to Y (or will mark the connection as desirable and announce this to peer Y, if such a connection is already open).

We denote by $N(X)$ the current set of direct neighbors of a peer X. If a connection from X to a peer Y in $N(X)$ has not been marked as desirable by any of the two peers (X and Y) during the past $T_{limit,2}$ seconds, the connection will be closed. It should be obvious that the sets of peers $I(X)$ and the structure of the peer-to-peer topology are co-dependent. The set $I(X)$ depends on the current structure of the topology and the structure of the topology changes based on the information in $I(X)$. We also draw attention to the fact that the connections we speak about are only logical connections. From an implementation point of view, they may correspond to one or several (open) TCP connections between the same pair of peers X and Y, or to UDP based communication.

The part concerning the way a peer X chooses its neighbors from the set $I(X)$ was left unspecified. This is because we want to construct a general framework and allow the possibility of using any neighbor selection method. However, the chosen method must create a topology which satisfies the 3 properties we mentioned. Several neighbor selection methods are presented next, after which the joining and leaving processes of the peers are described.

Note that the described model ignores NAT and firewall issues and assumes that any peer may initiate a connection to any other peer.

3.1.2. Neighbor Selection Methods

A desirable property of a neighbor selection method is to be convergent, i.e. considering that no new peer joins the system and no old peer leaves the system, after a finite amount of time every peer X should reach a state where it will not change its set of neighbors $N(X)$ anymore. Let's denote by $co(X,j)$ the j^{th} coordinate assigned to peer X.

In the first method, for every dimension j ($1 \le j \le d$), every peer X sorts the peers Y in $I(X)$ according to $co(Y,j)$. Then, peer X splits the peers Y from $I(X)$ into two sets: $A(X,j)$ contains the peers Y with $co(Y,j) \le co(X,j)$ and $B(X,j)$ contains those peers Y with $co(Y,j) > co(X,j)$. Peer X will connect to the $K \ge 1$ peers Y from each set $A(X,j)$ and $B(X,j)$, whose j^{th} coordinates are closest to $co(X,j)$ (thus, at most $2 \cdot K$ such peers). We call this method the *Independent Dimensions* method.

A second method (the *Hyperplanes* method) consists of choosing a set of H hyper-planes passing through the origin of the geometric coordinate system. These hyper-planes define Q regions in the d-dimensional space (all the points in the same region are in the same half-space relative to each hyper-plane). Each peer X translates itself to the origin and classifies all the peers Y in $I(X)$ according to the regions in which they are located, relative to the peer X. The desired neighbors of peer X are the (at most) $K \ge 1$ closest peers from each of the Q regions (closeness is measured using a distance function like, for instance, the L_1, L_2 or L_∞ norm).

Another method is based on computing the local Voronoi diagram, using any distance function (e.g. any L_p norm, $1 \le p \le \infty$). Peer X computes the (d-dimensional) Voronoi diagram of the points assigned to the peers in the set $EI(X) = I(X) \cup \{X\}$. We define by $V_{local}(X,Y)$ the Voronoi cell of a peer Y from $EI(X)$. The set of desired neighbors of peer X is composed of those peers Y in $EI(X)$ such that $V_{local}(X,Y)$ touches (is adjacent with) $V_{local}(X,X)$. This method stabilizes after a finite amount of time and every Voronoi cell $V_{local}(X,X)$ converges towards the Voronoi cell $V_{global}(X)$ of the point assigned to peer X, where $V_{global}(X)$ is computed by considering all the peers in the system (not just those in $EI(X)$). This *Local Voronoi* method has several advantages upon other methods, but the computation of the Voronoi diagram in $d \ge 3$ dimensions is a tedious task. Other methods, based on distributed geometric spanner construction [Farshi and Gudmundsson, 2005], can also be used.

3.1.3. Joining and Leaving

When a peer X joins the system, it must know at least one other peer which is part of the system. The peers it initially knows form the initial set $I(X)$. Based on this set, peer X chooses an initial set of neighbors $N(X)$. From now on, peer X is within the system and will gradually change

its neighbors until the topology stabilizes.

When a peer *X* leaves the system, it will no longer broadcast its existence to the neighboring peers (located at most *BR* hops away from *X*). Thus, after $T_{limit,1}$ seconds from the moment *X* leaves the system, no other peer will consider *X* as a potential (direct) neighbor.

3.2. An Implementation of the Peer-to-Peer Generic Architecture for Data Transfer Optimization

In this section we describe a communication framework which was developed and implemented according to the principles of the peer-to-peer architectural model introduced in the previous subsection. The main aspects which will be covered are:

- name space and topology construction and maintenance
- the kind of routing information that is being maintained by the peers
- the message routing algorithms used
- methods for estimating the available bandwidth on the connections between peers

3.2.1. Name Space and Topology Construction and Maintenance

We consider that the communication network is composed of one or more communication nodes (or cells). Each cell may have zero or more names. A name is simply a string of characters which must not contain the special character '*'. The names are coordinates in a metric space where the distance between two names is the lexicographic distance. The names assigned to a cell are important in establishing the topology of the overlay network. For each name it has, the cell tries to establish direct connections to the *K* closest cells having a name lexicographically smaller and the *K* closest cells having a name lexicographically larger.

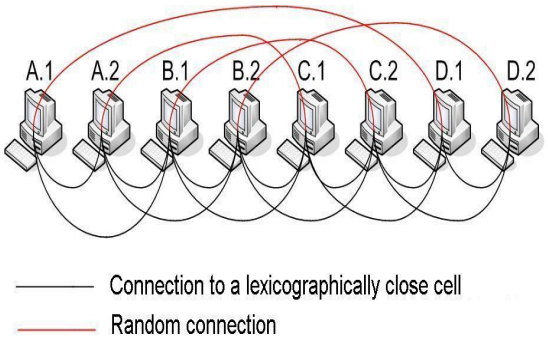


Fig. 3-1. A Possible Overlay Network Topology for K=2, M=1 and P=0.

Because some of these cells might not be directly accessible, because of NATs or firewalls, the *K* closest directly accessible cells are chosen. A cell also tries to connect to *M* other cells, chosen randomly from the set of known cells, and to *P* of the known cells which are very distant in the overlay network, in order to maintain a well-connected network. Fig. 3-1 shows a possible network topology for *8* cells, *K=2*, *M=1* and *P=0*.

The applications using a given cell for communication purposes may add a new name or remove one of the old names. They can do this based on their current communication interests. If a group of applications expects to communicate intensely, they may add names which are close to each other in the metric space, thus bringing the corresponding cells closer in the overlay network. These names may also be added in order to reflect geographic proximity and make the cells establish low latency connections, by using the reversed DNS name or a hierarchically structured name (for instance, "country.city.institution. machine_name").

The names of the cells form a 1D geometric space (using the terminology form the Section 3.1). A cell may have multiple names, thus effectively corresponding to multiple points in the 1D space.

3.2.2. Routing Information

Routing information is exchanged between cells by sending routing update messages at regular intervals. These messages are sent to all the cells located at a distance of at most R hops in the overlay network, where R is the neighborhood radius. There are two types of routing update messages: updates which advertise the state of the direct connections and updates advertising other known peers in the network.

The first type of messages contains information about the measured available bandwidth of each direct connection. This information is used by each cell in order to build a detailed view of their neighborhood.

The second type of routing update message contains information about other cells in the overlay network that the sending cell knows about. For each known peer, also called destination peer, a set of cells from the neighborhood of the sending cell is sent. The peers in this set are candidate nodes when choosing an intermediate cell in the routing process. For each peer in the set an estimation of the overall available bandwidth of all the paths towards the destination peer is also sent. Based on this information, the receiving cell computes the available bandwidth on all the paths from the sending cell to the advertised cell which do not contain any other node in the neighborhood of the receiving cell. The names of the advertised cells and the computed information are stored in a trie.

In order to avoid the propagation of every piece of information to every node in the system, each cell allows only a maximum number of T entries in the trie. Once this threshold is reached, either new entries or already existing entries should be discarded. The algorithm we used allows, in fact, for a maximum of $f \cdot T$ ($f>1$) entries in the trie and periodically selects only the T most significant such entries, deleting the others. Newer entries, entries which have a shorter common prefix with all other entries at the moment of insertion and entries which are frequently used in the routing process are preferred.

3.2.3. The Routing Algorithm

3.2.3.1. Routing Unicast Messages

Routing unicast messages is a two step process. First, the name of the destination is looked up in the trie. Here one of the peers which are located in the neighborhood of the cell routing the message and which advertised paths towards the destination is chosen. This choice is non-deterministic. A peer i is chosen as an intermediate cell according to the formula

$$P_i = AB_i / SAB \,. \qquad \textbf{(3-1)}$$

P_i is the probability to choose the i^{th} peer, AB_i is the advertised available bandwidth of peer i towards the destination and SAB is the sum of the available bandwidths of all the candidate peers. Only peers having a name which is lexicographically closer to the destination than any of the names of the cell making the routing decision are considered as candidate peers, in order to make sure that the lexicographic distance towards the destination decreases. In case the name is not found in the trie, the name which is lexicographically closest to the destination is chosen. This operation can be performed very efficiently using the trie data structure.

The second step consists of choosing a path towards the intermediate peer selected in the first step. In order to make this decision, the knowledge about a cell's neighborhood is interpreted as a directed graph. The vertices of the graph are represented by the cells of the network and the edges of the graph are represented by connections between these cells. Each edge has an associated capacity equal to the measured available bandwidth of that edge. The graph is directed because the measured available bandwidth might differ when measured from each of the two end points of each edge.

Considering, in turns, each cell in the neighborhood as a sink and the current cell as a source, the Edmonds-Karp algorithm is used in order to compute the maximum flow in the corresponding flow network. After computing the flow, a maximum of D paths from the source to the sink are

selected and stored. Each path is chosen considering only the graph's edges which have a positive amount of flow. For each such path, the minimum amount of flow *f* on any of the edges on the path is computed. This will be the capacity of that path. The quantity *f* is then subtracted from the amount of flow existing on each of the graph's edges which are part of the path. The next path is computed considering the new amounts of flow on the graph's edges, and so on. The process of selecting a path from the current cell to an intermediate cell located in its neighborhood consists of non-deterministically choosing one of the paths. Each path *i* is chosen with a probability computed according to the formula

$$PA_i = Cap_i / SumCap. \quad \textbf{(3-2)}$$

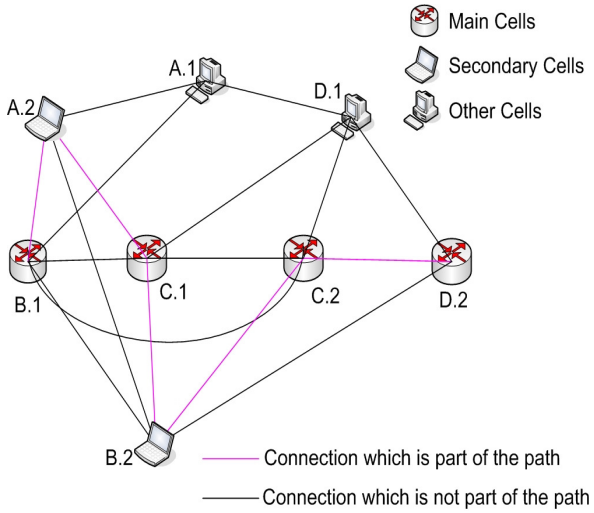


Fig. 3-2. A Possible Path from B.1 to D.2.

PA_i is the probability to choose the i^{th} path, Cap_i is the capacity of the i^{th} path and *SumCap* is the sum of the capacities of all the selected paths towards the intermediate cell. The message is then routed along this path. The next cell making a routing decision is the last cell of the selected path.

The path a message follows to the destination is composed of two types of cells, chosen in the two steps of the routing process. The first type is represented by the main cells and they are the ones making routing decisions. The names of these cells get increasingly closer to the destination's name, as the cells are located further away along the path. Between two consecutive main cells on the path there may be several secondary cells. The secondary cells only pass the message forward and are not subject to naming restrictions. However, all the secondary cells between two consecutive main cells *A* and *B* are in the neighborhood of the cell *A*.

The separation into main and secondary cells is based only on the actions taken when routing a message along a path. Any cell could be a main cell for some messages and a secondary cell for others. Fig. 3-2 shows a possible path a message could follow between the cells *B.1* and *D.2*. The path contains the cells *B.1*, *A.2*, *C.1*, *B.2*, *C.2*, *D.2*, in this order. The network topology is the one presented in Fig. 3-1 and the radius *R* of a cell's neighbordhood was considered to be *1* hop.

3.2.3.2. Routing Multicast Messages

The presented architecture can also route multicast messages, although this is not its main purpose. A multicast message is sent to a destination of the type *"PREFIX*"*, meaning that the message must reach all the cells having a name starting with the prefix *"PREFIX"*. Routing such a message consists of two stages. In the first stage, the message is routed as a unicast message. The message is being routed towards cells having an increasingly longer common prefix with the destination.

Eventually, the message will reach a cell having the prefix *"PREFIX"*. When such a cell is reached, the second stage begins. The multicast message is broadcasted towards all the neighbors of the cell which have a name starting with the prefix *"PREFIX"*. These neighbors will broadcast the message further to other peers having a name beginning with the string *"PREFIX"*, and so on.

Because of the way the topology is built, all the cells sharing the same common prefix will try to form a connected subgraph in the overlay network and the message will reach all of the destinations.

3.2.4. Measuring the Available Bandwidth

We need to measure the available bandwidth for every direct connection. In order to do this, we will consider every connection to be a cylindrical pipe, having a certain length and a certain cross section. The "length" *L* of the pipe will be measured in milliseconds. The cross section of the pipe has an area *A* and will be measured in kilobytes per second. The volume of the pipe, which is the length multiplied by the area, is measured in kilobytes. The capacity of the pipe is its cross-section *A* and is the value we want to determine.

In order to do this, *L* is periodically estimated based on the round-trip times of several small messages (a few bytes in size). Then, when sending a "large" message having *X* bytes through the pipe, its "length" will be *X/A*. The time it takes for the whole message to reach the other side will be *(L+X/A)*. Fig. 3-3 shows a message having *X* bytes being sent on the connection. The measured *RTT* (round-trip time) will be equal to twice the value of *(L+X/A)* and the desired value of *A* will be

$$A = \frac{X}{\frac{RTT}{2} - L}. \qquad \textbf{(3-3)}$$

However, if *X/A* is less than *1*, the *RTT* will be approximately equal to $2 \cdot L$ and the denominator in *(3)* would be close to *0*. In order to achieve a good result, we would need a value of *X* for which *X/A* is significantly larger than *1*. We can use binary search for *X*, between a lower limit of *1* byte and some specified upper limit. An upper limit of *3* megabytes for X is enough even for connections having a bandwidth of *10* Gbps. With the binary search, we are looking for values of *X* for which the *RTT* is not too close to $2 \cdot L$, aiming for an *RTT* value close to $F \cdot L$ ($2.2<F<4$). Binary search messages are sent at regular intervals.

The value of the estimated available bandwidth is computed with the following formula:

$$AB_{new} = t \cdot AB_{old} + (1-t) \cdot AB_{computed}. \qquad \textbf{(3-4)}$$

t is a real number between *0* and *1*, $AB_{computed}$ is the value of the available bandwidth computed based on the size and *RTT* of the most recent message, according to *(3)*, AB_{old} is the previous estimation of the available bandwidth and AB_{new} is the new estimation of the available bandwidth.

Because the value of the available bandwidth might largely fluctuate over time, we do not use a tight binary search. Instead, when we need to increase the lower limit of the binary search, we will slightly increase the upper limit, too. Similarly, when decreasing the upper limit of the binary search, we will also decrease the lower limit a bit. Doing so, we provide a solution for the case when the value of the available bandwidth suddenly moves outside the current range of the binary search.

The algorithm estimates the available bandwidth of the TCP connection, as seen from the application level. It does not try to measure the end to end available bandwidth outside the context of the running application. The measurements are affected by the TCP buffer sizes and by the time the messages spend in the message queues. In order to achieve higher throughput, several TCP tuning techniques should be used, as mentioned in [Lee et al., 2001]. These techniques are outside the scope of this book.

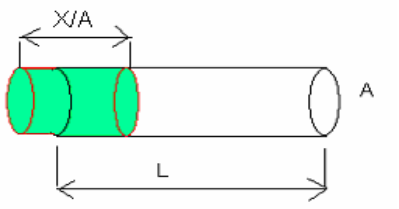


Fig. 3-3. Sending an X Bytes Message on the Connection.

3.2.5. Experimental Results

The communication framework was implemented using the Java programming language. The implementation has been tested using only a small number of cells. Our test scenarios consisted of *8* cells running on *8* different machines, located at *3* different sites. *2* machines were located at the California Institute of Technology (Site 1), *3* machines were located at CERN (European Center for Nuclear Research), in Switzerland (Site 2) and *3* machines were located at the Polytechnic University of Bucharest, in Romania (Site 3). 2 machines (one at Site 2 and one at Site 3) were located behind firewalls. At the beginning, every machine had a single name and knew the IP and port of one or two other peers. The neighborhood radius was chosen to be *2* hops.

The *2* communication cells located at the first site were named: Site1.Host1 and Site1.Host2. The *3* cells located at the second site had a similar naming model, with names ranging from Site2.Host1 to Site2.Host3. The machines at Site 3 were named following the same pattern, ranging from Site3.Host1 to Site3.Host3. Every communication cell tried to connect to the *2* closest cells having a name lexicographically smaller, the *2* closest cells having a name lexicographically larger and another randomly chosen cell. All the communication cells were started roughly at the same time.

The test cases consisted of sending trains of packets. We considered *3* test scenarios. In the first one, *10.000* messages each having *25.000* bytes were sent from Site1.Host1 to Site3.Host1. Fig. 3-4 shows the way the total amount of bytes received varied in time. As can be noticed, all the *250* million bytes were received within a time interval of approximately *3* minutes, achieving an average transfer rate of nearly *1.4* MB/s, with a peak rate of about *2.3* MB/s during the first minute. For comparison purposes, we also transferred *250* million bytes using *SCP* (Secure copy), which achieved a peak rate of *210* KB/s.

Except for the *SCP* test, we also wanted to know what the transfer rate would be if only the direct connection between Site1.Host1 and Site3.Host1 was used. In order to achieve this, we generated a second test scenario, where only Site1.Host1 and Site3.Host1 were started. Fig. 3-5 shows the variation with time of the total amount of bytes received by Site1.Host1. We achieved a constant transfer rate of approximately *280* KB/s. The transfer rate was noticeably higher when using all the *8* cells, proving that our balanced message flow distribution could be a feasible technique for achieving high throughput data transfers.

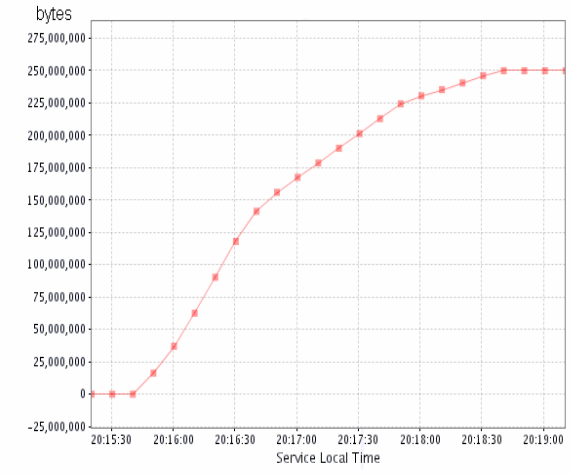


Fig. 3-4. The Variation with Time of the Total Amount of Bytes received by Site3.Host1 during the First Test Scenario.

In the third test scenario we wanted to understand if the message routing overhead was significant. For this, we sent *10.000* messages each having *25.000* bytes between two communication cells located on two machines at Site 2 (Site2.Host1 and Site2.Host2). The transfer rates ranged from *6* MB/s to *9* MB/s. We also transferred *250* million bytes between the *2* machines using *SCP*. With *SCP*, the transfer rate reached a peak of approximately *11.3* MB/s. Since in this

scenario most of the messages were routed along the direct connection between the *2* machines, the difference between the transfer rates of our system and those achieved by *SCP* was caused by the routing overhead.

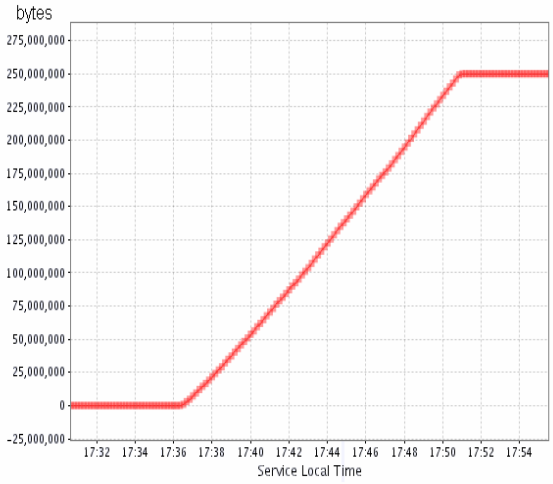


Fig. 3-5. The Variation with Time of the Total Amount of Bytes received by Site3.Host1 when using only the Direct Connection to Site1.Host1.

Other test scenarios consisted in testing the multicast message delivery, but the amount of transferred data was quite small and the tests were only intended to verify the correctness of the implementation.

The test results showed that our implementation worked well in a small distributed system. However, the good behavior cannot be extended to large distributed systems without further testing. The test results were also insightful, because we noticed that the routing overhead may be significant in some situations.

3.3. A Hybrid Implementation of a Peer-to-Peer Data Transfer Optimization Framework

This section introduces another peer-to-peer architecture for optimizing end-to-end data transfers between its peers, by forwarding messages over multiple paths. The architecture mixes centralized components (whose functionality could have been implemented in a distributed manner only with some difficulty) with a decentralized behavior. We assume that the peers are located on machines throughout the Internet and, thus, information about and the control of the underlying physical network links is impossible (particularly as other applications with unpredictable traffic patterns share the same medium).

The main components of the architecture are:

- *Central Tracker* - maintains information about all the peers in the system and the content stored by them
- *Peer* - it participates in the data transfer process

The main components of a peer application are: the Communication Module, the Routing Module and the Data Handling Module.

The Central Tracker initially contains information about the locations (IP addresses and ports) of a set of (*K*) servers distributed throughout the world. These servers will be used as landmarks. When a peer joins the system, it contacts the *Central Tracker* and obtains the information regarding these servers. It then measures its ping times towards each of these servers and computes an identifier (*ID*) based on these ping times - there are many possibilities, ranging from a point in a *K*-dimensional space to a simple combination (e.g. maximum, average) of the *K* ping times. In our application we chose the simple case of computing their average. After a peer computes its *ID*, it announces the *Central Tracker*. The Central Tracker may slightly adjust the *ID* (in case it is already used by a different peer) and then it communicates the final value of the ID to

the peer. Afterwards, the association between the ID and the IP address and port of the peer is stored on the tracker.

A data transfer request consists of transferring certain data from a source to a destination. The request is submitted by the destination node. We will consider here a slightly more general case, in which, instead of the source, the request specifies a piece of content to be downloaded. Then, the central tracker is queried and the destination peer obtains a list of all the source peers which contain the requested piece of content. Thus, the data transfers will in fact be multipoint-to-point. The peer application on the destination node is responsible for splitting the desired content into smaller pieces and requesting each piece separately from one of the sources. Piece requests are sent from the destination to one of the sources directly.

Once a peer receives a piece request message, it grabs the requested data into memory and then the data is passed to the routing module. The routing module will choose a peer to which the data will be sent - that is, the content is not necessarily sent directly to the requesting peer. The routing module works as follows. It receives data from the communication module (received from another peer) or is passed data from the user application. The data packets processed by the routing module are of two types: *CONTENT* and *ACK*. Every data piece that arrives at the routing module has an associated destination identifier.

For a piece of data of type *CONTENT*, the routing module first checks if the current peer is the final destination of the data. If the current peer *P* is the final destination, then the data is passed to the user application and an *ACK* is sent back to the (last) peer which forwarded the data to *P*. The *ACK* will follow the reverse path of the piece of data and will eventually reach the source of the data. If *P* isn't the final destination of the data, then it contacts the central tracker in order to obtain a list of "neighbors" of the current peer *P*. This list can alternatively be requested from the tracker periodically and maintained locally.

In order to define what a neighbor *Q* of a peer *P* is, let's consider that *X.ID* denotes the identifier of a peer *X*. Let *destid* be the identifier of the final destination of the data. The distance *dist* between *P.ID* and *destid* is computed (in our case, the absolute difference between the two identifiers). The neighbors of *P* are those peers *Q* such that the distance between *Q.ID* and *P.ID* belongs to the interval *[a·dist, b·dist]* and *Q.ID* is closer to *destid* than *P.ID*. In the implementation, we used *a=0.15* and *b=0.85*. However, every peer may use its own values of *a* and *b* and may even update them any time it considers appropriate. We decided that the peers *Q'* at a distance smaller than *a·dist* are too close to *P* and would induce too many intermediate hops towards the destination. On the other hand, the peers *Q''* at a distance larger than *b·dist* may be too close to the destination and, thus, the transfer may not benefit completely from the multi-path property. Among the "neighbors" of the current peer *P*, we will choose the one having the smallest average *ACK return time* towards the destination for the given data size.

Every peer *P'* maintains the values *ACKRT(P'.ID,R'.ID,D)*, representing a weighted average of the time durations between the moment when a piece of size *D* was received by *P'* in order to be sent towards the destination *R'*, and the moment when the corresponding *ACK* was received. If we have some neighbors *Q* for which *ACKRT(Q.ID,destid,D)* was not computed, yet (because no *ACK* was received back), then we will choose one of these neighbors as the next intermediate peer (in order to give them a chance). We also implemented the option of not using *ACKs*. In this case, one of the neighbor peers is chosen randomly as the peer to which the data piece will be forwarded. If no neighbor satisfying the specified restrictions is found, then the current peer *P* sends the data directly to the destination peer.

When the routing module of a peer *P* receives a piece of data from another peer *Q*, it stores an association between the identifier of the piece of data (every piece of data has a unique identifier), its size *D* and the peer *Q*, together with a time stamp and the destination identifier of the data. Then, when an *ACK* corresponding to the data piece is received from another peer (or generated by peer *P* if it is the final destination of the data piece), then peer *P* will forward the *ACK* to the peer *Q* from which the original data piece was received.

When receiving an ACK, peer *P* will update *ACKRT(P.ID,R.ID,D)* as

pa(P.ID,R.ID,D)·ACKRT(P.ID,R.ID,D)+(1-pa(P.ID,R.ID,D))·currACKRT, where *R* is the destination of the original piece of data, *currACKRT* is the difference between the current time moment and the timestamp of the association stored when the data piece corresponding to the *ACK* was received and *D* is the size of the original piece of data. *pa(P.ID,R.ID,D)* is a weight factor, between *0* and *1*, and its value may be changed dynamically.

When a peer *P* forwards a data piece (not an *ACK*) to another peer *Q*, it will increment by *1* a counter *cnt(P.ID,Q.ID,D,destid)* representing the number of data pieces of size *D* whose destination identifier is *destid* which were sent from peer *P* to peer *Q* and for which no *ACK* was received. When an *ACK* is received from a peer *Q* corresponding to a data packet of size *D* sent to a destination *destid*, the corresponding counter *cnt(P,Q,D,destid)* is decremented by *1*. Then, when routing a new piece of data of size *D* towards a destination *destid*, peer *P* will disconsider as neighbors all the peers *Q* for which *cnt(P.ID,Q.ID,D,destid)* (or a combination of the values *cnt(P.ID,Q.ID,*,destid)*) is above a threshold set by peer *P*.

A peer uses the following strategy for requesting a piece of content from the list of sources owning the requested content. It logically splits the content into pieces of equal size *D* (e.g. *256 KB* chunks), except possibly for the last one. Then, it starts by requesting *C≥1* different pieces from each source node (or possibly fewer from the last node from which a request is made). As soon as a piece arrives from a source node and not all the pieces have been requested, a new unrequested piece is requested from that source node. This way, whenever a piece is received, *C-1* other pieces are on the pipeline to the considered source node. Source nodes which transfer the content at higher speeds to the destination will be asked data pieces more frequently and will thus send more data overall.

Note that *C* is a parameter which may be different for every peer. Moreover, the same peer may modify the parameter *C* in real-time. For instance, it may use a value *C(i)* for every source node *i*. It could start with small values *C(*)*; then, the destination could increase slightly the values *C(i)* corresponding to those source nodes *i* with high bandwidths and high latencies (in order to fill the pipeline better).

When a peer wants to send a message to another peer, it first checks if it has an open TCP connection to that peer. If it doesn't, then a new TCP connection is opened. Then, whenever a message is sent or received along a TCP connection, a *lastUsed* variable associated to the connection is updated (i.e. set to the current time moment). When a TCP connection has not been used for a time duration larger than one chosen by the peer, the connection is closed.

The described peer-to-peer architecture was implemented in the Java programming language. *4* peers were started on *4* different computers, *3* of which were located in Bucharest (denoted by *E*, *T* and *P*) and one of them located at the Technical University of Delft, in Netherlands (denoted by *O*). We used a video file (*video.mp4*), with a size of *30.696.688* bytes (approx. *29.3 MB*). We chose a file which was large enough to emphasize the improvements that our architecture can bring. The file was split into pieces of *256 KB* each (except possibly for the last chunk).

We computed the data transfer duration under several conditions. The duration was computed as the time difference between the moment when the file was requested and the moment when the last piece of the file was received. At first we measured the duration of the data transfer between the computers *E* and *O* using *SCP* (Secure Copy). *SCP* is not one of the fastest data transfer applications, particularly because it needs to encrypt and decrypt the data, but we considered that it was sufficiently popular to be a relevant baseline for our application. Then, we started the central tracker on computer *E*, where we also started the source peer. We measured the duration of the data transfer from *E* to *O* using our application, without using any other intermediate peers. Then, we progressively added the peers *T* and *P* and re-ran the test each time. These peers were used as intermediate peers. Due to the limited network testbed we decided not to use *ACK* messages. We used *C=1* in all the tests.

Fig. 3-6 presents the time required to transfer the video file from *E* to *O* in the following conditions. At *x=0* we have the duration of the *SCP* data transfer (*253* sec). At *x=1* we have the duration of the transfer between *E* and *O* using our application (*285* sec). We notice that the

duration is higher than in the case of *SCP*, mostly due to the lack of code optimizations. At *x=2* we used one extra peer and at *x=3* we used two extra peers.

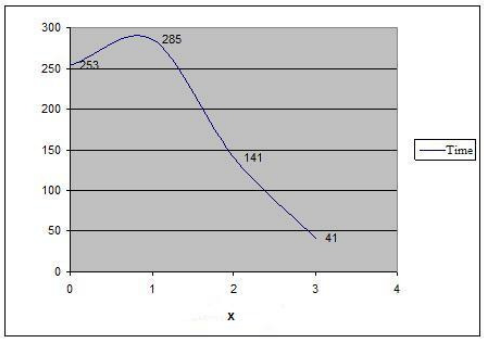


Fig. 3-6. Data Transfer Duration under Various Circumstances.

The test results indicate that the application scales well and improves the duration of the data transfers significantly, even under the restricted conditions of our network testbed.

3.4. Fairness and QoS Enhancement Models and Techniques for Peer-to-Peer Content Sharing Systems

In this section we consider peer-to-peer content sharing systems and focus on two objectives which are very important in such systems. The first objective is concerned with system level fairness. The system's participants which contribute more to the well-being of the system should have more privileges than other participants. The second objective concerns user-level (perceived) quality of service. The users of the system will only use it if the quality of service of accessing (and transferring) the content found within the system is (perceived to be) good enough. Thus, the experienced QoS of the users is an important aspect of a peer-to-peer content sharing system.

We will conceptually divide the functionality of a peer-to-peer content sharing application into two layers: the application layer and the communication layer. In the rest of this section we will present several novel techniques for optimizing the system level fairness and the user's (perceived) quality of service, which are applicable at the communication layer. We also developed application layer techniques addressing the same two objectives in [Andreica, Borozan, Bălăceanu and Ţăpuş, 2009], but they were not included in this book.

The peer-to-peer network is considered to be completely distributed and each peer knows only of those peers with which it has previously established contact. The connections between neighboring peers in the overlay network consist of (bidirectional) TCP connections, for reliability purposes. The network allows peers to join or leave the system at any time, and without prior notice. The communication (and data transfer) between peers is performed through other intermediate peers of the overlay network. In order to transmit information from a peer *A* to a non neighboring peer *B*, the message will travel on a path of directly connected peers:

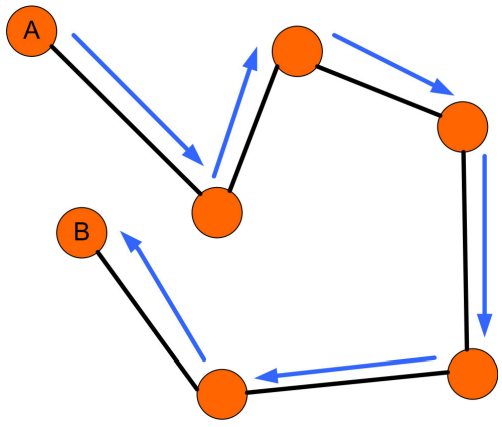


Fig. 3-7. A Path from Peer A to Peer B in the Overlay Network.

3.4.1. QoS Enhancement through Path Reservations

The method proposed for enhancing the quality of the data transfers is that of using path reservations. Every peer *X* of the system has a reservation budget *rv(X)* and every data connection *DC* between two peers *X* and *Y* has a reservation budget of *re(DC)* (the value is stored both by *X* and *Y*). When a peer *X* wants to transfer data from/to a peer *Y*, we will initiate a path reservation request from peer *X* towards peer *Y*. The reservation request has a unique identifier *rid* and a priority *prio(rid)*. The system needs to find a path from *X* to *Y* for which the reservation budgets of each connection and peer on the path is at least *prio(rid)*. In order to find this path we will use a flood search method. Peer *X* sends the reservation request to every neighbor *Z*, which then sends it further, until it eventually reaches peer *Y*.

When a peer *A* receives a reservation request *rid* from a neighboring peer *B* on a connection *DC*, it first checks if the request was received before (in which case it is dropped immediately). If it is a new reservation and $rv(A) \geq prio(rid)$, then peer *A* creates a *partial reservation*, sets $prev(A,rid)=(p=B, c=DC)$ and decreases *rv(A)* by *prio(rid)*. Then, peer *A* will send the reservation request to all of its neighbors (except *B*). When a peer *A* wants to send a reservation request *rid* further to a peer *C*, it will choose a connection *DC* between *A* and *C* s.t. $re(DC) \geq prio(rid)$ (among all the connections satisfying this constraint, it may choose the one with the largest, smallest or p^{th} largest/smallest current reservation budget). If such a connection *DC* exists, then a partial reservation for *DC* is created by peer *A*. *re(DC)* is decreased by *prio(rid)* and then the reservation request is sent to peer *C* (on any of the connections between *A* and *C*, not necessarily on the connection *DC*).

Every partial reservation has an associated timer (set by the peer creating the reservation). If the reservation *rid* is not confirmed until the timer expires, then the partial reservation is cancelled (i.e. if it was a partial reservation for a connection *DC*, then $re(DC)=re(DC)+prio(rid)$; if it was a partial reservation for a peer *A*, then $rv(A)=rv(A)+prio(rid)$).

When a peer *A* wants to modify the reservation budget of a connection *DC* between *A* and *C*, it must do so in an atomic manner, since the connection is shared by both peers *A* and *C*. Peer *A* will ask peer *C* if it agrees with the proposed modified value of *re(DC)*. If a positive answer is not given within a specified time interval (e.g. because peer *C* also wanted to modify *re(DC)* at the same time), then: *(1)* if peer *A* wanted to create a new partial reservation (or transform one into a full reservation), then peer *A* either retries later (up to a maximum number of retrials) or just gives up (in case of giving up, peer *A* does not send the reservation request further to peer *C*, or the confirmation message to anyone else); *(2)* if peer *A* wanted to cancel a partial (or full) reservation, then peer *A* will retry after a specified time interval (cancellations of any reservations must succeed as long as *DC* is an active connection).

Eventually, the reservation request will either *get stuck* at some peers, or will arrive to peer *Y*, which will send a reservation confirmation message back to *X*. The confirmation message will be sent along the *prev* "pointers", i.e. when the confirmation of the reservation *rid* reaches a peer *A*, it will change its partial reservation (of peer *A*) into a full reservation. Then, if peer *A* previously created a (currently non-expired) partial reservation *rid* on a connection *DC* towards a peer *C*, it will change the partial reservation *rid* of the connection *DC* into a full reservation. After successfully doing this, *A* sends the reservation confirmation message to the neighboring peer *prev(A,rid).p* along one of the connections towards this peer. The confirmation will arrive back at peer *X* and a full reservation of a path between *X* and *Y* is made.

A problem which might occur is that some partial reservations along the path from *Y* to *X* expire before receiving the reservation confirmation message, while others are changed into full reservations. Full reservations also have an associated timer. If no data corresponding to the reservation is sent during the first TR_1 seconds after creating the full reservation, then the full reservation is cancelled (just like a partial reservation). After the first message corresponding to a path reservation is sent along a full reservation, then the timer is cancelled and another timer is started, which will cancel the full reservation only if no data is sent on it for TR_2 consecutive seconds ($TR_2 > TR_1$). TR_1 and TR_2 are chosen by the peer *A* creating the reservation, the way *A* sees

fit. After the path reservation is created, data can be sent to/from *X* from/to *Y* along the path. When all the data is sent, a reservation cancellation message is sent from peer *X* to peer *Y* along the path. When this message traverses a full reservation, the full reservation is cancelled (and the corresponding reservation budgets are increased accordingly).

We will now discuss what exactly a peer reserves through a path reservation. Ideally, when negotiating a reservation, the requesting peer would ask for a specific bandwidth from each of the peers on the route to the destination peer. Throughout the life of that reservation, the bandwidth would remain free in case the peer that created the reservation wanted to use the path. However, this kind of reservation presents serious problems. Consider the network in Fig. 3-8, where peer *A* wants to reach peer *E*, and peer *B* wants to reach peer *D*. Both peer *A* and peer *B* are connected to peer *C*, but on the same Ethernet link. This means that the bandwidth each peer requests actually affects the bandwidth available to the other. More problematic is the fact that peer *C* is unaware that it is linked to a router before having access to peers *E* and *D*, and thus the bandwidth it measures to peer *E* and the bandwidth to peer *D* are not disjoint. They again restrict each other.

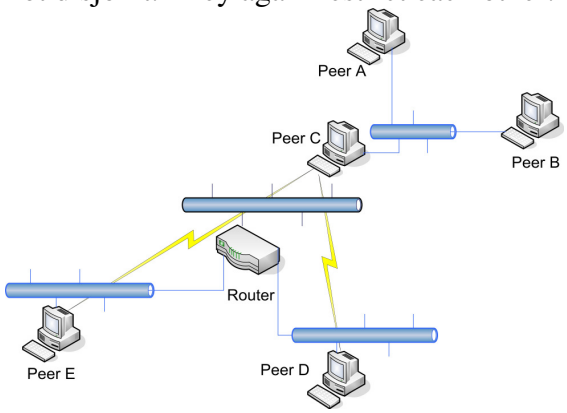


Fig. 3-8. Non-disjoint Paths at the Physical Level.

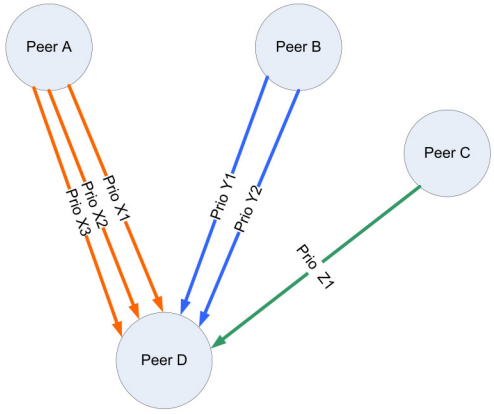


Fig. 3-9. Multiple Reservations sharing the Same Connections.

In order to avoid this problem, we decided that each reservation represents a kind of percentage from the available bandwidth in between peers, whatever that bandwidth may actually be. So, peer *A* does not request for *X* MB/s on each link in between peers along the reservation path, but that on each link in between peers, its data transfer priority is somehow proportional to *X*. *X* is not a percentage, but is turned into one according to the following example. Consider Fig. 3-9. The lines of the same color represent reservation paths that contain the corresponding connection. More precisely, reservations of the same color transmit data on the same sockets, in the direction pointed by the arrows (outgoing sockets are not shown).

At every peer *D*, every socket *S* will be assigned a priority *Pr(D,S)*=the sum of the priorities of the reservations which send data to *D* along the socket *S* (thus, only the incoming direction is considered). The reservation budget of a connection (socket) or peer is imposed such that the sum of all socket priorities does not surpass a certain value so that a scale of the importance is

maintained and, additionally, in order to ensure a somwehat equitable transfer even to small priority data reservations. Every socket *S* (on which data is sent towards peer *D*) is assigned a percentage by peer *A*: *P(D,S)=Pr(D,S)/SumPr*, where *SumPr* is the sum of the priorities of all the sockets of which peer *D* is an endpoint. In the example from Fig. 3-9 we consider the sockets S_X, S_Y and S_Z (colored with orange, blue and green). We have:

$$P(D,S_X)=\frac{PrioX1+PrioX2+PrioX3}{PrioX1+PrioX2+PrioX3+PrioY1+PrioY2+PrioZ1} \quad \textbf{(3-5)}$$

If an orange reservation (in Fig. 3-9) has priority *Prio X2*, this means that the sum of all the data transferred through the reservation from peer *A* to peer *D* (along the socket S_X) is maintained proportional to *Prio X2*, relative to $Pr(D,S_X)$.The algorithm that achieves this will be described next.

3.4.2. Limiting the Incoming Bandwidth of the Reservations

In order to ensure fairness (i.e. proportional bandwidth according to the priority of the reservation), we need to limit the bandwidth of some reservations. Considering that data both enters and leaves the peers following a predetermined, reserved path, we should decide mainly if we should limit the incoming bandwidth of a reservation, the outgoing bandwidth or both. We chose to restrict the incoming bandwidth of a reservation, due to several reasons.

Statistically, the download speed of a peer (the incoming bandwidth) is greater than the upload speed (the outgoing bandwidth). If we decided not to limit the incoming data, there is a possibility of receiving more data than a peer is capable of storing (even temporrarily), leading to potential overflows. In this sense, restricting the incoming data acts like a flow control mechanism. Flow control is not considered in any other ways in this section. Thus, it makes sense to impose limitations on the faster link (which is the incoming link). Note also that the main considered scenario is that of average computers, which have only one connection to the Internet. Thus, there is only one physical link on which all the data arrives and on which all the data is sent further.

3.4.3. The Significance of Reservation Priorities

Ideally, a user should be able to set the priority of its reservations only up to an upper limit which quantifies its contribution to the system. For instance, in a content sharing system, users which send (upload) more data to other users should be able to set higher priorities to their data transfers (i.e. to their downloads). Thus, reservation priorities can be seen as a sign of "social status". Alternatively, in a scientific community, high priorities represent critical data transfers which need to be allotted a larger share of the bandwidth than the other data transfers. From an implementation point of view, if the maximum value of a reservation is limited and increases (or decreases) with time, there should be an entity which supervises the actions of the users and decides their merits. This entity could be easily implemented as a central server, where every user needs to login and authenticate. It seems more difficult to implement such mechanisms in a distributed manner, although several such attempts have been made [Meulpolder, Pouwelse, Epema and Sips, 2009].

3.4.4. Implementation Details and the Fairness Decisions

We implemented a prototype of the described communication architecture in the Java programming language. We constructed an unstructured peer-to-peer overlay network, in which every peer connects to at most $KP \geq 1$ other peers, known in advance. The main modules of the application are: the Communication module, the Data module, and the Fairness Decisions module.

The Communication module is responsible with the transfer of data to and from the peer and is designed as a separate thread handling socket read, write and accept operations. Being built using the "non-blocking sockets" paradigm, it is the only thread to access the Selector object and its keys. It also contains methods for the creation of new SocketChannels to link the current peer to other peers in the network. Peers are interconnected through *K* separate channels (*K* may be different for each pair of peers), all of them "non-blocking". The channels are used both for sending data messages and operational messages (reservation requests and confirmations). Alternatively, we

could reserve one (or more) of the channels between every pair of connected peers *X* and *Y* for sending and receiving operational messages only. This way, operational messages will never be delayed by data messages (or by the decisions of the Fairness decision module, which will be presented next).

The Fairness Decisions module tries to ensure a certain degree of fairness among the data transferred through a peer. The sum of the priorities of the reservation paths res_i passing through a peer *A* cannot exceed a predetermined value *rv(A)*. This requirement is checked by the module when trying to establish a new (partial) reservation. Based on the current sum of the priorities of all the full reservations related to a peer ($\sum_{i=1}^{n} prio(res_i)$) and on the priority of each reservation *i* ($prio(res_i)$), the module tries to ensure that in a certain period of time *T*, the amount of data transferred through that reservation is at most:

$$Data(res_i) = \frac{prio(res_i)}{\sum_{i=1}^{n} prio(res_i)} \cdot InBw \cdot T \qquad \textbf{(3-6)}$$

where *n*=total number of reservations on any socket at the beginning of the period of length *T* and *InBw* is the (current) total incoming bandwidth of the peer.

The Fairness Decisions module implements a repeating algorithm taking place throughout the lifetime of the peer, at the end of each period of *T* seconds. Let's consider that the periods of length *T* are assigned consecutive numbers starting from *1*, according to their chronological order. Let $P_{sock\_i,q} = \sum_j prio(res_j)$, where *j* iterates over all the existing reservations which are sending data on the socket *sock_i* to the peer *A*, at the beginning of the q^{th} period. Let $x_{sock\_i,q} = f(P_{sock\_i,q}, InBw_q)$ = an estimation of the maximum allowed amount of data (bytes) that should be transferred by the reservations on the socket *sock_i* during the q^{th} period; $InBw_q$ is the incoming bandwidth available for the application at the beginning of the q^{th} period (we do not consider here the problem of estimating $InBw_q$). Let $y_{sock\_i,q}$=the exact number of bytes transferred by all the reservations on *sock_i* during the q^{th} period. The peer's total *available* buffer space is an upper bound on the sum of the values $y_{sock\_i,q}$.

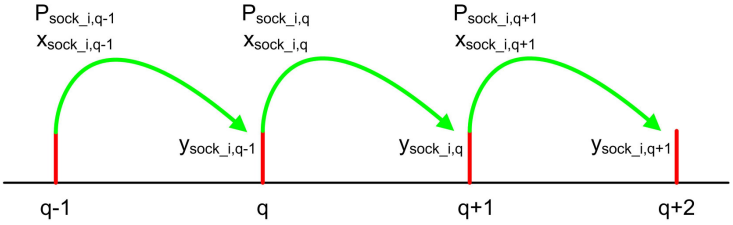


Fig. 3-10. Computation Steps of the Fairness Decisions Module (for Each Socket *sock_i*).

At the end of the q^{th} ($q \geq 1$) period, $x_{sock\_i,q}$ and $y_{sock\_i,q}$ are compared. If $y_{sock\_i,q} > fr \cdot x_{sock\_i,q}$ (where *fr* is a fraction which tolerates a small excedent, e.g. *fr=1.05*) then we compute $z_{sock\_i,q}$=the number of consecutive periods (starting from the period *q+1*) during which no more data will be read from the socket *sock_i*. Alternatively, during every period *q*, we can refuse to read more than $x_{sock\_i,q}$ bytes from the socket *sock_i* (by setting the read buffer size appropriately; thus, $y_{sock\_i,q} \leq x_{sock\_i,q}$). Then, if $x_{sock\_i,q}$ (total) bytes are read during the period, we unregister the *read* operation of *sock_i* from the Selector (for the rest of the period) and re-register it at Step 2 (described below), at the end of the q^{th} period. We consider a virtual, separate, incoming socket at a peer *P* for every reservation whose origin is at peer *P* (thus, not reading data from such a virtual socket for a time period means not accepting data send calls from the application layer, or blocking such calls until new data can be accepted).

From an implementation point of view, the socket *sock_i*'s read operation is unregistered from the *Selector* for $z_{sock\_i,q}$ periods. (i.e. we ignore events regarding data arrival on the socket *sock_i* for $z_{sock\_i,q}$ periods). In the Java source code, the steps of the Fairness Decisions module are implemented in the same thread in which the selector is running. Periodically, a timer sets a Boolean variable *runFD* to *true* and then wakes up the Selector. The Selector is woken up either by

the timer or when an interesting event on one of the registered sockets occurs (e.g. new data arrives on a socket, a socket is ready for writing new data on it, or a new connection request is received). When the Selector is woken up, it handles all the remaining read, write and accept operations (if any) and, afterwards, the boolean variable *runFD* is checked. If it is *true*, it runs the steps of the Fairness Decisions module. The steps taken at the end of the q^{th} period are summarized below:

Step 1. For every *active* socket *sock_i* (a socket whose *read* operation is registered at the Selector) verify if data reception on the socket should be delayed; if yes, then compute $z_{sock\_i,q}$ and unregister the socket's *read* operation from the Selector.

$$z_{sock\_i,q} = \left\lceil \frac{y_{sock\_i,q} - x_{sock\_i,q}}{x_{sock\_i,q}} \right\rceil \quad \textbf{(3-7)}$$

Step 2. Verify for which of the previously delayed sockets we should re-register their *read* operation to the Selector (i.e. their delay expired at the end of the current period); re-register the read operation to the Selector for these sockets.

Step 3. For every active socket *sock_i* (including those whose *read* operation was just re-registered at Step *2*):

3.1) (re)compute its priority for the next period $q+1$, $P_{sock\_i,q+1}$; note that new reservations may have been created on this socket and old ones may have been cancelled.

3.2) compute $x_{sock\_i,q+1} = max\{(min\{InBw_{q+1} \cdot T,\ TotalBuf_{q+1}\} \cdot P_{sock\_i,q+1} / SumPsock_{q+1}) - OccBuf_{sock\_i,q+1},\ 0\}$, where $SumPsock_{q+1}$ is the sum of the priorities $P_{sock\_j,q+1}$ of all the active sockets *sock_j*, $TotalBuf_{q+1}$ is the total amount of temporary buffer space (both available and occupied) during the $(q+1)^{st}$ period, and $OccBuf_{sock\_i,q+1}$ is the amount of buffer space occupied at the end of the q^{th} period by data received on the socket *sock_i* and not yet sent further (or consumed by the application layer, if the current peer is the data's destination). If $x_{sock\_i,q+1}=0$ then the *read* operation of the socket *sock_i* is unregistered from the Selector until the end of the next period.

3.4.5. Experimental Results

The testbed consisted of two machines located at the Politehnica University of Bucharest (peers *A* and *B*), one machine at the University of Craiova (peer *C*), and another one at the Technical University of Delft, in Netherlands (peer *D*). All these machines had public IP addresses. We also used two other peers with private IP addresses, located in Bucharest, Romania (peers *E* and *F*).

An unstructured peer-to-peer overlay with all the *6* peers (*A*-*F*) was constructed. The peers were connected on a path (in the order *A, B, D, E, C, F*). Peer *E* had the lowest download bandwidth, of about *55* KB/s. We considered $InBw_q$ to be constant for all the peers and known in advance (from a configuration file). We initiated a reservation from peer *A* to peer *E*, with priority *10*, and computed a transfer speed of nearly *50 KB/s*. Then, we initiated another reservation, from peer *F* to peer *E*, with priority *x* ($1 \le x \le 20$). After initiating the reservation, we estimated the transfer speed of the first reservation (as an average over the following *120* seconds). The results are shown in Fig. 3-11: the transfer speed of the first reservation varied according to the priority of the 2^{nd} reservation. We used $T=2$ seconds.

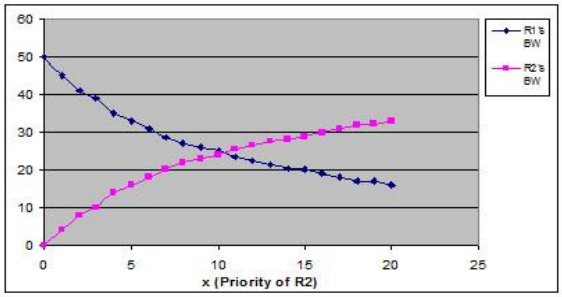


Fig. 3-11. The Bandwidths of Reservations 1 and 2 (R1 and R2).

3.5. A Fault-Tolerant Peer-to-Peer Object Storage Architecture with Multidimensional Range Search Capabilities

This section presents a new peer-to-peer architecture for storing and retrieving data objects. The retrieval mechanism is particularly efficient and the architecure supports multidimensional range reporting queries (i.e. find all the objects whose properties belong to a given multidimensional range). We will first present the functional requirements of such an architecture and then we will discuss the way the peer-to-peer topology is constructed and maintained. In the end, we will present simulation results for evaluating the performance of the proposed architecture.

This solution was published in [Andreica, Tîrşa and Ţăpuş, 2009a], where we presented the full functionalities of the architecture, including how objects are managed, modified and replicated, and how load adaptation occurs (the load of a peer *Y* was defined as the ratio between the total size of the objects assigned to *Y* and *Y*'s total storage space; nevertheless, the size of the objects and the storage space can be replaced by any other metrics, and many other functions for computing the *load* may also be used). In this section we will only consider the restricted case of *read-only* data objects and *load-unaware* topology, in order to better focus on the processes taking place within the peer-to-peer architecture. Moreover, we will assume that, once inserted, objects are never removed by the user applications.

3.5.1. Functional Requirements

The objects (data items) stored by the system have a fixed number $D \geq 1$ of *index* properties (numbered from *1* to *D*). For simplicity, we will assume that every index property has numeric values (although we could use any value type, as long as a total order exists for the values of the property's type). Every object must also have a unique identifier (which may be one of the *D* index properties). The unique identifier and the values of the index properties must be read-only. Besides these, an object may have any number of read-only data fields.

The system must provide an external interface (API) to the user applications which should be independent of its internal structure. The API should allow the user applications to:

- add an object inside the system
- retrieve all (or at most *M* of) the objects whose index property values are within a given orthogonal range and for which the values of the data fields and of the index properties satisfy a given *filtering condition*

We expect the user applications to handle the objects as follows. They can create objects of their own and handle them to the system for storage. They can retrieve (copies of) all (or at most *M* of) the objects within a given (orthogonal) range and read their index properties or data fields. The range is given as a cartesian product of *D* intervals, corresponding to each of the *D* index properties. Moreover, a range search may specify a *filtering function*. The filtering function is a boolean function, whose value depends on the values of the data fields and index properties of the searched objects. Thus, the values of the index properties of a returned object are within the given range and the filtering function returns *true* when applied on the values of the data fields and index properties of the object.

The system is composed of any number of peers which are interconnected in an overlay topology which is not exposed to the user applications. A user application should be able to use any peer as an interface to the system. In fact, for any action it wants to perform, a user application should be able to use any peer and get the same response. This way, no dedicated resources (machines) are required for the user interface and no performance bottlenecks are introduced.

3.5.2. The Peer-to-Peer Overlay Topology

The peer-to-peer overlay topology must have the same properties of the generic peer-to-peer architectural model presented in Section 3.1. We will restate some of those properties here, in order to maintain the completeness of the presentation, and we will introduce new, specific properties,

that the peer-to-peer overlay topology of this system must have.

The internal structure of the system is composed of a peer-to-peer topology. Every peer maintains open connections (e.g. TCP or „logical" connections) with a subset of other peers which are called its *neighbors*. We denote the set of neighbors of a peer X at a time t by $N(X,t)$. We denote the *extended neighborhood* of a peer X, composed of all the peers located at a distance of at most R hops from peer X at time t, by $EN(X,R,t)$ ($X \notin EN(X,R,t)$). Thus, $EN(X,1,t)=N(X,t)$. We will assume that every peer can connect to any other peer if it so wishes. Thus, we ignore NAT and firewall issues on purpose. Furthermore, we assume that connections never fail permanently as long as the two connected peers are still within the system (e.g. if a connection fails, it can be restored after a finite amount of time; if a connection fails permanently, it is because at least one of the two connected peers has left the system).

Every peer X is assigned a point $coord(X)$ in the D-dimensional space of the objects' index property values (i.e. a set of D coordinates); $coord(X,j)$ is the coordinate in dimension j (i.e. corresponding to the j^{th} index property). These coordinates are self-generated, using D hash functions (whose arguments are a combination of random parameters and parameters which are specific to peer X). The obtained hash values are then mapped and scaled to the corresponding ranges of the D dimensions. Note that the ranges of the values in each dimension are scaled to the same range $[0,VMAX]$. Thus, the property space is $[0,VMAX]^D$. Some of the parameters mentioned from now may be local to the peers (i.e. every peer may use its own values for these parameters); such a parameter P will be marked by $P^{(*)}$ when it is introduced; the local parameters of a peer X may have constant values or X may use its own strategies for assigning values to these parameters according to real-time conditions. All the other parameters are global parameters (i.e. all the peers must use the same values for these parameters).

The overlay topology must have two main properties. The first property is the following: starting from any peer X in the system, we must be able to find the peer $closestPeer(Q)$ which is closest (in coordinate space) to a given query point Q. We may use any distance function we consider suitable. To be more precise, let's consider the following scenario. Peer X receives a request at time t for finding the closest peer to a given query point Q. As long as the closest peer to Q was not reached, the request is forwarded from the current peer to one of its neighbors (or one of the peers in $EN(X,RD_1,t)$, where $RD_1^{(*)} \geq 1$ is a small value). Thus, the request passes through several (other) peers and eventually reaches peer Y whose coordinates are closest to Q than those of any other peer in the system.

Every peer X maintains a set of peers based on which it modifies its local connections. We denote by $I(X,t)$ the set of peers maintained by peer X at time t. Periodically (every $TP^{(*)}$ seconds), peers announce their existence (and current coordinates) to every peer in their extended neighborhood $EN(X,RD_2,t)$ (where $RD_2(*) \geq 1$ is a small value). To make the analysis simpler, we will consider that time advances in discrete time steps and that peers announce their existence at every time step. Thus, at time step t, peer X sends an announcement to all the peers in $EN(X,RD_2,t)$. $I(X,t+1)$ consists of all the peers which announced their existence to peer X at time step t.

Afterwards, peer X chooses its set of neighbors $N(X,t+1)$ based on the set of peers $I(X,t+1)$. Peer X will then close the connections to the peers in $(N(X,t) \setminus N(X,t+1))$ (except for those peers Y belonging to $(N(X,t) \setminus N(X,t+1))$ for which $X \in N(Y,t+1)$; $N(X,t+1)$ is modified by also adding to it these peers Y) and open connections to the peers in $(N(X,t+1) \setminus N(X,t))$. We point out the obvious fact that there is a co-dependency between the sets $N(X,*)$ and $I(X,*)$: $N(X,t)$ is computed based on $I(X,t)$ and $I(X,t)$ depends on the announcements received by peer X from the peers in its extended neighborhood $EN(X,RD_2,t-1)$ (which depends on the sets $N(*,t-1)$). From a practical point of view (where time does not advance in discrete time steps), $I(X,t)$ will be composed of all the peers Y which anounced their existence to peer X within a time interval $[t-TLIMIT,t]$ (where $TLIMIT^{(*)}$ is carefully chosen).

The second property of the overlay topology of the system must be that, considering that no new peers join the system and no old peers leave the system, there must exist a finite time moment t_0, such that for every $t > t_0$, the set of neighbors $N(X,t)$ of a peer X does not change anymore, i.e. the

topology eventually converges to an equilibrium. The equilibrium can only be disturbed when a new peer joins the system or when an old peer leaves it. When a new peer X joins the system at time t, it knows the identity of (at least) one peer Y which is already within the system. Thus, X connects to peer Y (updating the sets $N(X,t)$ and $N(Y,t)$). After this initial connection, peer X will gradually change its neighbors, until a new equilibrium is reached. When an old peer X leaves the system, the process is even smoother. Since it does not announce its presence to the peers in its extended neighborhood, peer X will not be considered as a potential neighbor in the near future.

We will now consider the *routing* problem (starting from a peer X which receives a routing request at time t, we must reach the peer *closestPeer(Q)*, for a given point Q), for which we chose the following algorithm:

1. peer X computes $dist(X,Q)$=the distance between its coordinates and the point Q
2. if there is a peer $Y \in N(X,t)$ such that $dist(Y,Q)<dist(X,Q)$, then choose any peer $Y \in N(X,t)$ with $dist(Y,Q)<dist(X,Q)$ and forward the request to Y (or choose the peer $Y \in N(X,t)$ with the smallest value $dist(Y,Q)$ and with $dist(Y,Q)<dist(X,Q)$)
3. otherwise, if there is a peer $Y \in EN(X,RD_3,t)$ (where $RD_3^{(*)} \geq 1$ is a small value) such that $dist(Y,Q)<dist(X,Q)$, then choose any peer $Y \in EN(X,RD_3,t)$ with $dist(Y,Q)<dist(X,Q)$ and forward the request to Y (or choose the peer $Y \in EN(X,RD_3,t)$ with the smallest value $dist(Y,Q)$ and with $dist(Y,Q)<dist(X,Q)$)
4. otherwise, we decide that $X=closestPeer(Q)$

When $RD_3=1$ for all the peers it is well-known that the overlay topology must be a supergraph of the *Delaunay Graph* of the peers' coordinates if we want the algorithm described above to find *closestPeer(Q)* correctly for any point Q, no matter from which peer X we start. If the Delaunay graph is not a subgraph of the overlay topology, then there may be cases in which the algorithm „gets stuck" in a local minimum (i.e. we reach a peer X which is closer to Q than any other peer in $EN(X,1,t)$, but is not the closest peer to Q in the whole system). The Delaunay graph of n points is the dual of the Voronoi diagram of the points. At every (discrete) time moment t, every peer X could use the *Local Voronoi* method described in section 3.1. This procedure converges towards the Delaunay graph of all the peer's coordinates. However, computing the Voronoi diagram in $D \geq 3$ dimensions is a tedious task for any distance function. Because of this, we developed a simpler algorithm which is an instance of the *Hyperplanes* method (also described in section 3.1).

We consider the L_1 (Manhattan) metric as the distance function. For this metric and restricting our attention to two dimensions, the following graph is known to be a connected subgraph of the Delaunay graph. We consider the *8* octants around every point P, determined by the OX and OY axes, and by the two straight lines $x=y$ and $x=-y$ (considering point P as the origin). We add an edge between point P and the nearest neighbor in every octant (the point Q located in that octant for which the distance to P is minimum). We generalize this approach to D dimensions, as follows.

We consider the set of hyper-planes $a(1)\cdot x(1)+...+a(D)\cdot x(D)=0$, with $a(i) \in \{-1, 0, +1\}$ and $x(i)$ is a variable corresponding to the i^{th} dimension ($1 \leq i \leq D$). We disconsider the case where all the $a(*)$ values are *0*. Moreover, we only consider the hyper-planes for which the first non-zero parameter $a(i)=1$ (thus, $a(1 \leq j \leq i-1)=0$), because two hyper-planes A and B are identical if we can obtain the hyper-plane B by mutiplying the parameters of the hyper-plane A by *-1*. There are $(3^D-1)/2$ such hyper-planes. For every peer $V \in I(U,t)$ (t=the current time moment), we compute its *sign coordinates* $(s(1), ..., s((3^D-1)/2))$. $s(i)$ ($1 \leq i \leq (3^D-1)/2$) is computed as follows. We consider the i^{th} hyper-plane.

We compute $vv(i)=a(i,1)\cdot dif(1)+...+a(i,D)\cdot dif(D)$, where $dif(j)=(coord(V,j)-coord(U,j))$ ($1 \leq j \leq D$) and $a(i,1)\cdot x(1)+...+a(i,D)\cdot x(D)=0$ is the equation defining the i^{th} hyper-plane.

Then, if $vv(i)<0$, $s(i)=-1$; otherwise, if $vv(i)>0$, then $s(i)=+1$; otherwise, $s(i)=0$. We divide all the peers $V \in I(U,t)$ into TS sets, where TS is the total number of distinct *sign coordinates* of the peers from $I(U,t)$, and two peers A and B belong to the same set if they have the same sign coordinates. From every set ST ($1 \leq ST \leq TS$), the peer U chooses (connects to) the $K(ST)^{(*)} \geq 1$ peers

whose coordinates are closest to *coord(U)* (or fewer, if there aren't *K(ST)* peers in the set *ST*) according to the L_1 distance.

The only problem is that we cannot provide any guarantees that the Delaunay graph of the peers is a subgraph of the obtained topology, in which case the *routing* algorithm (with $RD_3=1$ for all the peers) would not work; but we can always make the algorithm work by increasing the *K(*)* values, RD_3, or both.

3.5.3. Object Management and Multidimensional Range Queries

3.5.3.1. Object Management

When a user application inserts an object *O* starting from a peer *X*, the peer *X* forwards the object to the peer *Y* whose coordinates are closest to object *O*'s properties' values (we use the L_1 distance for measuring *closeness*). Peer *Y* will be object *O*'s *owner*. Each peer *Y* maintains a set with all the objects it *owns*. We denote this set by *masterCopies(Y)*.

Periodically (every $TPObj^{(*)}$ seconds), each peer performs object management updates. The time duration between two consecutive object management updates is significantly larger than the time period between two consecutive topology updates. Thus, we will assume that the topology is stable (i.e. it converged to its equilibrium state) when an object managenement update is performed. An object management update consists of two steps.

In the first step, every peer *X* sends a copy of every object $MCO \in masterCopies(X)$ to every peer $Y \in EN(X,RD_4,t)$ (where $RD_4^{(*)} \geq 1$ is a small value). When a peer *Y* receives a copy of an object *O*, it verifies if it already contains a copy of the same object in its *replicas(Y)* set. If it doesn't, then it inserts the copy into *replicas(Y)*, after setting its *lastUpdated* field to the current time moment. If peer *Y* already contains a copy of the object in *replicas(Y)*, then it only updates the *lastUpdated* field of the stored copy (setting it to the current time).

The second step performed by each peer *X* is to recompute the owner of every object copy stored in the set *replicas(X)*, which has not been updated for a long time. We consider two time parameters $\text{tmax}_1^{(*)}$ and $tmax_2^{(*)}$ ($tmax_1 < tmax_2$). If the difference *tdif* between the current time and the *lastUpdated* field of some object copy $O \in replicas(X)$ has the property $tmax_2 < tdif$, then the peer *X* assumes that $X \notin EN(Y,RD_4,t)$, where *Y* is the (most recent) owner of the object's master copy; thus, peer *X* deletes *O* from *replicas(X)*. If $tmax_1 \leq tdif \leq tmax_2$, then peer *X* may assume that the owner of the object's master copy has left the system. Thus, it finds the peer *Y* which is closest to the object *O* in the property space.

If peer *Y* owns the master copy of the object *O*, then peer *X* takes no further actions regarding this object. However, if $O \notin masterCopies(Y)$, then we distinguish two cases. If *X=Y*, then *X* assumes that it should be the new owner of the object *O* and removes *O* from *replicas(X)* and inserts it into *masterCopies(X)*; otherwise, it tells peer *Y* to insert a copy of *O* into *replicas(Y)* (unless one is already there) and set the *lastUpdated* field of the stored copy to the current time of peer *Y*. This way, at a future object management update, peer *Y* will compute the owner of the object *O* (since $O \in replicas(Y)$) and will decide by itself if it is the (new) owner of the object's master copy.

Also as a part of this second step, each peer *X* considers every object $O \in masterCopies(X)$. For each such object *O*, it finds the peer *Y* which is closest in property space to object *O*. If *X=Y*, then no actions are taken regarding this object. However, if $X \neq Y$, then peer *X* tells peer *Y* to insert object *O* into *replicas(Y)* (unless peer *Y* already contains a copy of the object *O* in *replicas(Y)*) and set the *lastUpdated* field of the stored copy to the current time of peer *Y*. Afterwards, peer *X* removes *O* from *masterCopies(X)* and inserts it into *replicas(X)*. The *lastUpdated* field of the copy inserted into *replicas(X)* is set to the current time of the peer *X*. The case $X \neq Y$ may arise, for instance, after a new peer has joined the system.

3.5.3.2. Multidimensional Range Queries

A range is a *D*-dimensional hyper-rectangle *R=[x(1,1),x(1,2)]x...x[x(D,1),x(D,2)]*, with

$x(i,1) \leq x(i,2)$ ($1 \leq i \leq D$). A range query consists of finding all (or at most a given number M of) the objects whose index property values are within the specified range (assuming that $Prop(O,i)$ is the value of the i^{th} index property of the object O, we must have $x(i,1) \leq Prop(O,i) \leq x(i,2)$ for every $1 \leq i \leq D$) and whose data fields and property values satisfy a given filtering condition.

When a user application wants to perform a range query it chooses any peer X and tells it to perform a range query for a given range R. Peer X computes the coordinates Q of the center of the hyper-rectangle R and then forwards the range query to the peer $Y=closestPeer(Q)$. After reaching this peer Y, the range query enters into the second stage. During this stage, the request is broadcasted according to the following rules. As soon as a peer P receives a range query request (in the second stage), it forwards it further. If P's coordinates belong to the query range R, then the request is forwarded to all of P's neighbors (the peers in $N(P,t)$; t=the current time moment). If P's coordinates do not belong to the query range R, then it forwards the request only to those peers $W \in EN(P,RD_5,t)$ (where $RD_5^{(*)} \geq 1$ is a small value) whose coordinates either belong to the query range R or are closer than $coord(P)$ to some part of R.

To be more precise, we denote by $closestPoint(X,HR)$ the closest point on the contour of a D-dimensional hyper-rectangle HR to the coordinates of peer X. Such a point is easy to compute in the L_1 metric. P may forward the request to a peer $P' \in EN(P,RD_5,t)$ if $dist(P',closestPoint(P',R)) < dist(P,closestPoint(P,R))$. Thus, in the second stage of a range query, only peers which can store an object whose properties are within the query range are visited.

When the request reaches a peer P, this peer checks every object O in the sets $masterCopies(P)$ and $replicas(P)$. If the index properties' values of such an object O belong to the range R of the query and its index properties' values and data fields satisfy the filtering condition, then the object's identifier is sent to the peer Y which initiated the second stage. Every range query has a unique (self-generated) identifier and every peer maintains the identifiers of the queries it received in a cache. Thus, if a peer receives the same query request for the second time, then it ignores it (and does not forward it further). The entries in the cache expire after a certain time period.

As soon as peer Y receives the identifier oid of an object from a peer P in the context of a request rid, it checks if it previously received the same identifier in the context of the same query: if it did not receive the identifier oid before in the context of the same query (and the threshold regarding the maximum number of objects specified by the query has not been reached), then it tells the peer P from which the object identifier was received to send the copy of the object (with identifier oid) to the peer X which initiated the range query (which will send the object to the user application). Peer Y (and/or the user application) may assign a time limit to every query and ignore every answer to the query which is received after the time limit is exceeded.

We should mention that the same conditions which are required for the *routing* algorithm to work correctly are also required for the range query process to report all the objects in the range.

Peers may store the objects in their *masterCopies* and *replicas* sets as they see fit. For instance, they could use any data structure which facilitates the range query process one way or another. Note also that, except for range reporting queries, we can also allow range aggregate queries. Such a query asks for an aggregate value (e.g. sum, max, product) of the values of a given property of all the objects whose index properties belong to a given range. In this case, when a query is processed by a peer P, this peer will compute the range query answer for all the objects from $masterCopies(P)$ (and not $replicas(P)$) which belong to the query range (this computation may be facilitated by the use of some efficient data structures, as mentioned previously). The answer is then sent to the peer Y, which aggregates all the received answers and sends the final aggregate value to the peer which issued the query.

3.5.4. Experimental Evaluation

In order to validate and evaluate the proposed system, a time step-based simulation framework was developed and implemented in the Python programming language. For all the tests, the property space was considered to be $[0,1000]^D$, $RD_1=...=RD_5=1$ and $TS=10$. The peers'

coordinates were uniformly distributed in the property space.

3.5.4.1. Topology Test

We incrementally inserted *100* peers into the system (with *D=2* and *K(*)=1*). After every insertion, we computed the number of time steps before the topology stabilized. Moreover, after every insertion, we computed the diameter of the network (the longest shortest path between any pair of peers), the maximum degree of a peer and the average degree of the peers (degree=number of neighbors). The results are shown in Fig. 3-12 (left). We can see that the number of iterations before the topology stabilizes (after every peer insertion) is proportional to the diameter of the network (plus or minus a very small constant). We can also see that both the average degrees and maximum degrees are sufficiently small. Thus, a good distribution of the peers in the property space was achieved, as well as (routing) traffic load balancing. Then, we incrementally inserted *1000* peers into the system. After every insertion we computed the diameter of the network; the maximum diameter values were: *18* (for *D=2* and *K(*)=1*), *13* (*D=2*, *K(*)=2*), *5* (*D=3*, *K(*)=1*) and *4* (*D=3*, *K(*)=2*).

3.5.4.2. Orthogonal Range Query Test

At first, we performed one range query test, with *D=2*, *K(*)=2*, *1000* peers and *1000* objects, whose property values were randomly generated within a square whose area occupied approximately *20%* of the property space. We performed *130* random range queries and we measured the following parameters: the ratio between the number of reported objects and the total number of objects (objects ratio), the ratio between the number of queried peers and the total number of peers (peers ratio) and the ratio between the volume of the range and the total volume of the property space (volume ratio). The results are presented in Fig. 3-12 (right), where the range queries were sorted according to the objects ratio. The peers ratio was close and sometimes lower than the objects ratio. The volume ratio was strongly correlated with the peers ratio, due to the uniform distribution of the peers. Then, we performed similar range query tests for *D=3* and *D=4*, with *K(*)=1*, *100* peers and *100* objects. The peers ratio was always very close to the volume ratio and close to or lower than the objects ratio.

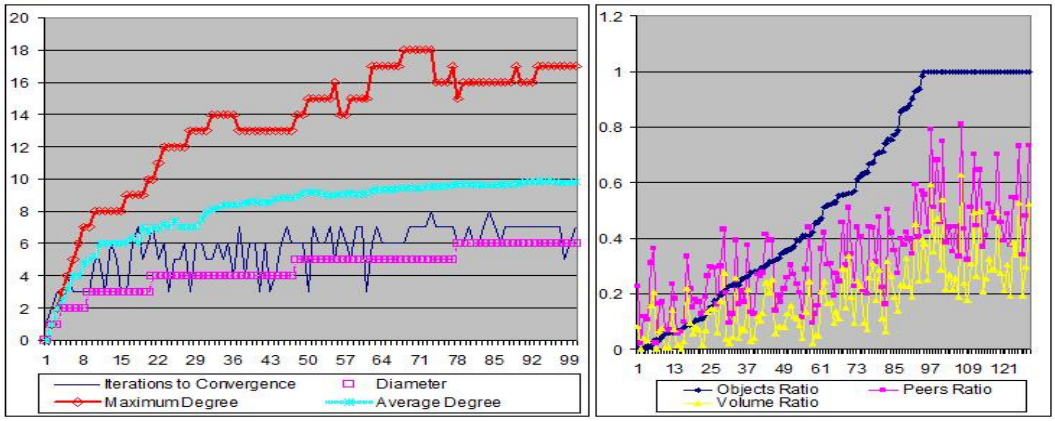


Fig. 3-12. Topology Test (left). Range Query Test (right).

3.6. Upload Bandwidth Estimation and Congestion Control

3.6.1. Upload Bandwidth Estimation

Estimating the upload bandwidth of a machine (e.g. computer) is extremely useful in a wide variety of scenarios and applications, like, for instance, peer-to-peer applications based on the Bittorrent tit-for-tat mechanism or other similar techniques (many file sharing, live streaming, and video on demand systems belonging to this class have been proposed during the past few years – see, for instance, [Tewari and Kleinrock, 2007] and [Choe, Schuff, Dyaberi and Pai 2007]). In such systems, the downloaded data of every peer *P* is proportional to the data uploaded by peer *P* to the

other peers. Since in order to maximize its overall utility, a peer wants to download data at high transfer rates, it must also be able to upload data to other peers at high speeds.

However, most Internet users are connected to the Internet via asymmetric links, in which the download speed (bandwidth) is significantly higher than the upload speed (bandwidth). As such, the situation in which the upload bandwidth is fully utilized can easily occur. Such a situation may cause some problems. One of the most pregnant ones is the behavior of TCP flows when the upload link is congested. Through experiments, we determined that if a peer *P* downloads data at a rate *D* through a TCP connection, then an upload rate *U* of up to *2-5%* of *D* is used by the TCP protocol for sending *ACK* messages.

If the upload link is congested and less than *U* bandwidth is available, the download rate *D* cannot be maintained and the TCP protocol makes use of its well known AIMD mechanism, which reduces the download speed drastically in a short time (while allowing it to increase back to its former values only slowly). Thus, when the upload link is congested, the download rates of TCP connections are, on average, far from the optimal performance. If, however, we knew the (available) upload bandwidth, we could reserve part of it for TCP acknowledgements, thus maintaining the download rate at a high average value. Other situations in which knowing the (available) upload bandwidth of a machine is useful are concerned with the implementation of higher-level functions and behaviors, like content seeding, peer selection, bandwidth trading, and so on.

In this section we will present a novel upload bandwidth estimation technique, which was partly developed in the context of the European Union FP7 project P2P-Next. At the moment, the technique is applicable for estimating the upload capacity of a machine (i.e. its total upload bandwidth), in the absence of background traffic. The technique also works when background traffic is present, but it does not compute the available upload bandwidth, because the background communication flows can be influenced by this method.

An upload bandwidth estimation technique should be as non-intrusive as possible (i.e. it should generate little extra traffic). If possible, it would be desirable to make use of the existing traffic in order to estimate the upload bandwidth. Due to portability reasons, the technique should be implemented in user-space and should not make use of operating system-specific functions.

3.6.1.1. A Novel Upload Bandwidth Estimation Technique

The proposed technique works as follows. When a peer *S* wants to estimate its upload bandwidth, it will need the help of $N \geq 1$ other helper peers (*P(1), ..., P(N)*). Peer *S* will send *M(i)* packets to each peer *P(i)* ($1 \leq i \leq N$). The packets sent to the same peer *P(i)* must have equal sizes (*PSize(i)*), but packets sent to different peers may have different sizes. It is also not necessary to send the same number of packets to every peer *P(i)*. Peer *S* will send the *M(1)+…+M(N)* packets one after another, in some order, such that any *2* consecutive packets sent by *S* should preferably be sent to two different peers. What is important, however, is that the upload bandwidth of the peer *S* should be constantly used, i.e. there should be no delays between two consecutive packets sent by *S*. We assume a FIFO queue at the sender (as is usually the case), i.e. the packets are transferred on the upload link in the order in which they are sent by *S* (no matter to which helper peer they are sent).

When a peer *P(i)* receives the j^{th} packet, this packet will also contain the value *TAB(i,j)*=the total amount of bytes that peer *S* has sent to all the *N* peers up to the moment when the currently received packet was sent by *S* (including the size *PSize(i)* of the currently received packet). Then, let *TAB(i,j-1)* be the value received by *P(i)* at the previous packet (we consider the case $j \geq 2$). Let's assume that packet *j-1* was received by *P(i)* at time *T(i,j-1)* and packet *j* was received at time *T(i,j)*. Peer *P(i)* will compute an estimation

$$U(i,j-1)=(TAB(i,j)-TAB(i,j-1))/(T(i,j)-T(i,j-1)) \quad \textbf{(3-8)}$$

of the upload bandwidth of peer *S*. Note that some of the packets sent by peer *S* may be lost and the j^{th} packet received by peer *P(i)* may not necessarily be the j^{th} packet sent by peer *S* to *P(i)*. The packets may also be received out of order. When a peer *P(i)* receives a packet, it first checks if the information contained in the packet regarding the total number of bytes sent so far by peer *S* is

larger than that of the previously received packet (unless it is the first received packet) - if the information value is not larger, then the currently received packet is discarded. The information regarding the total number of bytes sent by peer *S* acts as a sequence number for the packets, because it increases with time.

After sending the last packet to every peer *P(i)*, peer *S* notifies every peer *P(i)* that the test is complete (the notification should preferably not be lost, although it is not important if a small fraction of peers do not receive the notification). Every peer *P(i)* has a time limit for waiting for new packets. When this limit is exceeded, it will assume that the test is complete (i.e. it will behave as if it had received the test completion notification). At the end, every peer *P(i)* has *E(i)* estimations: *U(i,1), ..., U(i,E(i))*. We will remove from this set the outliers (the values which are too high or too low) and compute an average *Uavg(i)* of the remaining values. Peer *P(i)* will then send *Uavg(i)* to peer *S*.

For the outliers removal, the following technique was considered. We compute the median value U_{med} of the estimations. Then, we remove all the estimations which are smaller than $p_1 \cdot U_{med}$ or larger than $p_2 \cdot U_{med}$ (for some carefully chosen values $0 \leq p_1 \leq 1$ and $p_2 \geq 1$). Afterwards, we perform an iterated removal of borderline values. As long as we have more than *K* estimations left (e.g. *K=3*) we perform the following actions:

1. we compute U_m=the average of the values of the remaining estimations and *sgm*=the standard deviation;
2. we remove all the estimations whose values do not belong to the interval $[U_m - q \cdot sgm, U_m + q \cdot sgm]$ (for a carefully chosen value of *q*; e.g. *q=1*);
3. if no values were removed in step *2* then we break the loop.

In the end, peer *S* will receive the estimations *Uavg(i)* from (some of) the peers *P(i)*. If at least a fraction *PA* (e.g. *PA=0.6*) of these values are "close" (and at least *PB·N* values were received; $0 < PB \leq 1$), then we remove the other values and compute the average of the remaining values: this will be the estimated upload bandwidth. We define *closeness* as follows. We compute the median U_{md} of the received values and then we compute the number of received values which lie in the interval $[p_3 \cdot U_{md}, p_4 \cdot U_{md}]$ (where $0 \leq p_3 \leq 1$ and $p_4 \geq 1$).

If we do not have at least a fraction *PA* of "close" values, then it is possible for the estimated values to be too low, because the upload bandwidth estimations of peer *P(i)* are also influenced by the available bandwidth *AB(S,P(i))* of the path between *S* and *P(i)* (in fact, theoretically, we have $T(i,j)-T(i,j-1)=max\{(TAB(i,j)-TAB(i,j-1))/SUB, PSize(i) / AB(S, P(i))\}$, where *SUB* is the upload bandwidth of peer *S*). This issue can be solved by sending larger packets or by using more helper peers: this way, two consecutive packets will reach a peer *P(i)* after a larger time interval, overcoming the influence of *AB(S,P(i))*. Fig. 3-13 depicts the proposed technique, in which the same number of equally sized packets is sent to each of the *N=4* helper peers in a round-robin fashion.

Let's have a closer look now at the way the upload bandwidth estimation technique works. If *N=1*, then *P(1)* actually estimates a value *B* which is upper bounded by the smaller of the following two values: the (available) bandwidth of the path between *S* and *P(1)* and the upload bandwidth of *S*. In fact, it is possible that the available bandwidth from peer *S* to any of the helper peers is smaller than the upload bandwidth of peer *S*. However, by sending packets to multiple peers (e.g. in a round-robin fashion), peer *S* does not congest the paths to the helper peers. Moreover, a helper peer *P(i)* also receives the total number of bytes sent by peer *S* so far during the test. The difference between the total number of bytes transmitted with two consecutive packets received by *P(i)* is larger than the number of bytes that peer *S* could have sent directly to *P(i)* in the same time period (when *N>1*).

A requirement for the technique to work correctly is that the sum of the bandwidths of the paths from *S* to every helper peer should be at least as large as the upload bandwidth of peer *S* (that is why more helper peers are useful). If, however, the paths from *S* to multiple helper peers share common bottleneck links (other than the upload link of peer *S*), then the technique may still

incorrectly estimate (e.g. underestimate) the upload bandwidth. It is, thus, desirable for the helper peers to be geographically distributed, so that the paths from *S* to the helper peers may be as disjoint as possible.

Let's notice that we can also use the method presented above for estimating the upload bandwidth in a continuous manner. Peer *S* repeatedly sends packets to each of the peers *P(i)*. After receiving the j^{th} packet ($j \geq MinP(i)$), peer *P(i)* can provide an estimation *Uavg(i,j)* using the previous *MinP(i)* estimations (thus, we use a sliding window kind of approach). *MinP(i)* is the minimum number of packets peer *P(i)* needs to receive in order to consider the estimations to be statistically relevant.

If the upload bandwidth estimation technique is used in order to help the decision making process of an application *App*, then the technique can make use of the information regarding the upload bandwidth consumed by *App* (this information can be made available, as the technique is integrated into *App*). We will consider that all the upload traffic generated by *App* is sent to a "virtual helper peer" *P(N+1)*, which will not send any estimation back (although, in reality, the upload traffic of *App* may have multiple destinations).

Note that in order for the presented technique to produce reliable results, peer *S* must never be idle in terms of upload traffic (i.e. it should always upload something): this is because when a peer *P(i)* measures the time difference between two consecutive packets that it receives, it makes the implicit assumption that peer *S* has been uploading data during all this time. Thus, as long as the upload buffer(s) of peer *S* are not empty, peer *S* does not have to send a new packet to any of its helper peers; it just has to increase the counter of the total number of bytes sent by *S*. However, since upload bandwidth estimations are received by peer *S* only from the peers *P(i)*, peer *S* cannot postpone indefinitely the sending of a packet to a peer *P(i)* (even if the application *App* generates enough traffic).

Thus, the previously described technique can be modified as follows. Peer *S* will only send the next packet to the next peer *P(i)* (e.g. in the round-robin order) if the amount of upload buffer space used by *App* is below a certain threshold or the duration between the moment when the previous message was sent to a helper peer *P(*)* and the current time moment exceeds a given time limit. Note that when we also use existing *App* traffic, we can reduce the number of packets after which a peer *P(i)* computes an estimation, thus reducing the overall extra traffic generated by this method. The number of packets after which an estimation is computed could even be determined by each helper peer separately, based on the values *(TAB(i,j)-TAB(i,j-1))* and *(T(i,j)-T(i,j-1))* (e.g. the larger these values are, the fewer packets are required before obtaining an accurate estimation).

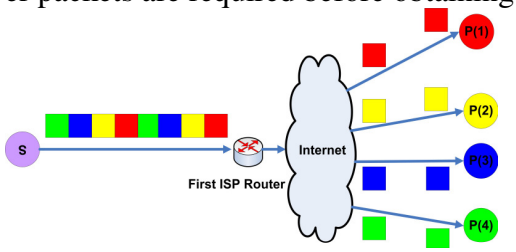


Fig. 3-13. Upload Bandwidth Estimation in Progress.

3.6.1.2. Experimental Evaluation

We implemented the proposed technique in the Python programming language and we validated it as follows. The source peer *S* was located behind a NAT (in Bucharest), running Windows Vista; we used *N=3* helper peers, all running Linux: *P(1)* was located at the Technical University of Delft (Netherlands), *P(2)* was located at the University of Craiova (Romania), and *P(3)* was located at the Politehnica University of Bucharest (Romania). We sent *M=20* packets to every helper peer, in a round-robin manner; we used $p_1=0.2$, $p_2=5$, $p_3=0.8$, $p_4=1.2$, *PA=PB=0.6* and *PSize(1)=…=PSize(N)*. We ran *6* tests, in which we only changed the packet sizes: *1024*, *2048*, *4096*, *8192*, *16384*, and *32768* bytes. The source peer opened one TCP connection to every helper peer, for sending the corresponding packets (we designed our own protocol in order to mark the

beginning and the end of a packet). The estimation computed by each helper peer is presented in Fig. 3-14.

Later, we also performed the same test, by sending UDP packets (instead of TCP). The results were similar. We also computed the upload bandwidth of the computer by using the *SpeedTest* website (*http://www.speedtest.net*), obtaining a value of approximately *232.500 Bps*. We noticed that the results of our test were closer to *240.000-245.000 Bps*, which was, in fact, the appropriate range for the upload bandwidth of the tested computer.

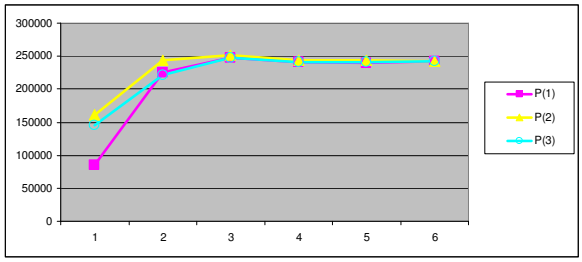


Fig. 3-14. Estimated Upload Bandwidth (B/sec) as a Function of Packet Size (2^{10}-2^{15} bytes).

Then, we wanted to decide which packet size is most suitable for estimating the upload bandwidth for the tested computer. We considered that the correct upload bandwidth was *240.000 Bps* and we computed the error between the estimated value and the correct value (i.e. the absolute difference between them) for each of the *6* tests. Then, we multiplied the error by the total generated traffic and we plotted the results in Fig. 3-15. Some good packet sizes are between *2048* and *16384* bytes; of course, the lower the packet size, the better.

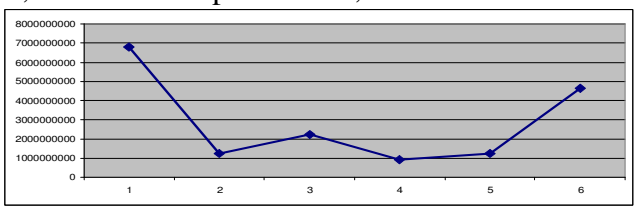


Fig. 3-15. The Product between the Error of the Estimation and the Total Generated Traffic for Each of the *6* Tests.

We also performed some tests which showed us that the technique is not currently efficient for estimating the available upload bandwidth. We used the same scenario, in which we started two background applications on the tested computer, uploading data at *70 KBps* and, respectively, *80 KBps* overall. The applications were custom made. They uploaded random data to a given destination, using $PT \geq 1$ parallel TCP streams each and sending *4 KB* packets. We first set $PT=10$ and then we ran the same upload bandwidth estimation tests, except the one with packet size of *1024* bytes. The overall transfer speed of each application decreased by at most *3 KBps* during the tests and the test results were close to *90 KBps* (i.e. the available upload bandwidth was estimated rather accurately).

However, when we used $PT=1$, the transfer speeds of the two background applications dropped significantly, depending on the packet size. For packet sizes of *32768* bytes used during the test, the transfer speeds of the applications dropped down to *20-30 KBps* each. Thus, the TCP flows of the background applications can be severely influenced by our proposed technique. If we could somehow instruct the operating system to handle the test packets as low priority packets (i.e. send the test packets only when no other packets are waiting to be sent, or after they were ignored for more than a certain time duration), then we might be able to use only the actual available upload bandwidth. However, it seems that most operating systems consider that every flow has the same priority and packets are sent in a first-come first-served manner.

A possible way of estimating the available upload bandwidth *AUB* is the following. We can introduce an upload speed limit *R* in our technique – thus, peer *S* will not necessarily upload data continuously. Let *U(R)* be the upload bandwidth estimation obtained for a limit *R*. If $U(R) \geq cr \cdot R$, then $R \leq AUB$; if $U(R) < cr \cdot R$ then $R > AUB$ (where $0 < cr \leq 1$, but close to *1*). Thus, we could use a

search technique (e.g. exponential and binary search) for finding the largest limit R for which $U(R) \geq cr \cdot R$ (i.e. $R \leq AUB$).

In the end we mention that the presented technique also works when the tested machine has multiple physical upload links. In this case we must find a suitable set of helper peers, such that when performing the test, all the upload links are saturated (enough packets are sent through each link), or we could try to test every upload link separately.

3.6.1.3. Alternative Approaches

We also considered a different approach, for estimating the available upload bandwidth, based on measuring ping times to a set of carefully chosen landmarks from the Internet. The source peer S uploads data at (at most) a (total) given rate R to a subset of helper peers, for a duration T, during which it measures the ping times to the set of landmarks. We expect that, as the transfer rate R gets closer to the available upload bandwidth AUB, a larger fraction of pings exceed their time limit. Then, we could increase (or decrease) the rate R with small increments, until the ping times satisfy some quality conditions (e.g. a percentage of them are below some threshold), thus converging towards AUB. We present below the results of a first set of experiments.

Peer S was located in Bucharest, did not have a public IP address, was running Windows Vista and its upload capacity was approx. *60 KBps*. We chose only one helper peer P, located at the Politehnica University of Bucharest (UPB), running Linux and having a public IP address. We ran the test scenario *5* times. The maximum transfer rate was limited at: *25 KBps*, *35 KBps*, *40 KBps*, *45 KBps* and *unbounded*. Every time, the total duration of the upload test was *10* minutes. We measured ping times from the peer S to a machine located at the UPB site. Without the test traffic, the ping times ranged from *15* to *60* milliseconds. For the *25 KBps* upper bound, most of the ping times were under *100 msec*, with only *3* occasional ping time spikes (two of which were ping timeouts). For the *35 KBps* limit, most of the ping times were under *400* msec and no ping timeouts occurred. For the *40 KBps*, several pings timed out in the beginning of the test; however, except for this, the ping times were quite constant, not exceeding *500 msec*. For the *45 KBps*, all the ping times during the actual data transfer exceeded our *20* second time limit. In the unbounded case, the average upload rate was *55 KBps* and the ping times showed a steady increase towards our *20* second time out limit, followed by many ping timeouts.

From this set of experiments, we draw the following preliminary conclusions. During an upload bandwidth test without variable background traffic, the ping times present quite a regular behavior. We mention that this behavior is also the result of the technique used to limit the transfer rate. We considered several techniques, some of which led to irregular ping time behavior, and we settled on one where the actual upload rate is constantly corrected (both by introducing time delays and by sending at most a number X of bytes at a time, where X depends on the current upload rate and on the total number of bytes transmitted so far). As expected, the average ping time and the median ping time increase with the upper bound of the upload rate.

The implemented mechanism is intrusive, because it needs to send a significant amount of extra traffic in the network. However, we believe that it can be used in a useful non-intrusive manner, as follows. In order to estimate AUB accurately using this technique, we might need to send data at the same rate as the available bandwidth. We consider this to be too intrusive and we propose the following use in applications *App* which want to use this technique in order to increase their total upload speed. We can estimate if AUB is larger than a small value R (by sending data at the rate R and checking if the ping times satisfy the quality conditions). Let's assume that the current upload rate of *App* is U. If $AUB \geq R$, we will use the technique again only after the upload transfer rate of *App* becomes $U+R$. Thus, we only generate as much extra traffic as *App* can use.

As future work, we intend to find a correlation between a statistical measure SM of the ping times and the upload rate U. By using the technique for several small values $\{U_1, ..., U_r\}$ of the upload rates and computing the corresponding statistical measures $\{SM_1, ..., SM_r\}$, we hope to find a correlation $U=f(SM)$. Then, by setting an upper limit SM_{max} on the statistical measure, we could compute the largest upload rate $Umax$ that we can use.

3.6.2. Congestion Control

Congestion control is an important problem in peer-to-peer applications. As peer-to-peer traffic constitutes the majority of Internet traffic nowadays, we need to make sure that network resources are efficiently utilized and also that the quality every user perceives is optimized. Experimental studies [Pouwelse, Garbacki, Epema and Sips, 2005] have shown, for instance, that Bittorrent users do not stay online for too long (e.g. more than one hour after finishing their downloads) in order to share the content they downloaded and, thus, to improve the download speeds of the other users. Because of this, a large amount of work has been dedicated to devising efficient incentives which should convince the users to remain online longer. However, it is possible that users do not stay online partly because they are worried that the uploading of data may reduce the transfer quality of the other applications they currently use, like email, web browsing, FTP traffic, and so on. Thus, it might be possible that users would feel comfortable staying online for longer periods if they knew that other applications were not affected by their upload share contribution to the peer-to-peer content sharing system. Novel congestion control algorithms are required in order to achieve this purpose.

A "less than best effort" communication flow has the property that it has a lower priority than all the other normal "best effort" communication flows. This means that, in the presence of other flows, it should back-off and make room for the other flows. However, in order to be efficient, it would be desirable for the transfer speed of the communication flow to be as large as possible (i.e. fill the network link as much as possible) without affecting the "best effort" communication flows.

In this section we briefly mention a novel congestion control method which makes the communication flows using it act as "less than best effort" flows in relation to other flows, but, at the same time, it attempts to fill the upload link as much as possible. The method is based on the experience derived from the upload bandwidth estimation experiments and is based on imposing a sending rate limit and then estimating the upload bandwidth (while this limit is in effect). The principle is that if the upload bandwidth is close to the sending rate limit, then the sending rate may be increased; otherwise, it should be decreased.

3.7. Towards an Automated Framework for Testing Large Scale Distributed Systems

Testing (large scale) distributed systems and applications (e.g. peer-to-peer systems) is important in order to be able to validate and evaluate the performance of the system. In the absence of automatic testing means, the most common procedure for testing a distributed application consists of deploying the application *manually* on several nodes and then running the tests. Such an approach is obviously not scalable. Its most important drawback is the large amount of work the application developer (or tester) needs to perform in order to run the tests once. Since a distributed application must be tested multiple times, in different stages of its development (or possibly after fixing some bugs), the amount of required work increases too much. It is obvious that a framework for automating the deployment and testing process of a distributed application would be highly desirable.

In this section we will present the design and implementation of ServMark, a framework for automating the testing of large scale distributed systems (like, for instance, peer-to-peer systems). The framework was initially developed with the purpose of testing Grid and Web services, but we will explain how it can also be used for testing peer-to-peer systems.

ServMark is a system that integrates two previously developed evaluation systems: DiPerF and GrenchMark. DiPerF is a distributed testing system and test generator, and GrenchMark is a centralized system that can generate complex testing scenarios. ServMark makes use of the properties of both systems in order to generate truly significant testing scenarios. The intended use for ServMark is to evaluate the performance of Grid environments and Grid and web services, but can also be used for deploying and testing peer-to-peer applications, as we will show in a future sub-section.

3.7.1. The Design of ServMark

The testing process is initiated by a central controller, which distributes the testing parameters to multiple nodes. Each node generates its own test scenario based on the given parameters and then "plays" the generated scenario.

The general requirements of a testing architecture like ServMark are the following:

1. uniquely identify each test (REQ1)
2. generate a multi-node test according to the user specifications (REQ2)
3. store the test and make it available for replay (REQ3)
4. run the test and store its results (REQ4)
5. analyze the results and compute statistics (REQ5)
6. the performance evaluation must be online: results should be able to be visualized as the testing process advances (REQ6)

Fig. 3-16 shows the proposed architecture for ServMark, highlighting the relationship between GrenchMark, DiPerf and the new ServMark modules. The interaction between the user and the ServMark Controller goes as follows: the user decides the parameters to be used in the testing process (see REQ2), starts the ServMark Controller, then is notified when the testing operation has completed. The ServMark Controller should generate a test ID for the testing process initiated by the user (see REQ1), update the database and send the testing parameters to the DiPerF controller. The DiPerF controller controls the testing process, by invoking the DiPerF submitter. It also updates the results into the database.

The DiPerF submitter creates the tester processes and communicates with them, sending in parameters and receiving back test results. The DiPerF tester invokes GrenchMark, which performs the actual testing process and communicates with GrenchMark, sending parameters and receiving back test results. GrenchMark generates a workload according to the user parameters and then submits the generated workload for execution, computing the test results and sending them to the DiPerF tester. The test parameters are inserted into the database by the ServMark controller. The DiPerF controller inserts and updates the test results into the database as the testing process advances.

The behavior described above is very natural for testing deployed (Grid and web) services, but less adequate for testing custom peer-to-peer applications. In order to test a peer-to-peer application, together with a DiPerF tester, a peer-to-peer applications is also deployed and executed. Its parameters are sent along with the parameters of the DiPerF tester. Then, GrenchMark generates the workload for testing the peer-to-peer application. For instance, in order to test a data transfer optimization peer-to-peer application, the application is sent and run together with the DiPerF tester. GrenchMark then generates data transfer requests in order to test the peer-to-peer system.

3.7.2. The Implementation of ServMark

Fig. 3-18 presents the detailed architecture of ServMark. The ServMark Controller interacts directly with the database, in order to insert general information about the testing scenario, while the DiPerF controller interacts with the database through a database module, in order to insert or update the information gathered during the testing process. GrenchMark is composed of two major modules: the workload generator and the workload submitter. The workload generator schedules the execution times of the jobs which compose the testing scenario.The workload submitter is a multi-threaded module which manages the job submission process, computes metrics and sends the results back to the DiPerF tester.

In order to run test, the user places all the relevant test parameters in a test file. A sample test file is given in Fig. 3-17.

The "`Granularity`" parameter refers to the testing strategy. When testing Grid systems, the jobs usually have a run time of the order of minutes, whereas when testing web services, the jobs have a running time of the order of tens of milliseconds. Other differences also exist, based on the way the results are sent back and the frequency at which the results are sent. This is specified by

the parameters „MonitoringInfoGathering" and "PushPeriod".

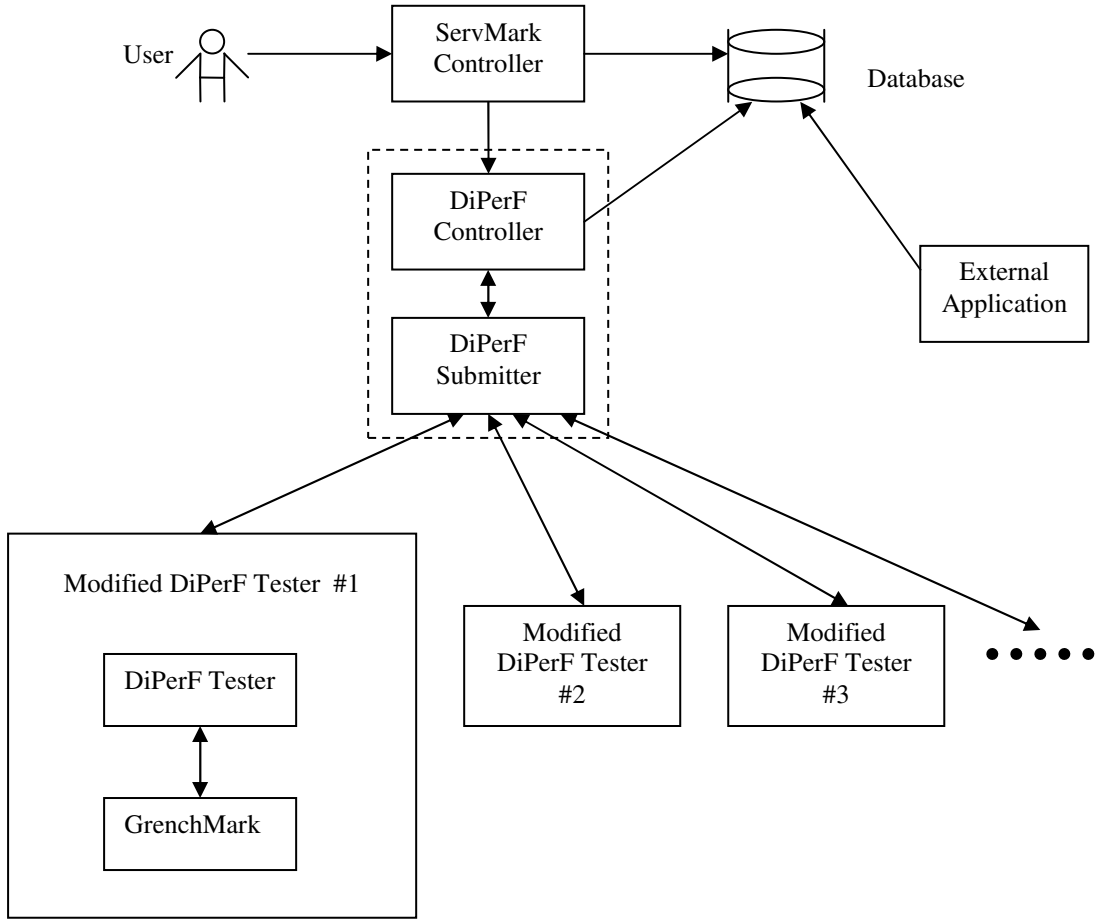


Fig. 3-16. The Proposed ServMark Architecture.

```
Granularity = custom
MonitoringInfoGathering = push
PushPeriod = 3000  # msec
ExecutableFileName = wget
CommandLineArguments = http://141.85.99.160:8080
NumberOfTesters = 50
JobsPerTester = 100
WorkloadDistribution = Poisson(1000)
SitesFile = planetlab-serv.txt
JobType = exe
LogFile = single
ProjectID = "servmark project"
DBServerName = myserver
DBUserName = myusername
DBPassword = mypassword
DBName = mydb
```

Fig. 3-17. A Sample Test File.

The "ExecutableFileName" parameter specifies the executable file name. When testing grid environments, this should be the name of the job to be executed. When testing web services, this should be the name of the client which will use the web services. When testing peer-to-peer applications, this should be the name of the executable of the application. The executable is copied from the machine initiating the test to the machines on which the DiPerF testers are run.

The parameter "CommandLineArguments" specifies the command line arguments which will be passed to the executable file (they can be enclosed between "" if they contain white

space – like "`-a x y -g no`" ; if no command line arguments are given, the user must specify "", i.e. an empty string).

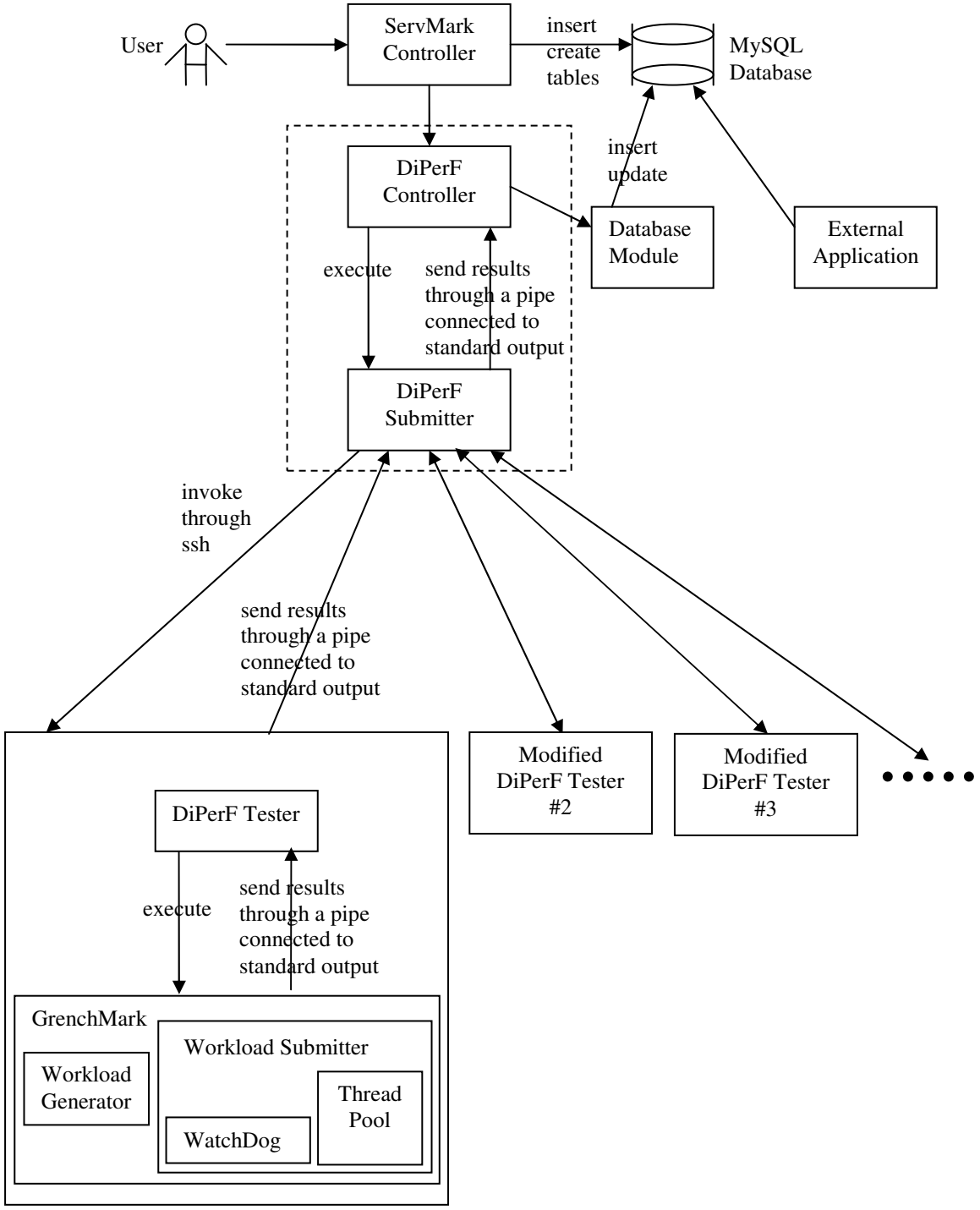


Fig. 3-18. ServMark's Detailed Architecture.

The parameter "`NumberOfTesters`" specifies the number of testers and the parameter "`JobsPerTester`" specifies the number of jobs which will be issued by each tester. The parameter „`WorkloadDistribution`" specifies the distribution of the times at which jobs are submitted. This parameter must be given in a format specific to GrenchMark (see [GrenchMark]).

The parameter "`JobType`" refers to the type of job and is a piece of information used by GrenchMark. Type "`exe`" represents a stand-alone application. Currently, there are several other types of jobs, all of which use the Koala Grid Resource Manager (see [Mohamed and Epema, 2004] and [Mohamed and Epema, 2005]). In order to test web services or peer-to-peer applications, type

"exe" is the most likely to be used. In order to test Grid environments, a type of job must be defined for its resource manager. GrenchMark is an extensible framework and new types of jobs can easily be defined.

The parameter "LogFile" is used by GrenchMark and can take one of two values: „single" and „multiple". The value „single" means that all the jobs write their standard output and standard error to the same file, while „multiple" means that each job uses its own file. When many jobs are issued, it is more appropriate to use only one file, in order to avoid the creation of too many files.

The parameter "ProjectID" is used as a project identifier. It is part of the primary key of some of the tables in the database, together with an auto-generated test id. It is useful in order to group together several testing processes which are part of the same project.

The parameter "SitesFile" represents the name of a file which contains a weighted list of machines on which testers will be spawned, one element on a line. A sample file is the following:

```
fs3.das2.ewi.tudelft.nl/20
s8.diperf.cs.uchicago.edu/10
alice01.rogrid.pub.ro/5
```

Fig. 3-19. A Sample Sites File.

The numbers are separated by the names by a '/' character. The number represents a weight (and can be a real number). When deciding on which machines the testers will be spawned, these weights will be considered. For instance, considering the above file and using 7 testers, 4 of them would be spawned on fs3.das2.ewi.tudelft.nl, 2 of them on s8.diperf.cs.uchicago.edu and 1 on alice01.rogrid.pub.ro.

The parameters "DBServerName", "DBUserName", "DBPassword" and "DBName" refer to the database where results will be stored. "DBServerName" represents the name of the machine where the database server is installed (currently, only MySQL is supported), "DBUserName" and "DBPassword" represent the username and password used to connect to the database server, and "DBName" represents the name of the database. The database tables used by the testing process will be created (if they do not already exist) by the ServMark controller.

3.7.2.1. The ServMark Controller

The ServMark controller parses the test file and assigns default values to the parameters which are not given in the file. It uses the given parameters to generate a file containing the machines on which the testers will be spawned. The file is in a format specific to DiPerF. It then generates an input file for the DiPerF controller. Finally, the controller creates the corresponding tables in the database (if they do not already exist), inserts into the database the test parameters, thus generating a unique test ID and invokes the DiPerF controller.

3.7.2.2. The Modified DiPerF Controller

The DiPerF controller invokes the DiPerF submitter and the standard output of the submitter is connected to a pipe from which the controller will read back the results. The DiPerF controller keeps reading characters from the pipe. For every complete line it receives (ended by a newline character), it checks if it is a line containing results and, if so, it sends it to the database module. A line containing results has a specific prefix, called LOGFILE_PREFIX. When sent to the database module, this prefix is stripped off.

3.7.2.3. The Modified DiPerF Submitter

The DiPerF submitter receives its parameters in its command line invocation. Some of these extra parameters will be sent to each tester. Except for these parameters, each tester receives a unique ID from the submitter (the tester IDs are consecutive integer numbers ranging from 0 to the number of testers minus 1).

Each tester is invoked through a SSH connection on the machine on which it needs to be executed. Its standard output is connected to a pipe. The submitter reads from the pipes connected to each tester's standard output and sends the lines read to the controller (by writing them to its own standard output).

DiPerF allows for two modes of executing a tester. In the first mode, the tester receives the name of an executable file which will be executed and needs to be located on the machine of the tester. In the second mode, the tester receives through its standard input a .tar archive which is decompressed and then a file is executed which is contained inside the archive. This is the only way files can be transferred from the machine of the controller to the machine on which each tester is executed. In ServMark, only this second mode is used. The archive contains the GrenchMark files and, if required, also the executable file which needs to be invoked by the tester.

3.7.2.4. The Modified DiPerF Tester

The DiPerF tester receives its parameters in the command line, which are passed to GrenchMark. The tester decompresses the .tar archive given through its standard input and then executes a bash script from the archive. The bash script is located inside the archive and it provides the GrenchMark functionality. When testing a specific web service or peer-to-peer application, the script first starts the corresponding application.

3.7.2.5. The Modified GrenchMark Module

At first, a workload description file is generated in a format specific to GrenchMark by a Python script **wl-gen.py**. The file contains parameters and job types which are known by GrenchMark. The Python file **wl-submit.py** reads the XML file written by **wl-gen.py** and actually submits the jobs for execution. The wl-submit.py file receives extra parameters (test id, project id, tester id, start time) used for reporting the results.

The jobs are submitted for execution at specific times and a thread pool is used for submitting the jobs. A watchdog is used to check for threads which might have been blocked waiting for their job to execute and in order to report periodic statistical results for each thread.

3.7.2.5.1. The Thread Pool

The original behavior of the worker threads in the thread pool has been modified. In the original GrenchMark, each thread would get a job request from a job request queue and then execute a callback function given as a parameter in the job request. The callback function would actually submit the job for execution, compute all the needed values and return a result object to the worker thread. Now, the work of the callback function is partly done inside the worker thread. The callback functions is passed as a parameter a function of the worker thread (called **runningProcess**), which is called right before submitting the job. This function inside the thread actually submits the job and computes the most important values and the callback function compute only the remaining values contained in the result object.

Each worker thread has a Cstats object for each metric it computes. This object is fed individual values computed for each job and is used to compute statistical values for the corresponding thread.

In order to obtain the process ID of the executed job and a pipe to its standard output, the worker thread uses an object of type subprocess, contained in the popen5 package [Popen5]. Currently, Python does not offer any possibility to obtain both the process ID and a pipe connected to the standard output.

A job is not executed directly. Instead, a wrapper is being used, called **waiter.py**. This wrapper changes its process group ID and then executes the job. This is useful in case the job spawns many processes and then blocks, because using a single kill command, all the spawned can be killed, because they would be part of the same (known) process group. This approach is useless in case the job changes its process group ID itself, in which case the process group will not be known or if each process spawned by the job changes its process group ID.

3.7.2.5.2. The Watchdog

The WatchDog is implemented as an extension of the class of worker threads, because it has a similar behavior. It periodically checks if any threads are blocked waiting for their associated job to finish execution. If there are any threads blocked for a period longer than a specified amount of time, the job is killed by the watch dog. After the job is killed, the worker thread regains control as if the job had terminated normally. By inspecting the return code, the worker thread could notice that the job was, in fact, killed by the watchdog.

The watchdog has another important function. It periodically collects statistical information from the worker threads, for every computed metric. Currently, there are 5 metrics computed: Run Time, Response Time, Waiting Time, Time to Job Failure and Time To Job Completion. Each metric is computed on a per thread basis.

3.7.2.6. The Database Module

The database module is implemented in Python (the file **dbpy.py**). It receives as a single command line argument a line which contains information to be entered into the database. Information is encoded. The fields are separated by the character having ASCII code 1 and the line may contain a prefix which specifies the table into which the information will be inserted (or updated). The DiPerF controller invokes the database module every time it receives a line containing information to be entered into the database (such a line has a particular prefix).

The database module interacts with a MySQL database containing specific tables.

3.7.2.7. The Metrics

There are 5 metrics computed: Run Time, Response Time, Waiting Time, Time To Job Completion and Time To Job Failure. All of them are computed on a per thread basis. Currently, because of insufficient information, the waiting time is always considered to be 0 and the run time is always equal to the response time. The relations ship between them is: Response Time = Waiting Time + Run Time. However, once a job is submitted, there is no module implemented to measure the waiting time (or get it from the resource manager), so we consider the waiting time to be 0.

The Time To Job Failure metric is computed for approximately equal intervals of time. For each failed job, the difference between the moment it failed and the previously moment when a job has failed (or the beginning of the time interval) is computed and passed to the corresponding Cstats module. This metric is a measure of how frequently job failures occur.

The Time To Job Completion metric is computed in a similar way. For every correctly completed job, the difference between the previous moment when a job was completed correctly (or the beginning of the time interval) is computed and passed to the corresponding Cstats module.

3.7.3. Validation and Testing

3.7.3.1. Validation

We have validated the implementation on the DAS-2 environment [Bal et al., 2000], a wide-area distributed system consisting of 200 Dual Pentium-III computer nodes. The environment is built out of clusters of workstations, which are interconnected by SurfNet, the Dutch university Internet backbone for wide-area communication, whereas Myrinet, a popular multi-Gigabit LAN, is used for intra-cluster communication. The clusters are located at five Dutch Universities and from this point of view it can be considered as an experimental Grid system operating in the Netherlands. The validation focus was to show that ServMark can operate correctly, that is, that it can generate complex tests involving several test nodes, run the tests, obtain and analyze the results, and store all the produced output. We have used one node in each cluster to validate our implementation, by running on each of them several ServMark test nodes. Throughout the validation tests, ServMark displayed the expected functionality.

3.7.3.2. Testing

In order to test the ServMark implementation, we chose to evaluate the performance of 6 web servers: Apache, Null HTTPD, Apache Tomcat, Nweb, Jetty and Awhttpd. The purpose of this testing scenario was to prove the capabilities of our system and not to establish which of these web servers is the best, from an absolute point of view.

3.7.3.2.1. Experimental Setup

The ServMark "core" was installed on **s8.diperf.cs.uchicago.edu** , a machine located at the University of Chicago Computer Science Department. The characteristics of this machine are presented in table 3-1.

Table 3-1. The Characteristics of the Machine on which the ServMark "Core" was Installed.

OS	Linux SuSE
GCC version	3.3.3
Python version	2.3.3
Database Server	MySQL
MySQL version	4.0.18

The web servers were started on **alice01.rogrid.pub.ro**, a machine located at the Politehnica University of Bucharest, Faculty of Computer Science. The characteristics of this machine are presented in table 3-2.

Table 3-2. The Characteristics of the Machine on which the Web Servers were Started.

OS	Linux
GCC version	3.2.3
Java version	1.5 SE

3.7.3.2.2. Test Setup Overview

For every test, we used 22 testers, each executing 100 requests, generated using a Poisson distribution. The testers were spawned on machines which are part of PlanetLab. PlanetLab currently consists of 700+ machines, hosted by 350+ sites, spanning over more than 25 countries. Most of the machines are hosted by research institutions, although some are located in co-location and routing centers (e.g., on Internet2's Abilene backbone). All of the machines are connected to the Internet. All PlanetLab machines run a common software package that includes a Linux-based operating system; mechanisms for bootstrapping nodes and distributing software updates; a collection of management tools that monitor node health, audit system activity, and control system parameters; and a facility for managing user accounts and distributing keys. The advantage to researchers in using PlanetLab is that they are able to experiment with new services under real-world conditions, and at large scale.

For each test, the testers were selected to run on hosts from North and South America, Asia, and Europe, simultaneously.

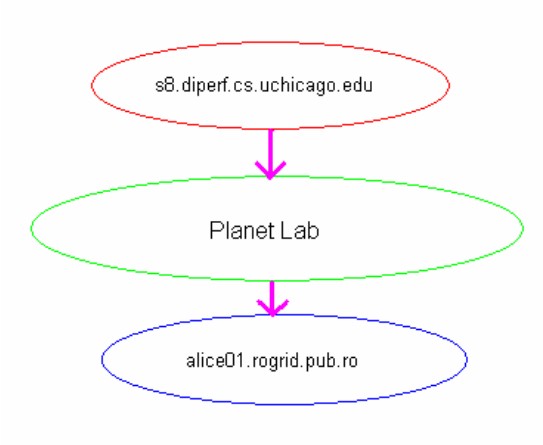


Fig. 3-20. The Test Setup Overview.

A job which was running for more than 25 seconds was considered to be blocked and was, subsequently, killed. The watchdog gathered statistical information from the worker threads approximately every 15 seconds.

3.7.3.2.3. Test Results

Table 3-3 presents the statistical values for the response time of the 6 web servers we tested.

Table 3-3. The Response Times computed for the 6 Web Servers (in Seconds).

Web Server	Average(Standard Deviation)	Minimum	Maximum	Weighted Average
Apache	1.0779 (0.647)	0.0810	16.5440	1.0969
Null HTTPD	0.9442 (0.482)	0.1244	30.4872	0.9495
Apache Tomcat	1.3617 (0.732)	0.1724	24.2665	1.3930
Nweb	0.9731 (0.565)	0.1293	10.9908	1.0152
Jetty	10.0745 (1.210)	0.2651	35.4375	9.0297
Awhttpd	1.1739 (0.558)	0.1242	29.5580	1.0117

The amount of data stored in the database generated by ServMark for each web server was estimated to be about 1.6 megabytes. However, the size of the information for each submitted job depends on the output of each test job, which is application-specific, and cannot be reduced by the testing infrastructure (e.g, by GrenchMark).

The test parameters we chose (22 testers and 100 queries per tester) were large enough to make good use of the resources available at the testing nodes. However, they may not have been stressing enough to make the web servers use all of their resources.

Chapter 4 – Real-Time Centralized Scheduling of Data Transfer Requests

Many types of communication flows require hard QoS guarantees, like a guaranteed minimum bandwidth, maximum delay, or a guaranteed data delivery deadline. Such constraints are very difficult to provide when the network infrastructure is shared by multiple users who may unpredictably generate network traffic and, thus, consume network resources. Hard QoS guarantees can only be provided when the network infrastructure within which the communication flows are delivered is under the control of a single entity. Moreover, (background) traffic generation is also controlled (or limited) by the same entity. Such a centralized control provides the opportunity for centralized scheduling of the concurrent communication flows within the network. At the moment, Internet Service Providers (ISPs) are the most likely to meet the condition mentioned above (as they own and can have full control over their network infrastructure). Although they do not currently use their network infrastructure in order to provide QoS guarantees to their users, they may some day decide to provide QoS-constrained data transfer services, given enough economic incentives.

In the first part of this chapter we introduce several new types of data transfer services which may provided by a *data transfer service provider*, as well as guidelines for managing their risks and prices. The rest of the chapter is dedicated to presenting algorithmic techniques for computing efficient schedules under multiple (types of) constraints. The original contributions presented in this chapter were published in [Andreica, Deac and Tipa, 2009], [Andreica and Ţăpuş, 2009e], [Andreica and Ţăpuş, 2009f], [Andreica, Tîrşa, Ţăpuş, Pop and Dobre, 2009], [Andreica, Tîrşa and Ţăpuş, 2009b], [Costan, Stratan, Tîrşa, Andreica and Cristea, 2009], [Mogoş and Andreica, 2009], [Andreica, 2009a], [Andreica, Andreica and Cătăniciu, 2009], [Andreica, Andreica and Ardelean, 2009], [Carpen-Amarie, Andreica and Cristea, 2008], [Andreica and Tîrşa, 2008], [Andreica and Ţăpuş, 2008b], [Andreica and Ţăpuş, 2008d] , [Andreica and Ţăpuş, 2008e], [Andreica and Ţăpuş, 2008f], [Andreica and Ţăpuş, 2008g], [Andreica and Ţăpuş, 2008j], [Andreica and Ţăpuş, 2009b], [Andreica, 2008b] and [Andreica, 2008c].

4.1. Context and Types of Data Transfer Services

The context in which dedicated data transfer services can be provided is that in which the entire network infrastructure is owned by the service provider. Thus, we define the network as a graph composed of n vertices and m edges. A vertex is either a user computer or the device through which the user connects to the network infrastructure of the provider (e.g. a cable modem or ADSL modem), a router, a switch, or any other network device. An edge between two vertices corresponds to a physical link between the two vertices and has an associated latency, bandwidth and, possibly, a cost (if we have a full-duplex link, then we need a latency, bandwidth and cost value for each direction). We only consider point-to-point links and we ignore shared network media (e.g. Ethernet on a bus topology, or wireless) on purpose.

Note that the provider does not need to map its entire network in this graph. In this case, a vertex may stand for a group of network devices or for part of a network. However, the physical characteristics of the network links (graph edges) should be adjusted accordingly (e.g. an edge between two vertices i and j may now stand for multiple physical links between the network devices of the groups corresponding to the two vertices; in such a case, the bandwidth for each direction of the edge could be the sum of the bandwidths of the physical network links for that direction, but it's difficult to say what the "combined" latency should be). We consider that all the traffic that exists in the network is the result of using some of the data transfer services which we describe next.

The most common type of data transfer service, which is provided by most Internet Service Providers (ISPs) nowadays, is the following. They guarantee a maximum upload and download

bandwidth to the users and make no guarantees for anything else. The user can choose a quality level – the larger the quality level, the larger the upload and download bandwidths are. The users are charged a flat fee *F(j)* (for quality level *j*) per month as long as a combination *C* of the total upload traffic *U* and download traffic *D* (during the current month) does not exceed an upper bound *B* (*C* may be *U+D*, or some other function chosen by the provider). Then, the users are charged an extra fee, usually directly proportional with the excess: *E(j)·(C-B)*. This business model works well, but it is not useful in any of the situations requiring QoS guarantees. This is because these are only best-effort services and provide no end-to-end guarantees. The following types of services focus on providing end-to-end guarantees. The best model for using such services is that in which the users (or user applications) submit data transfer requests to a central scheduler (like in [Andreica, Tîrşa, Ţăpuş, Pop and Dobre, 2009] and [Andreica and Tîrşa, 2008]).

The first type of requests is given by *fixed bandwidth-fixed duration* requests. Such a request has the following parameters: $(t_1, t_2, B, D, lmax, s, f)$, meaning that the data transfer requires a minimum amount of bandwidth *B* for a duration of time *D*, the earliest possible starting time of the transfer is t_1, the latest possible finish time is t_2, and the transfer takes place between nodes *s* and *f*. The scheduler must assign to the request a (directed) path from *s* to *f* in the graph and a time interval *[ts, ts+D]* in which the required amount of bandwidth is available on each edge on the path (in the direction from the source to the destination) and the sum of latencies of the edges on the path is at most *lmax*. Then, data is transferred on the assigned path and during the assigned time interval at (at most) the requested bandwidth.

A second type of requests is given by *fixed data-fixed duration* requests. The parameters of such a request are: $(t_1, t_2, TD, dataid, s, f, o)$, meaning that it needs to transfer *TD* bytes of data, the transfer takes place between nodes *s* and *f*, and the transfer must occur between time moments t_1 and t_2, but the transfer speed does not have to be constant (i.e. it can vary in time; all that matters is that the moment *tf* when the last byte of data arrives at the destination is $\leq t_2$, and we do not start transferring the first byte of data before t_1). The data is initially located at the source node *s* and is identified by an identifier *dataid* (e.g. file name, location and offset within the file). The data does not have to be transferred on a single path – this is up to the scheduler; but the transfer must be reliable (i.e. all the data must reach its destination). The *o* parameter is a boolean flag which indicates if the data must be received in order (*o=true*) or if it can be reordered at the destination (*o=false*) by adding extra information regarding its position in the flow to every transferred packet.

Another type of requests is given by *fixed bandwidth-variable duration* requests. The parameters of such a request are: $(t_1, B, lmax, s, f)$, meaning that the scheduler needs to allocate to the request a (directed) path from *s* to *f*, starting at time moment t_1 and for an indefinite duration. A bandwidth of at least *B* must be available on every edge of the path (in the corresponding direction, from *s* to *f*) and the sum of the latencies of the edges on the path must be at most *lmax*.

So far, we only considered independent and unrelated requests. However, the two types of *fixed duration* requests can be extended as follows. The scheduler receives a group of *M* requests (*r(1), …, r(M)*) which are connected by precedence constraints. To be more precise, besides satisfying the constraints of each request, we are also given a directed acyclic graph (DAG) which has a vertex *i* for every request *r(i)*. We have a directed edge from vertex *i* to vertex *j* if the data transfer corresponding to request *r(j)* must start only after the data transfer corresponding to request *r(i)* is complete. The group may contain both *fixed bandwidth-* and *fixed data- fixed duration* requests.

4.2. Guidelines for Managing Prices and Risks

Economic feasibility is strongly related to the profits the data transfer service provider can obtain from its customers. Thus, the pricing policy plays an important role. Depending on the (expected) number of customers, the provider may choose a fixed pricing policy or may negotiate the price for every data transfer request (it may even present multiple alternatives with different prices to the customer).

In the case of *fixed bandwidth-fixed duration* requests, the price should be proportional to *B*

and D, and inversely proportional to the *slack* (t_2-t_1-D) and *lmax*. That is, the larger the requested bandwidth and duration are, the larger the price should be, and the smaller the upper bound on latency and the slack are, the larger the price should be. Prices may also depend on the time interval $[t_1,t_2]$. If it is difficult to find a time interval of length D where a path satisfying the Quality-of-Service (QoS) constraints exists, then the price should be higher. Moreover, the price can be proportional to the amount of already reserved bandwidth on the edges on the chosen path (the higher the bandwidth of the path is utilized, the higher the price).

Fixed data-fixed duration requests should be charged proportional to the total transferred data (TD), and inversely proportional to the length of the time interval $[t_1,t_2]$. Moreover, if the data should be delivered in an ordered fashion, the price should be higher (as there are more constraints imposed on the provider).

Fixed bandwidth-variable duration requests are slightly more complicated. Obviously, the price should be proportional to B, to the actual duration of using the service, and inversely proportional to *lmax*. A combination of fixed and variable costs could be used here. For instance, if the service is used for a duration of at most D, then the price could be CF; otherwise, if the usage duration is $D'>D$ the price will be $CF+CV\cdot(D'-D)$ (CF and CV depend on the other parameters of the request).

Note that not all of the requests may be satisfied, as the provider may not have sufficient resources to accommodate all the requests. When receiving a request, if it can be satisfied, the provider should choose the price also based on the risk that this request may force the rejection of future requests which might bring larger revenues (we consider that once accepted, a request cannot be cancelled or rejected later).

Fixed bandwidth-variable duration requests present the highest risk, as resources might need to be reserved for a long time in order to make sure that the request is satisfied (if, however, the provider is over-provisioned compared to the actual customer demand, these requests may be the most desirable, as they might use the network resources for larger time intervals and, thus, they may be favoured in some sense). *Fixed bandwidth-fixed duration* requests present the second highest risk and fixed *data-fixed duration* requests present the lowest risk (those with unordered data delivery are less risky than those with guaranteed ordered data delivery). However, handling a DAG of a group of requests presents significantly higher risks than handling independent requests. Thus, the provider should use a good risk model, as this will influence its pricing policy.

A forecast and a simulation component should be included in the risk model. The forecast component should identify patterns of the parameters of the requests received so far and patterns of behaviour for the fixed bandwidth-variable duration requests (e.g. estimations of the actual durations). The forecast component should be used as follows. Given all the available information regarding the requests and a time interval $[t_1,t_2]$, the forecast component should be able to generate a list of *fake* requests, which it estimates that might be received during the interval $[t_1,t_2]$. Then, when deciding the price of a newly received request, we use the simulation component to estimate the overall revenue if the request were accepted (ignoring its price) and the overall revenue if the request were rejected. The simulation is run for a carefully chosen duration (e.g. it simulates T seconds or minutes in the future) and uses as input the list of fake requests estimated by the forecast component for the interval *[present time moment, present time moment+T]*, and the currently scheduled requests. The price of the request should be chosen such that the revenue in case of not accepting the request is not larger than the revenue in the case of accepting request (but ignoring its price) plus the price of the request.

The forecast component may also be used differently. It could generate $K\geq 1$ lists of fake requests for a given time interval $[t_1,t_2]$ and it could assign a probability of occurrence *prob(i)* to every list i ($1\leq i\leq K$). Then, the simulation is run for each of the K lists, a revenue $R(K)$ is computed for every list and then an expected revenue ER=the sum of the values $prob(i)\cdot R(i)$ is computed and used (we assume that the sum of the values $prob(i)$, $1\leq i\leq K$, is 1).

4.3. Online Scheduling of Fixed-Bandwidth Fixed-Duration Data Transfer Requests on a Single Network Link

In this section, we will consider a particular situation, in which the network is composed of only two nodes, connected by a single link. Fixed-bandwidth fixed-duration data transfer requests are sent from both nodes to a central scheduler (the maximum latency argument is ignored here). We will consider two *bandwidth models* for the requests. In the first model, each request asks for the whole bandwidth of the link and, thus, we can schedule at most one data transfer at a time. In the second model, each request asks for a fraction of the link's bandwidth. The first model is, obviously, a particular case of the second model, but it can be handled more efficiently in some situations. The requests will be handled one at a time (i.e. online).

The focus of this section is on techniques for deciding quickly if a request can be satisfied and, if it can, for creating a reservation for the request. We will present several data structures which can be used for this purpose, which bring successive performance improvements. We will start with some well-known data structures (for which the novelty consists of the way they are used) and we will end up with a new data structure, presented in [Andreica and Ţăpuş, 2008b]. All the data structures assume that time is divided into discrete, equally-sized, time slots and they will work with time slot numbers (time slots are numbered starting either from *0* or from *1*, in their chronological order), rather than with time *moments*.

4.3.1. The Time Slot Array

We will first consider the first bandwidth model (i.e. each data transfer uses the full bandwidth of the link). A common approach is to divide the time horizon into m equally-sized time slots and build a time slot array [Burchard, 2005] over these slots. For each time slot t ($1 \leq t \leq m$), the array ts contains an entry $ts[t]$, which can be either *0* (no transfer is scheduled during this time slot) or *1* (a transfer was scheduled during this time slot). Using the time slot division, each transfer is started only at the beginning of a time slot and lasts for an integer number of consecutive time slots (even if the last time slot is not fully occupied, it is still marked as being fully occupied). In order to obtain a good performance, the time horizon must be divided into a large number of time slots (a fine-grained time resolution). In this situation, however, an important aspect to consider is the time it takes to traverse the time slot array for each request.

We consider two operations to be performed on the array: a *query* and an *update*. A query verifies if a request can be granted and an update sets all the time slot entries of a data transfer to the same value (*1*, when scheduling a transfer; *0*, when canceling a transfer). The pseudocode of these operations is described below (the time parameters are converted into time slots):

```
TSAQuery(t1, t2, D):
nfree=0
for t=t1 to t2 do { // t=ES, ES+1, …, LF-1, LF
  if (ts[t]=0) then {
    nfree=nfree+1
    if (nfree=D) then return [t-D+1, t]
  } else nfree=0
}
return "no interval found"

TSAUpdate(ts, D, value):
for t=ts to ts+D-1 do ts[t]=value // or ts[t]+=value
```

Pseudocode 4-1. Update and Query Functions for the Time Slot Array.

The time complexity of each operation is $O(m)$. The time slot array can be easily enhanced in order to support the second bandwidth model. In this case, it is possible for multiple transfers to be scheduled simultaneously, as long as the maximum bandwidth of the link is not exceeded. The

entry *ts[t]* of a time slot represents, in this case, the available bandwidth during that time slot. The update and query functions run in *O(m)* time.

```
DSQuery(t1, t2, D):
nfree=0; t=t1
while (t≤t2) do {
  if (ts[t]=0) then {
    nfree=nfree+1
    if (nfree=D) then return [t-D+1, t] else t=t+1
  } else {
    nfree=0
    t=right[Find(t)]+1
  }
}
return "no interval found"

DSUpdate(tstart, D, value=1):
for t=tstart to tstart+D-1 do {
  ts[t]=value
  if ((t>1) and (ts[t-1]=value)) then Union(t-1, t)
}
if ((tstart+D≤m) and (ts[tstart+D]=value)) then Union(tstart+D-1, tstart+D)
```

Pseudocode 4-2. Update and Query Functions for the Disjoint Sets Data Structure.

4.3.2. Disjoint Sets for Non-Cancellable Reservations

Using the first bandwidth model and if the data transfer reservations cannot be cancelled, we can use a disjoint sets data structure [Cormen, Leiserson, Rivest and Stein, 2001] in order to maintain the maximal intervals of time slots with an entry equal to *1* in the time slot array. This data structure provides two operations: *Find(t)*, which returns the representative of the disjoint set containing element *t* and *Union(a,b)*, which combines the sets of the elements *a* and *b* into one set (if they are not already in the same set).

For each set representative *sr* we maintain two values: *left[sr]* and *right[sr]*, the left and right endpoints of the interval of time slots represented by the set. Within the *Union* procedure we compute the set representatives of the elements *a* and *b*, sr_a and sr_b (sr_a=*Find(a)* and sr_b=*Find(b)*). Then, using some of the well-known heuristic criteria (like *union by rank* or *union by size*), one of the two representatives (call it sr_u) will be selected as the representative of the combined set. We will set $left[sr_u] = min\{left[sr_a], left[sr_b]\}$ and $right[sr_u]=max\{right[sr_a], right[sr_b]\}$.

The total time complexity of the updates is $O(m \cdot \alpha(m,n))$ (where $\alpha(m,n)$ is the inverse of the Ackermann function) and the query time is reduced, because we can jump over large intervals of occupied time slots.

4.3.3. Using an Algorithmic Framework based on Block Partitioning

In this section we will introduce an algorithmic framework for the block partitioning method. We consider an array of *n* cells, numbered from *0* to *n-1*, where each cell has a value v_i (each cell corresponds to a time slot). We will divide the *n* cells into *n/k* blocks of size *k* (we assume that *k* is a divisor of *n*; if it is not, *n* can be extended to be a multiple of *k* or the last block may contain fewer cells). The blocks are numbered from *0* to *(n/k)-1* (or *n/k*, if *n* is not a multiple of *k*).

The cells *0, …, k-1* belong to block *0*, the cells *k, …, 2·k-1* belong to block *1*, …, the cells *(i-1)·k, …, (i·k)-1* belong to block *i-1*, and so on. Thus, the cell *j* belongs to block *(j div k)* (integer division). For simplicity, we store for each block *B* the first and last cells of the block (*left[B]* and *right[B]*). Using this partitioning, we can support several update and query functions in *O(k+n/k)* time. By choosing *k=sqrt(n)*, we will have *O(k+n/k)=O(sqrt(n))*; *sqrt(x)* denotes the (integer part of

the) square root of *x*. Queries consist of computing a function on the values of a range of cells *[a,b]* (range query) or on retrieving the value of a single cell (point query).

Range Query(a, b): compute $qFunc(v_a, v_{a+1}, \ldots, v_b)$.

Analogously, we have point and range updates:

Range Update(u, a, b): $v_i = uFunc(u, v_i)$, $a \le i \le b$.

The *qFunc* function must be binary and associative, i.e. $qFunc(v_a,..,v_b) = qFunc(v_a,qFunc(v_{a+1},..,qFunc(v_{b-1}, v_b)..))$ and $qFunc(a,qFunc(b,c)) = qFunc(qFunc(a,b),c)$. We must also have $uFunc(x,y) = uFunc(y,x)$. Only values v_i with *O(1)* size are considered (numbers and tuples with a fixed number of elements). *uFunc* and *qFunc* must be able to handle *uninitialized* arguments. If one of their arguments is *uninitialized*, they must simply return the other argument; this part will be intentionally left out of the functions' descriptions. The algorithmic framework consists of the functions from Table 4-1.

Table 4-1. Algorithmic Framework Functions.

Update Functions	Query Functions
BPpointUpdate	BPpointQuery
BPrangeUpdate	BPrangeQuery
BPrangeUpdatePoints	BPrangeQueryPoints
BPrangeUpdatePartialBlock	BPrangeQueryPartialBlock
BprangeUpdateFullBlock	BPrangeQueryFullBlock

In order to perform a range update, we will call the *BPrangeUpdate* function with the corresponding parameters (the update value *u* and the update interval *[a,b]*). This function splits the update interval into three zones: the first block B_a intersected by the interval (containing the cell *a*), the last block B_b intersected by the interval (containing the cell *b*) and all the blocks in between B_a and B_b (the inner blocks).

The blocks B_a and B_b may not be fully contained inside the interval: they will be updated in *O(k)* time (partial update). All the inner blocks are fully contained inside *[a,b]*: they will be updated in *O(1)* time each (full update). Since there are *O(n/k)* such blocks, the overall complexity of a range update will be *O(k+n/k)*.

The range query function (*BPrangeQuery*) works similarly. For each block *B* we will maintain two values: *uagg* and *qagg*. *uagg* is the aggregate of the update parameters of the function calls which updated all the elements of *B* (for which *B* was an inner block). *uagg* is reset to an *uninitialized* value on each partial update of the block. *qagg* is the answer to the query function called on all the elements of *B*.

The point update and query functions are: *BPpointUpdate* and *BPpointQuery*. The framework also uses a "multiplication" operator *mop*, which computes the effects of an update operation upon the query result on a range of cells. This operator must exist when range queries and range updates are used together, but can be ignored otherwise. When the data structure is initialized, the *uagg* value of each block is set to *uninitialized* (*qagg* is initialized with the query result on the range of the block's cells).

In the case of point queries with range updates, only the *uagg* values are meaningful; similarly, only the *qagg* values are meaningful in the case of point updates with range queries. Common update and query functions can be easily integrated into the framework. For example, with $uFunc(x,y)=(x+y)$, $qFunc(x,y)=((x+y)\ (modulo\ M))$ and $mop(u,a,b)=((u\cdot(b-a+1))\ (modulo\ M))$, we can support point and range sum queries, together with point and range addition updates. For $uFunc(x,y)=x+y$, $qFunc(x,y)=min(x,y)$ and $mop(u,a,b)=u$, we can support point and range minimum (or maximum) queries, together with point and range addition updates. We can also consider point and range multiplication updates, $uFunc(x,y)=x\cdot y$, with point and range queries: $qFunc(x,y)=((x\cdot y)\ (modulo\ M))$ (with $mop(u,a,b)=((u^{b-a+1})\ (modulo\ M))$), $qFunc(x,y)=min(x,y)$ and $qFunc(x,y)=((x+y)\ (modulo\ M))$ (with $mop(u,a,b)=u$).

With $mop(u,a,b)=u$, we can support range queries and updates for some bit functions (where $v_i=0$ or 1). For $uFunc(x,y)=(x\ or\ y)$ and $uFunc(x,y)=(x\ and\ y)$, we can have $qFunc(x,y)=(x\ and\ y)$

and *qFunc(x,y)=(x or y).*

```
BPpointUpdate(u, i):
v_i=uFunc(u,v_i)
B=the block to which the cell i belongs
qagg[B]=BPrangeQueryPoints(left[B], right[B])
BPrangeUpdate(u, a, b):
B_a, B_b=the blocks of cells a and b
if (B_a=B_b) then {
if ((a=left[B_a]) and (b=right[B_a])) then BPrangeUpdateFullBlock(B_a, u)
else BPrangeUpdatePartialBlock(B_a, u, a, b)
} else {
BPrangeUpdatePartialBlock(B_a, u, a, right[B_a])
BPrangeUpdatePartialBlock(B_b, u, left[B_b], b)
for block=B_a+1 to B_b-1 do BPrangeUpdateFullBlock(block, u)
}
BPrangeUpdatePoints(u, a, b):
for p=a to b do v_p=uFunc(u, v_p)
BPrangeUpdatePartialBlock(B, u, a, b):
BPrangeUpdatePoints(uagg[B], left[B], right[B])
uagg[B]=uninitialized
BPrangeUpdatePoints(u, a, b)
qagg[B]=BPrangeQueryPoints(left[B], right[B])
BPrangeUpdateFullBlock(B, u):
uagg[B]=uFunc(u, uagg[B])
qagg[B]=uFunc(mop(u, 1left[B], right[B]), qagg[B])
BPpointQuery(i):
B=the block to which the cell i belongs
return uFunc(uagg[B], v_i)
BPrangeQuery(a, b):
B_a, B_b=the blocks of cells a and b
if (B_a=B_b) then return BPrangeQueryPartialBlock(B_a, a, b)
else {
q_a=BPrangeQueryPartialBlock(B_a, a, right[B_a])
q_b=BPrangeQueryPartialBlock(B_b, left[B_b], b)
q=uninitialized
for block=B_a+1 to B_b-1 do q=qFunc(q, BPrangeQueryFullBlock(block))
  return qFunc(q_a, qFunc(q, q_b))
}
BPrangeQueryPoints(a, b):
q=uninitialized
for p=a to b do q=qFunc(q, v_p)
return q
BPrangeQueryPartialBlock(B, a, b):
BPrangeUpdatePoints(uagg[B], left[B], right[B])
uagg[B]=uninitialized
return BPrangeQueryPoints(a, b)
BPrangeQueryFullBlock(B):
return qagg[B]
```

Pseudocode 4-3. Functions of the Block Partitioning Algorithmic Framework.

For the *and* update, we can also have *qFunc(x,y)=(x xor y)*. We can support range *xor* updates and queries (*uFunc(x,y)=qFunc(x,y)=(x xor y)*), but with *mop(u,a,b)=(if (((b-a+1) mod 2)=0) then 0 else u)*.

In order to obtain any combination of bit functions, we notice that the result of a query depends only on the number of *0* and *1* values (cnt_0, cnt_1) in the query range: if $(cnt_1>0)$ then *or* returns *1*; if $(cnt_1 \bmod 2=1)$ then *xor* returns *1*; if $(cnt_0=0)$ then *and* returns *1*. Thus, we will work with (cnt_0, cnt_1) tuples as values. We will also consider the *conceptual values* cv_i, which are the numerical values we conceptually work with. We have $v_i=(1-cv_i, cv_i)$. A query asks for the number of *0* and *1* conceptual values in the query range and an update changes this number according to the bit function used.

bitTupleQuery(($cnt_{0,x}$, $cnt_{1,x}$), ($cnt_{0,y}$, $cnt_{1,y}$)):
return ($cnt_{0,x}+cnt_{0,y}$, $cnt_{1,x}+cnt_{1,y}$)

bitTupleUpdate((1-u, u), (cnt_0, cnt_1), func):
if *(func=and)* **and** *(u=0)* **then return** $(cnt_0+cnt_1, 0)$
else if *(func=or)* **and** *(u=1)* **then return** $(0, cnt_0+cnt_1)$
else if *(func=xor)* **and** *(u=1)* **then return** (cnt_1, cnt_0)
else return (cnt_0, cnt_1)

Pseudocode 4-4. Update and Query Functions for any Bit Function.

If the update function has the effect of setting all the values in a range to the same value *s* (range set), we will again need to work with tuples: the values v_i and the update parameters *u* will have the form *(numerical value, time_stamp)*. We need to have a *timestamp()* function which returns increasing values upon successive calls. We can use a global counter as a time stamp, which is incremented at every call. The initial numerical values are assigned an initial time stamp and every update parameter gets a more recent time stamp. The update function is:

uFunc((w_x, t_x), (w_y, t_y)):
if $(t_x>t_y)$ **then return** (w_x, t_x) **else return** (w_y, t_y)

Pseudocode 4-5. Update Function for the ‚Set' Operation.

With these definitions, a point query function call on a position *i* will return the last update parameter of an interval containing that position.

A useful range query function (used together with point updates) is finding the maximum sum segment (interval of consecutive cells) fully contained in a range of cells *[a,b]* (see [Chen and Chao, 2007] for this problem without updates). Conceptually, the value of a cell *i* is a number cv_i, but in the framework we will use tuples consisting of *4* values: *(totalsum, maxlsum, maxrsum, maxsum)*. Assuming that these values correspond to an interval of cells *[c,d]*, we have the following definitions:

$$\text{totalsum} = \sum_{p=c}^{d} cv_p \qquad \text{maxlsum} = \max_{c-1 \le q \le d} \sum_{p=c}^{q} cv_p$$
$$\text{maxrsum} = \max_{c \le q \le d+1} \sum_{p=q}^{d} cv_p \qquad \text{maxsum} = \max_{\substack{c \le q \le d \\ q-1 \le r \le d}} \sum_{p=q}^{r} cv_p \qquad \textbf{(4-1)}$$

In the framework, a value v_i will be a tuple corresponding to the interval *[i,i]*. If $cv_i<0$, then $v_i=(cv_i, 0, 0, 0)$; otherwise, $v_i=(cv_i, cv_i, cv_i, cv_i)$. The point update function changes the value of cv_i of a cell *i* and then recomputes v_i. The *qFunc* function is given below:

qFunc((t_x,ml_x,mr_x,m_x), (t_y,ml_y,mr_y,m_y)):
return $(t_x+t_y, max\{ml_x, t_x+ml_y\}, max\{mr_y, t_y+mr_x\}, max\{m_x, m_y, mr_x+ml_y\})$

Pseudocode 4-6. Query Function for the Range Maximum Sum Segment Problem.

We can use the range set update together with the range maximum sum segment query. Conceptually, each cell has a numerical value cv_i. Practically, the framework's values v_i will be tuples of the following form *(totalsum, maxlsum, maxrsum, maxsum, time_stamp)*. The update, query and multiplication functions are given below. We notice that the fundamental combination (range set update, range sum query) is also solved. However, no suitable function definitions for the combination (range addition update, range maximum sum segment query) were found.

uFunc(($total_x$, ml_x, mr_x, m_x, t_x), ($total_y$, ml_y, mr_y, m_y, t_y)):
if (t_x>t_y) **then return** *($total_x$, ml_x, mr_x, m_x, t_x)*
else return *($total_y$, ml_y, mr_y, m_y, t_y)*

qFunc(($total_x$, ml_x, mr_x, m_x, t_x), ($total_y$, ml_y, mr_y, m_y, t_y)):
return *($total_x$+$total_y$, max{ml_x, $total_x$+ml_y}, max{mr_y, $total_y$+mr_x}, max{m_x, m_y, mr_x+ml_y}, max{t_x, t_y})*

mop(($total_x$, ml_x, mr_x, m_x, t_x), a, b):
return *((b-a+1)·$total_x$, (b-a+1)·ml_x, (b-a+1)·mr_x, (b-a+1)·m_x, t_x)*

Pseudocode 4-7. Functions for the Range Maximum Sum Segment Problem with the ‚Set' Operation.

The framework's behaviour can be improved by adding a dirty flag to each block. With the dirty flag, the *qagg* value will be recomputed only "on demand" and not after every point or partial block update. We only need to replace the functions *BPpointUpdate*, *BPrangeUpdatePartialBlock* and *BPrangeQueryFullBlock* with the following definitions:

We can use the block partitioning framework we just presented in order to handle data transfer scheduling requests. For the first bandwidth model, an update operation will consist of a range set operation (setting all the time slots in a range to the same value) and a query operation will consists of a range maximum sum segment query. To be more specific, we associate a value v_t to each time slot *t*. This value will always be either *1* or -∞. An update with the parameter *value=0* sets the values of all the time slots in the interval *[ts, ts+D-1]* to *1*. An update with *value=1*, sets the slots in the same interval to -∞. A query consists of the maximum sum segment query contained in the interval $[t_1, t_2]$. Such a segment will never contain a value v_t=-∞ (unless no value equal to *1* exists in that range). Thus, the returned interval is, in fact, the longest interval of free consecutive time slots and its length can be compared to the parameter *D*. We obtain a time complexity of *O(k+m/k)* for each query and update (i.e. *O(sqrt(m))* for *k=sqrt(m)*). For t_2-t_1+1=*D* and considering the second bandwidth model, we can use the same framework. In this case, the operations used are: range addition update and range minimum query.

BPpointUpdate(u, i):
v_i=*uFunc(u,*v_i*)*
B=the block to which the cell i belongs
dirty[B]=true

BPrangeUpdatePartialBlock(B, u, a, b):
BPrangeUpdatePoints(*u, a, b*)
dirty[B]=true

BPrangeQueryFullBlock(B):
if (*dirty[B]*) **then {**
  BPrangeUpdatePoints(*uagg[B], left[B], right[B]*)
  uagg[B]=uninitialized
  qagg[B]=**BPrangeQueryPoints**(*left[B], right[B]*)
  dirty[B]=false
}
return *qagg[B]*

Pseudocode 4-8. Functions of the Block Partitioning Algorithmic Framework using a Dirty Flag.

4.3.4. Using an Algorithmic Framework based on an Extended Segment Tree

```
STpushUpdates(node):
if (node.left<node.right) then {
 STrangeUpdateNodeFit(node.lson, node.uagg)
 STrangeUpdateNodeFit(node.rson, node.uagg)
 node.uagg=uninitialized
}

STrangeUpdate(node, u, a, b):
STpushUpdates(node)
if ((a=node.left) and (node.right=b)) then STrangeUpdateNodeFit(node, u)
else {
 lson, rson = left and right son of the current tree node
 if ((a≤node.lson.right) and (node.lson.left≤b)) then
  STrangeUpdate(node.lson, u, max(a, node.lson.left),min(b, node.lson.right))
 if ((a≤node.rson.right) and (node.rson.left≤b)) then
  STrangeUpdate(node.rson, u, max(a, node.rson.left),min(b, node.rson.right))
 STrangeUpdateNodeIncl(node, u, a, b)
}

STrangeUpdateNodeFit(node, u):
node.uagg=uFunc(u, node.uagg)
node.qagg=uFunc(mop(u, node.left, node.right), node.qagg))

STrangeUpdateNodeIncl(node, u, a, b):
node.qagg=uFunc(mop(node.uagg, node.left, node.right), qFunc(node.lson.qagg,
node.rson.qagg))
// for some uFunc functions, we can just modify node.qagg directly:
// node.qagg=uFunc(mop(u, a, b), node.qagg)

STrangeQuery(node, a, b):
STpushUpdates(node)
if (a=node.left and node.right=b) then return STrangeQueryNodeFit(node)
else {
 q=uninitialized
 if ((a≤node.lson.right) and (node.lson.left≤b)) then
  q=qFunc(q, STrangeQuery(node.lson, max(a, node.lson.left), min(b, node.lson.right))
 if ((a≤node.rson.right) and (node.rson.left≤b)) then
  q=qFunc(q, STrangeQuery(node.rson, max(a,node.rson.left), min(b, node.rson.right))
 return uFunc(STrangeQueryNodeIncl(node, a, b), q)
}

STrangeQueryNodeFit(node):
return node.qagg

STrangeQueryNodeIncl(node, a, b):
return mop(node.uagg, a, b)
```

Pseudocode 4-9. Functions of the Segment Tree Algorithmic Framework.

The segment tree [Andreica and Ţăpuş, 2008f] is a binary tree structure used for performing operations on an array v with m cells. Each cell i ($1 \le i \le m$) contains a value v_i. Each node p of the tree has an associated interval *[p.left, p.right]*, corresponding to an interval of cells. If the node p is not a leaf, then it has two sons: the left son (*p.lson*) and the right son (*p.rson*). The interval of the left son is *[p.left, mid]* and the right son's interval is *[mid+1, p.right]*, where *mid=floor((p.left+p.right)/2)*.

The leaves are those nodes whose associated interval contains only one cell. The interval of the root node is *[1,m]*. The height of the segment tree is *O(log(m))*. The tree can be built in *O(m)* time. We consider the same types of (point and range) query and update operations as in the case of the block partitioning method.

In order to perform an update we call the function *STrangeUpdate* with the segment tree root as its node argument and the appropriate update parameter. In order to query the segment tree, we call the function *STrangeQuery*, with the segment tree root as its node argument and the left and right cells of the query range. A query/update range is decomposed into *O(log(m))* intervals, corresponding to *O(log(m))* "covering nodes" of the tree (the query/update call stops at these nodes and does not go further down the tree). Besides the covering nodes, all the nodes on the path from the root to each covering node are visited (*O(log(m))* nodes overall). We would like to use the segment tree in order to perform the same operations as in the previous subsection (regarding the block partitioning method), i.e. range set update and range maximum sum segment query.

```
uFunc((w_x, t_x), (w_y, t_y)):
if (t_x>t_y) then return (w_x, t_x) else return (w_y, t_y)

qFunc((w_x, t_x), (w_y, t_y)):
res=w_x+w_y // or min{w_x,w_y} or max{w_x,w_y}
return (res, max{t_x,t_y})
```

Pseudocode 4-10. Update and Query Functions for the Range Set and Range Sum Operations.

```
STrangeUpdateNodeFit(node, u):
node.uagg=uFunc(u, node.uagg)
node.dirty=true

STrangeUpdateNodeIncl(node, u, a, b):
node.dirty=true

STrangeQueryNodeFit(node):
if (node.dirty=true) then {
  if (node.left=node.right) then node.qagg=uFunc(mop(node.uagg, node.left, node.right))
  else node.qagg = uFunc( mop(node.uagg, node.left, node.right),
            qFunc(STrangeQueryNodeFit(node.lson), STrangeQueryNodeFit(node.rson))
  node.dirty=false
}
return node.qagg
```

Pseudocode 4-11. Functions of the Segment Tree Algorithmic Framework using a Dirty Flag.

Each node of the segment tree maintains two values: *uagg* and *qagg*. *uagg* is the aggregate of all the update parameters of the update calls which "stopped" at that node. *qagg* is the query answer for the interval of cells corresponding to the current node, ignoring all the update calls which "stopped" further up in the tree (an update call which "stopped" at one of the current node's ancestors affects the interval of cells of the current node, but its effects are not considered in the *qagg* field of the current node). However, in order to support the kind of range queries and updates we are interested in, we will need to use a function called *STpushUpdates*. Basically, the update aggregates are pushed down to the two sons (and then cleared) on every update and query call. We would, in fact, like to push these updates all the way towards the leaves, but doing this on every update would take *O(m)* time. Instead, we "piggy-back" future update calls and push the update aggregates "on demand", without affecting the *O(log(m))* complexity of the update/query function calls. We also use a multiplication operator *mop*, which estimates the effects of an update on a range of cells.

With this framework we can support many types of pairs of updates and queries, like all

those mentioned in the previous subsection (e.g. *uFunc(x,y)=qFunc(x,y)=(x+y)* or *uFunc(x,y)=qFunc(x,y)=((x+y) (modulo M))*, and so on). We will focus now on range set updates. In order to perform a range set update, we need to consider tuples of values. For instance, if we want to perform range sum/minimum/maximum queries, each value of a cell is, in fact a pair *(value, time_stamp)*. Successive update parameters obtain increasing time stamps. The *uFunc* and *qFunc* functions which are used by the framework are defined in Pseudocode 4-10.

We can also add dirty flags to every node of the segment tree (initially, all such flags will be set to false) and recompute the qagg values of every such node only when needed. The modified functions (except *STpushUpdates*) are shown below. However, in this case, such an approach will not bring significant savings on the running time.

4.3.5. Time Slot Groups

In this section we will present an efficient data structure which can be used for providing bandwidth guarantees to non-preemptive data transfers on a single network link, subject to time constraints, in the following context: applications submit bandwidth reservation requests to a bandwidth broker which either satisfies the requests or rejects them. The data structure divides the time horizon upon which bandwidth reservations are performed into T discrete equally-sized time slots and supports efficiently the following types of operations:

- ***find*(s_1,s_2,D,B)** - finds a time slot interval $[s,s+D-1]$, where at least a given amount of bandwidth B is available during every time slot of the interval, subject to the following QoS constraints: the length of the interval is D time slots, the earliest possible starting time slot is s_1 and the latest possible finish time slot is s_2 (i.e. $s_1 \le s \le s+D-1 \le s_2$); this, in fact, a fixed-bandwidth fixed-duration data transfer request (except that the parameters were named differently)
- ***reserve*(s_1,s_2,B)** - decreases by B the available bandwidth for each slot within the time slot interval $[s_1,s_2]$ (if the value of B is negative, an increase takes place)

The reserve (update) operation takes $O(k+(T/k))$ time and the find (query) operation takes $O(k+(T/k)\cdot log(k))$ time, where $1 \le k \le T$ is a user-defined parameter (e.g. a constant value or a function $f(T)$). Some situations where this functionality is useful are the transfer of multimedia streams to customers who are only available within some specific time intervals or the transfer of large data files in Grids and other distributed systems.

4.3.5.1. The Enhanced Time Slot Array

We will first show how a time slot array (TSA), *availbw*, can support the two operations. The slots are numbered from *0* to *T-1* and the time parameters and the operations' results are expressed in time slots. We give the *reserve* function below:

```
reserve_TSA(s1 , s2 , B):
for s = s1 to s2 do availbw[s] = availbw[s] – B
```

Pseudocode 4-12. The *reserve* Function for the Time Slot Array.

The bandwidth of a time slot interval $[sa, sb]$ is:

$$B_{interval} = \min_{sa \le s \le sb} \{availbw[s]\}. \quad \textbf{(4-2)}$$

In the *find* function we are looking for an interval of D slots, with a bandwidth greater than or equal to B. We will traverse the $[s_1,s_2]$ interval with a sliding window consisting of D time slots and maintain a min-heap with the available bandwidths of the time slots in the window. When we move the right end of the window one position to the right, from s ($s \ge s_1+D-1$) to $s+1$, we remove from the heap the (leftmost) time slot $s-D+1$ (which now falls outside of the window) and insert into the heap the time slot $s+1$.

We can reduce the time complexity of the *find* operation from $O(T\cdot log(T))$ to $O(T)$, if we replace the min-heap by a double-ended queue (deque) [Berman et al., 2004] which stores *(time slot,*

available bandwidth) pairs. The elements of the deque are sorted increasingly according both to the bandwidth and the time slot. The first element of the deque (*deque.getFirst()*) is always the one with the minimum bandwidth among the slots inside the current window. As the right end of the window slides to the next slot *s*, all the pairs at the end of the deque whose bandwidths are larger than the available bandwidth of slot *s* are removed. The element at the front of the deque is removed when it falls outside of the sliding window. This takes *O(T)* time, because we insert each time slot once and remove it at most once from the deque. The last element of a deque is obtained as the result of the call *deque.getLast()*. Checking if a deque is empty is performed by calling the function *deque.gisEmpty()*.

The enhanced time slot array handles differently only the query and update function calls referring to all the slots (between $s_1=0$ and $s_2=T-1$). These calls will be named *full-period* calls. A *full-period update* needs to decrease the bandwidth of each time slot by the same value *B*. Instead of doing this, only the value of a variable called *globalbw* is modified. Thus, the real available bandwidth of each slot *s* will be *availbw[s]+globalbw*. For each *full-period query*, we will find the answer in *O(1)* time, by using a previously computed array *sibw*. *sibw[L]* stores the maximum bandwidth of an interval of *L* slots and the actual interval.

```
find_TSA(s1 , s2 , D, B):
deque = empty
(ts,Bmax)=(-∞,-∞)
for s = s1 to s2 do {
  while ((not deque.isEmpty()) and (deque.getLast().value≥availbw[s])) do
    deque.removeLast()
  deque.addLast((time_slot = s, value = availbw[s]))
  if ((s–s1+1>D) and (deque.getFirst().time_slot=s-D)) then deque.removeFirst()
  if ((s–s1+1≥D) and (deque.getFirst().value≥B)) then
    (ts, Bmax)=(s-D+1, deque.getFirst().value)
}
if (Bmax≥B) then return (intv=[tst=ts, tend=ts+D-1], maxbw=Bmax)
else return (intv="no interval found", maxbw=Bmax)
```

Pseudocode 4-13. The *find* Function for the Time Slot Array.

The *sibw* array will be computed after every non-full-period update. An efficient way to compute the *sibw* array would be to sort the values of the available bandwidths of the *T* time slots increasingly into an array called *values*. We will maintain a data structure (balanced tree) of (disjoint) time slot intervals which, initially, contains only one time slot interval consisting of all the *T* time slots. Then we will traverse the sorted *values* array. Every value will split the time slot interval inside which it is located into two time slot intervals (or one if it is located at the end of some time slot interval, or zero if the time slot interval consisted of just one slot).

We will also have a binary max-heap with the length of the current time slot intervals. Using the balanced tree, we can retrieve easily the time slot interval into which a given time slot resides. Before performing a split at the i^{th} value, we will retrieve the maximum value *L* from the max-heap, meaning that a time slot interval of *L* time slots having a bandwidth equal to *values[i]* exists. We will store this interval at the position *sibw[L]*. After performing the split, the time slot interval which was split is removed both from the heap and the balanced tree and will be replaced by the resulting smaller intervals (which are inserted in the tree and the heap). After traversing all of the values (and performing all the splits), we traverse the array *sibw* from the largest length to the smallest one; if no interval was stored for a length *L*, then we use the time slot interval for length *L+1*, removing from it the leftmost or the rightmost time slot. The overall time complexity is $O(T \cdot log(T))$, because each of the *T* splits takes *O(log(T))* time.

We can compute the *sibw* array more efficiently, in *O(T)* time. If we consider the available bandwidth of a time slot *s* as the "height" of that time slot, we obtain a histogram. We can find all the *O(T)* maximal area rectangles inside the histogram in *O(T)* time, by adapting an algorithm

presented in [Vandevoorde, 1998] for finding the largest area rectangle full of ones in a binary matrix. The algorithm maintains a stack of *(time slot, bandwidth)* pairs, sorted increasingly both according to the slot number and the bandwidth value. If, after processing a slot *s*, the stack contains some pair *(s',B)*, then the time slot interval *[s',s]* has bandwidth *B* and is the longest interval ending at slot *s* having this bandwidth. The pseudocode of the (first version of the) functions is presented in Pseudocode 4-14.

```
find_ETSA_v1(s1 , s2 , D , B):
if (s1=0) and (s2=T-1) then {
  if (sibw[D].bw + globalbw ≥ B) then
    return (intv=[tst=sibw[D].s1 , tend=sibw[D].s2], maxbw=sibw[D].bw + globalbw)
  else return "no interval found"
} else return find_TSA(s1 , s2 , D , B-globalbw)

reserve_ETSA_v1(s1 , s2 , B):
if ((s1=0) and (s2=T-1)) then globalbw=globalbw–B
else {
  reserve_TSA(s1 , s2 , B)
  computeSibw()
}

computeSibw():
stack=empty; availbw[T]=-∞
sibw[L]=undefined, for each  L=1,…,T
for s=0 to T do {
  lslot=s
  while ((not stack.isEmpty()) and (stack.top().bw≥availbw[s])) do {
    L=s-stack.top().leftmost_slot; H=stack.top().bw
    if ((H>availbw[s]) and ((sibw[L]=undefined) or (sibw[L].bw<H))) then {
      sibw[L].s1= stack.top().leftmost_slot
      sibw[L].s2=s-1; sibw[L].bw=H
    }
    lslot=stack.top().leftmost_slot; stack.pop()
  }
  stack.push((leftmost_slot = lslot , bw=availbw[s]))
}
for L=T-1 downto 1 do
  if (sibw[L]=undefined) then {
    sibw[L].s1=sibw[L+1].s1
    sibw[L].bw=sibw[L+1].bw
    sibw[L].s2=sibw[L+1].s2 -1
  }
  compute minbw, minBwLtoR and minBwRtoL
```

Pseudocode 4-14. Functions for the Enhanced Time Slot Array - Version 1.

Because by using *ETSA_v1*, updating short intervals takes a longer time, we will maintain a *dirty* flag for the *sibw* array. This way, after each non-full-period update, the *dirty* flag is set and at the next full-period query, the *sibw* array is recomputed. The complexity of the *find* function for a full-period query becomes *O(1)* in an amortized sense. The second version of the functions (*ETSA_v2*) is shown in Pseudocode 4-15.

The enhanced time slot array is augmented with three extra functions, *getMinBw*, *getMinBwLeftToRight* and *getMinBwRightToLeft*, computable in *O(1)* time. *getMinBw* returns the minimum bandwidth of any time slot. The minimum value in the *availbw* array, *minbw*, is computed in the *computeSibw* function (in *O(T)* time). The real minimum bandwidth is the sum of

minbw and *globalbw*. In the *computeSibw* function we also compute in *O(T)* time two arrays: *minBwLtoR* and *minBwRtoL*, defined as follows:

$$\text{minBwLtoR}[i] = \min_{0 \le j \le i}\{\text{availbw}[j]\}\ , \qquad \textbf{(4-3)}$$

$$\text{minBwRtoL}[i] = \min_{i \le j < T}\{\text{availbw}[j]\}\ . \qquad \textbf{(4-4)}$$

The pseudocode of the *getMinBwLeftToRight* and *getMinBwRightToLeft* functions is shown in Pseudocode 4-16.

```
find_ETSA_v2(s1 , s2 , D, B):
if ((s1 = 0) and (s2 = T-1) and (dirtyFlag.isSet())) then {
  computeSibw()
  dirtyFlag.clear()
}
return find_ETSA_v1(s1 , s2 , D , B)
reserve_ETSA_v2(s1 , s2 , B):
if ((s1 = 0) and (s2 = T-1)) then globalbw = globalbw – B  else
  reserve_TSA(s1 , s2 , B)
  dirtyFlag.set()
```

Pseudocode 4-15. Functions for the Enhanced Time Slot Array - Version 2.

```
getMinBwLeftToRight(p):
if (p<0) then return +∞
else return minBwLtoR[min{p,T-1}]+globalbw
getMinBwRigtToLeft(p):
if (p≥T) then return +∞
else return minBwRtoL[max{p,0}]+globalbw
```

Pseudocode 4-16. The *getMinBwLeftToRight* and *getMinBwRightToLeft* Functions.

4.3.5.2. The *Time Slot Groups* Data Structure

We divide the *T* time slots into *ng=O(T/k)* groups, containing *k* consecutive slots each (if the last group contains less than *k* slots, we increase *T* such that the last group also contains *k* time slots). Each group of time slots is an enhanced time slot array. The groups are stored in an array *tsg* and are numbered from *0* to *ng-1*. Group *i* contains the slots numbered from *(i·k)* to *((i+1)·k-1)*. Within the group, the slots are numbered from *0* to *k-1*. The execution of any function of any group takes at most *O(k)* time. Considering this division into groups, the time slot interval $[s_1,s_2]$ of the *reserve* function can have one of the two types of structures:

- **Type A:** s_1 and s_2 lie inside the same group *G*.
- **Type B:** s_1 is located inside some group G_1 and s_2 is located inside a group $G_2 > G_1$.

First, we will compute the group numbers (G_1 and G_2) and sr_1 and sr_2, the values of the time slots s_1 and s_2, relative to their groups:

$$G_i = \lfloor s_i / k \rfloor\ ,\ sr_i = s_i \bmod k\ (i = 1,2)\ . \qquad \textbf{(4-5)}$$

In the case of a *Type A* structure, we will simply call the *reserve* function of the group *G* with parameters (sr_1, sr_2, B), which will be executed in *O(k)* time. For a *Type B* structure, we will update the interval of time slots $[sr_1, k-1]$ in G_1, the interval $[0, sr_2]$ in G_2 and the intervals *[0, k-1]* for each group between G_1 and G_2. Updating G_1 and G_2 takes *O(k)* time, while updating the other *O(T/k)* groups takes *O(1)* time for each group. The overall complexity is *O(k+T/k)*.

```
reserve_TSG(s1 , s2 , B):
compute G1 , G2 , sr1 , sr2
if (G1=G2) then tsg[G1].reserve_ETSA_v(1/2)(sr1,sr2,B)
else {
  tsg[G1].reserve_ETSA_v(1/2)(sr1 , k-1 , B)
  tsg[G2].reserve_ETSA_v(1/2)(0 , sr2 , B)
  for G=G1+1 to G2-1 do tsg[G].reserve_ETSA_v(1/2)(0, k-1 , B)
}
```

Pseudocode 4-17. The *reserve* Function for the *Time Slot Groups* Data Structure.

For the *find* function (Pseudocode 4-18), the time slot interval $[s_1,s_2]$ can also have one of the two types of structures presented previously. We will also compute the group numbers of s_1 and s_2 (G_1 and G_2) and the values sr_1 and sr_2. If $G_1=G_2$, we just call the *find* function of G_1, with (sr_1, sr_2, D, B) as parameters. This call takes $O(k)$ time. If $G_1<G_2$, then we will first deal with a particular sub-case.

If D is at most k, then the desired slot interval could be completely located within one of the groups $G_1+1,\ldots,G_2-1$. For each such group, we call the *find* function, with parameters $(0, k-1, D, B)$. Each call takes $O(1)$ time. The time complexity of this particular case is $O(T/k)$, as there are $O(T/k)$ groups between G_1 and G_2. The interval could also lie in G_1 (between sr_1 and $k-1$) or G_2 (between 0 and sr_2), if D is at most $k-sr_1$, or at most sr_2+1, respectively. Calling the appropriate functions for G_1 and G_2 takes $O(k)$ time. The case where $D>1$ and the desired slot interval might cross several groups is presented next.

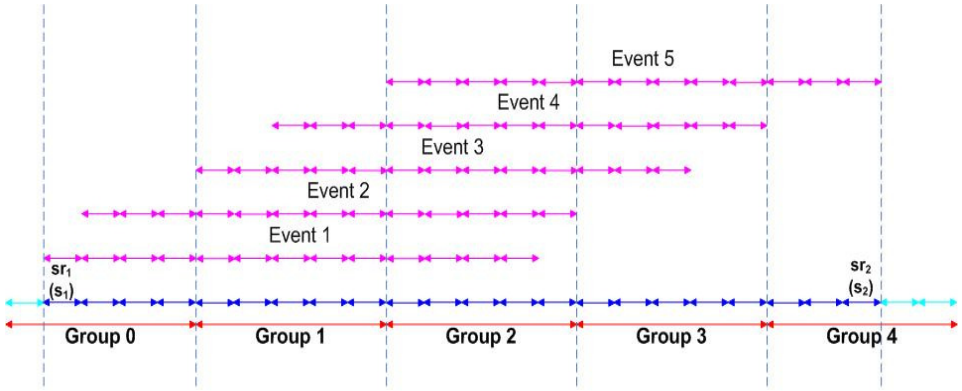


Fig. 4-1. All the Events for: T=25, k=5, s1=1, s2=22, D=13.

Before going any further, we will introduce several concepts. A candidate interval is an interval of D time slots fully included inside $[s_1,s_2]$. There are $s_2-s_1-D+2=O(T)$ candidate intervals, one for each possible starting time slot. We will traverse the $[s_1,s_2]$ interval from left to right with an interval consisting of D time slots, named the *event interval*. However, the first and last slots of this interval take only a special subset of values, each corresponding to an event. The set of all events is $\{[s_{first}, s_{last}] \mid [s_{first}, s_{last}] \subseteq [s_1, s_2]$ ***and*** $(s_{first}=s_1$ ***or*** $s_{first}=$*the first time slot of some group* ***or*** $s_{last}=s_2$ ***or*** $s_{last}=$*the last time slot of some group)* ***and*** $(s_2=s_1+D-1)\}$. The events can be sorted from left to right, according to their leftmost time slot (see Fig. 4-1).

The time slot interval $[s_1,s_2]$ contains at most $(T/k)-2$ groups between G_1 and G_2. Thus, the number of events is $O(T/k)$. For each event E, we will find the candidate interval having the maximum bandwidth, with the first time slot located between the first slot of E (inclusive) and the first slot of the next event (exclusive). If E is the last event, the next event is considered one slot to the right. Let's assume that for the current event E, the first time slot is in the group G_{begin} and its relative slot number in that group is s_{begin}. Similarly, the ending time slot's group is G_{end} and its relative time slot number in that group is s_{end}. If E is not the last event (i.e. the ending slot of E is not s_2), then we will also assume that s_{end} is not the last time slot in the group G_{end} (if it is, we consider $s_{end}=-1$ in the group $G_{end}+1$, i.e. right before the first time slot of $G_{end}+1$; after this, we

also set $G_{end}=G_{end}+1$).

```
find_TSG(s1 , s2 , D , B):
compute G1 , G2 , sr1 , sr2
if (G1=G2) then return tsg[G1].find_ETSA_v(1/2)(sr1,sr2,D,B)
else {
 ts=Bmax=-∞
 if (D≤k-sr1) then {
  (intv, maxbw)=tsg[G1].find_ETSA_v(1/2)(sr1, k-1, D, B)
  if (maxbw≥Bmax) then {
   Bmax=maxbw
   if (maxbw≥B) then ts=intv.tst }}
 if (D≤sr2+1) then {
  (intv, maxbw)=tsg[G2].find_ETSA_v(1/2)(0, sr2, D, B)
  if (maxbw≥Bmax) then {
   Bmax=maxbw
   if (maxbw≥B) then ts=intv.tst }}
 if (D≤k) then {
  for gr=G1+1 to G2-1 do {
   (intv,maxbw)=tsg[gr].find_ETSA_v(1/2)(0, k-1, D, B)
   if (maxbw≥Bmax) then {
    Bmax=maxbw
    if (maxbw≥B) then ts=intv.tst }}}
 fts=s1; lts=s1+D-1
 compute Gbegin , sbegin , Gend , send for the first event
 intGDQ=empty
 add to intGDQ the groups G in [Gbegin+1, Gend-1] in increasing order (by calling
intGDQ.addLast(G, tsg[G].getMinBw()))
 while (the last event has not been processed) do {
  if (send=k-1) then { // move to Gend+1
   if (Gbegin<Gend) then {
    remove all the elements from the end of intGDQ having a bandwidth larger than the
minimum bandwidth of the group Gend
     intGDQ.addLast((grp=Gend, bw=tsg[Gend].getMinBw())
   }
   send=-1 ; Gend=Gend+1 }
  dist=the distance between the current event and the next event
  compute popt // such that min{Bbegin(popt),Bend(popt)} is maximum
  if (not intGDQ.isEmpty()) then Bgroups=intGDQ.getFirst().bw
  else Bgroups=+∞
  if ((min{Bbegin(popt), Bend(popt), Bgroups}≥Bmax) and (Gbegin<Gend)) then {
   Bmax=min{Bbegin(popt), Bend(popt), Bgroups}
   if (Bmax≥B) then ts=fts+popt }
  move to the next event (update Gbegin, Gend, sbegin, send, fts, lts)
  if ((not intGDQ.isEmpty()) and (intGDQ.getFirst().grp=Gbegin)) then
   intGDQ.removeFirst()
 }
 if (Bmax≥B) then return (intv=[ts,ts+D-1], maxbw=Bmax)
else return [intv="no interval found", maxbw=Bmax] }
```

Pseudocode 4-18. The *find* Function for the *Time Slot Groups* Data Structure.

If $G_{begin}<G_{end}$, we compute in $O(1)$ time the distance $dist$ (in terms of time slots) between E and the next event (if E is the last event, then $dist=1$). If we slide the event interval by any number of time slots between 0 and $dist-1$, the values of G_{begin} and G_{end} will remain the same (although s_{begin}

and s_{end} would increase). Let's call the groups from $G_{begin}+1$ to $G_{end}-1$ *interior groups* and let's assume that we already know their minimum bandwidth B_{groups} (the minimum of the values returned by the *getMinBw* function of each group). We will define *cand(p)*=the candidate interval obtained by sliding the current event interval *p* positions to the right. For $0 \le p \le dist-1$, *cand(p)* contains all the interior groups of the event interval. Only the intersections with the groups G_{begin} and G_{end} change.

We define $B_{begin}(p)$=the minimum bandwidth among the time slots located in the intersection of *cand(p)* and G_{begin}. $B_{begin}(p)$ is computed as $getMinBwRightToLeft(s_{begin}+p)$ (in $O(1)$ time), called for the group G_{begin}. Analogously, we define $B_{end}(p)$=the minimum bandwidth among the time slots located in the intersection of *cand(p)* with the group G_{end}. $B_{end}(p) = tsg[G_{end}].getMinBwLeftToRight(s_{end} + p)$. Since $getMinBwRightToLeft(x) \le getMinBwRightToLeft(x+1)$ for any *x*, we have $B_{begin}(0) \le B_{begin}(1) \le \ldots \le B_{begin}(dist-1)$. Similarly, we have $B_{end}(0) \ge B_{end}(1) \ge \ldots \ge B_{end}(dist-1)$. We want to find the value of *p* ($0 \le p \le dist-1$) for which $min\{B_{begin}(p), B_{end}(p)\}$ is maximum. We distinguish between *3* cases:

- **Case 1:** $B_{begin}(0) \ge B_{end}(0)$. The optimal value for *p* is *0*, because for every *p*, $min\{B_{begin}(p), B_{end}(p)\}=B_{end}(p)$ and $B_{end}(0)$ has the highest value.
- **Case 2:** $B_{begin}(dist-1) \le B_{end}(dist-1)$. The optimal value for *p* is *dist-1*, because $min\{B_{begin}(p), B_{end}(p)\}=B_{begin}(p)$ and $B_{begin}(dist-1)$ has the highest value.
- **Case 3**: $B_{begin}(p) \le B_{end}(p)$ for $0 \le p \le pw$ and $B_{begin}(p) > B_{end}(p)$ for $pw < p \le dist-1$. The value of *pw* can be found using a simple binary search. Since $dist \le k$, the binary search takes $O(log(k))$ time. Then, the value $max\{ \min_{0 \le p \le dist-1} \{ B_{begin}(p), B_{end}(p)\}\}$ is found either for $p=pw$ or for $p=pw+1$.

Once the optimal value for *p* is found (p_{opt}), we compute the bandwidth of the candidate interval determined, which is $min\{B_{groups}, B_{begin}(p_{opt}), B_{end}(p_{opt})\}$ and compare it to the required bandwidth. In order to efficiently find the value B_{groups} while moving from the current event to the next, we will use again a sorted deque (*intGDQ*), as presented in Section III. It is easy to see that, when moving to the next event, the former rightmost group G_{end} may become an interior group and the leftmost interior group may fall outside of the interval of interior groups. Thus, B_{groups} can be computed in $O(1)$ time for each event. The time complexity of the *find* function is dominated by the computation of the best candidate intervals and is of the order $O(T/k \cdot log(k)+k)$. For the case of simple requests (i.e. $D=1$ or $D=s_2-s_1+1$), the complexity of the query operation is only $O(k+T/k)$.

We notice that all the *find* functions (of the time slot array, enhanced time slot array and time slot groups data structures) find the largest bandwidth of an interval of *D* time slots located between the corresponding time slot parameters (s_1 and s_2).

4.3.5.3. Experimental Results and Conclusions

Table 4-2. Running Times (in Seconds): T=262,144 ; k=512(=$T^{1/2}$).

Total # of operations	Number of time slots per query	Number of time slots per update	Time Slot Array	TSG without dirty flag	TSG with dirty flag
10,000	0	1 - 3000	0.13	1.4	0.1
10,000	0	150,000 - 262,144	8.75	1.17	0.1
10,000	10 - 2000	0	0.31	0.29	0.28
10,000	190,000 - 262,144	0	45.18	0.31	1.16
10,000	210,000 - 262,144	0	55.49	0.45	0.5
10,000	10 - 262,144	20,000 - 262,144	18.74	0.62	0.8
Sum of running times :			128.6	4.24	2.94

We implemented the data structure in the Java programming language and tested its performance against the standard time slot array (see Table 4-2). For all the updates, *B* was uniformly distributed between $-B_{max}$ and $+B_{max}$ (the initial value of the available bandwidth of each time slot). For all the queries, *D* was at most *75%* of the length of $[s_1,s_2]$ and *B* was between *0* and B_{max}. The tests were run one after another in the same session. We conclude that the experimental results confirmed the theoretical expectations: the running times are of the order $O(k+(T/k))$ for updates and $O(k+(T/k) \cdot log(k))$ for queries, where *k* can be chosen according to the expected ratio between update and query operations. Moreover, the data structure behaves significantly better than

the standard time slot array.

4.3.5.4. Possible Augmentations of the Time Slot Groups Data Structure

The Time Slot Groups (and, of course, the time slot array and enhanced time slot array data structures) data structure can be augmented with the following function:

- ***find'(s_1,s_2,B)*** - finds a time slot interval *[s,s+D-1]*, where at least a given amount of bandwidth *B* is available during every time slot of the interval, subject to the following QoS constraints: the length of the interval is the maximum possible value of *D* time slots, the earliest possible starting time slot is s_1 and the latest possible finish time slot is s_2 (i.e. $s_1 \leq s \leq s+D-1 \leq s_2$)

We can implement the *find'* function by binary searching the maximum number of time slots *D* of the interval we are looking for ($1 \leq D \leq s_2-s_1+1$). Then, we will call *find(s_1,s_2,D',B)*, where *D'* is the current value chosen by the binary search (we call *find_TSA* in the case of the time slot array, *find_ETSA_v(1/2)* in the case of the enhanced time slot array and *find_TSG* in the case of the time slot groups data structure). If the maximum bandwidth of an interval with *D'* time slots is $\geq B$ then we search for larger values in the binary search; otherwise, we will search for smaller values. The largest value *D'* found within the binary search for which we found an interval of *D'* time slots whose bandwidth is at least *B* is the length *D* of the interval we are looking for (the interval itself is also returned by the *find* function of the *3* data structures we mentioned).

4.4. Online Scheduling of Fixed-Bandwidth Fixed-Duration Data Transfer Requests on a Network Path

We now consider the particular situation when the network is a path composed of *n* nodes v_1, v_2, …, v_n (and there is a link between every two consecutive vertices in this order). A data transfer request contains, besides the parameters considered in the previous section, the identifiers of the source and the destination nodes. We could maintain a separate data structure for each network link and, for a data transfer request between two nodes *i* and *j*, we could query (and then update) the data structures of all the *O(n)* links in-between *i* and *j*. In many situations, this approach is good enough. However, we can do better than this, by using multidimensional data structures (for instance, two-dimensional structures, where the first dimension corresponds to the network links and the second dimension corresponds to the time slots).

4.4.1. The d-dimensional Segment Tree

An extended d-dimensional segment tree ($d \geq 2$) is composed of a segment tree for the first dimension in which every node contains two (d-1)-dimensional segment trees $T_{covering}$ and T_{total} (instead of the *uagg* and *qagg*, which are stored only in the 1-dimensional segment trees). Considering that each dimension has *O(m)* elements, then the number of nodes in a d-dimensional segment tree is $O(m^d)$. Each tree node will maintain an attribute *dim*, representing the corresponding dimension (the nodes with *dim=1* contain actual values; the other nodes only contain multidimensional data structures). Using the proposed algorithmic framework, we only need to redefine several functions (and add an extra parameter *dr* to *STrangeUpdate* and *STrangeQuery*). A range query/update consists of *d* intervals (one for each dimension): $dr=[l_1,h_1] \times [l_2,h_2] \times \ldots \times [l_d,h_d]$.

A range update modifies the values of *O(log(m))* intervals (tree nodes) in dimension *d*. For each tree node *p* in dimension *d*, the update function is called on the $p.T_{total}$ (and/or $p.T_{covering}$) *(d-1)*-dimensional extended segment trees. Thus, $O(\log^d(m))$ tree nodes are visited. A range query aggregates the values of *O(log(m))* covering nodes of each tree in dimension *d*. For each covering node, the query function is called on the $p.T_{total}$ (and/or $p.T_{covering}$) *(d-1)*-dimensional extended segment tree, thus taking $O(\log^d(m))$ time overall.

We can easily support any combination of range updates and range queries, as long as the range of every query (or every update) consists of only one cell (point query/update). However, supporting both range queries and range updates with an extended segment tree seems to be

possible only for a few types of query and update functions (e.g. range addition updates and range sum queries, with $mop(u,a,b)=(b-a+1)\cdot u$). We were unable to extend the 1D update aggregates pushing technique to multiple dimensions (as this would require pushing and merging multidimensional structures, instead of 0-dimensional structures, i.e. numerical values or tuples with a constant number of fields).

```
STrangeUpdateNodeFit(node, u, dr):
if (node.dim>1) then {
  STrangeUpdate(node.T_covering, u, l_node.dim-1, h_node.dim-1, dr)
  STrangeUpdate(node.T_total, mop(u, node.left, node,right), l_node.dim-1, h_node.dim-1, dr)
} else use the 1D STrangeUpdateNodeFit function
STrangeUpdateNodeIncl(node, u, a, b, dr):
if (node.dim>1) then STrangeUpdate(node.T_total, mop(u,a,b), l_node.dim-1, h_node.dim-1, dr)
else use the 1D STrangeUpdateNodeIncl function
STrangeQueryNodeIncl(node, a, b, dr):
if (node.dim>1) then return mop(STrangeQuery(node.T_covering, l_node.dim-1, h_node.dim-1), a, b, dr)
else use the 1D STrangeQueryNodeIncl function
STrangeQueryNodeFit(node, dr):
if (node.dim>1) then return STrangeQuery(node.T_total, l_node.dim-1, h_node.dim-1, dr)
else use the 1D STrangeQueryNodeFit function
```

Pseudocode 4-19. Functions for the d-dimensional Segment Tree Algorithmic Framework.

```
BPrangeUpdateFullBlock(blk, u, bpart, dr):
if (bpart.dim>1) then {
  BPrangeUpdate(u,l_bpart.dim-1,h_bpart.dim-1,bpart.Covp[blk], dr)
  BPrangeUpdate(mop(u, bpart.left[blk], bpart.right[blk]),
                l_bpart.dim-1, h_bpart.dim-1, bpart.Totalp[blk], dr) }
} else { bpart.uagg[blk]=uFunc(u, bpart.uagg[blk])
    bpart.qagg[blk]=uFunc(mop(u, bpart.left[blk], bpart.right[blk]), bpart.qagg[blk]) }
BPrangeUpdatePartialBlock(blk, u, a, b, bpart, dr):
if (bpart.dim>1) then BPrangeUpdate(mop(u, a, b),
                l_bpart.dim-1, h_bpart.dim-1, bpart.Totalp[blk], dr)
else bpart.qagg[blk]=uFunc(mop(u, a, b), bpart.qagg[blk])
for i=a to b do
  if (bpart.dim>1) then BPrangeUpdate(u, l_bpart.dim-1, h_bpart.dim-1, bpart.dp[i], dr)
  else bpart.v_i=uFunc(u, bpart.v_i)
BPrangeQueryFullBlock(blk, bpart, dr):
if (bpart.dim>1) then return BPrangeQuery(l_bpart.dim-1,
                h_bpart.dim-1, bpart.Totalp[blk], dr)
else return bpart.qagg[blk]
BPrangeQueryPartialBlock(blk, a, b, bpart, dr):
if (bpart.dim>1) then qres=mop(BPrangeQuery(l_bpart.dim-1,
                h_bpart.dim-1, bpart.Covp[blk], dr), a, b)
else qres=mop(bpart.uagg[blk], a, b)
for i=a to b do
  if (bpart.dim>1) then qres=qFunc(qres, BPrangeQuery(l_bpart.dim-1, h_bpart.dim-1, bpart.dp[i], dr))
  else qres=qFunc(qres, bpart.v_i)
return qres
```

Pseudocode 4-20. Functions for the d-dimensional Block Partitioning Algorithmic Framework.

4.4.2. The d-dimensional Block Partition

The block partitioning technique is extended by splitting each dimension into m/k k-sized blocks. In one dimension, the block division requires $(k+1)\cdot m/k$ memory locations. In d dimensions, the memory size increases to $((k+1)\cdot m/k)^d$. Extending range queries with point updates (and range updates with point queries) to multiple dimensions is rather easy. We only need to redefine some of the framework's functions. Range queries and updates can be supported when using the block division in time $O(3^d \cdot m^{d/2})$, when choosing $k=m^{1/2}$ (a range query/update has time complexity $O(3 \cdot m^{1/2})$ in one dimension).

Each d-dimensional partition into blocks consists of an array *dp*, where *dp[i]* is a *(d-1)*-dimensional block partition, corresponding to the i^{th} cell of the d^{th} dimension, and two arrays, *Totalp* and *Covp*, where each entry of the arrays is a *(d-1)*-dimensional block partition, corresponding to a block in the d^{th} dimension. As before, each block partition has a field *dim*, corresponding to its dimension (*dim=1* corresponds to a normal, one-dimensional block partition). Just like in the case of the multidimensional segment tree, a range query/update consists of d intervals (one for each dimension): $dr=[l_1,h_1] \times [l_2,h_2] \times \ldots \times [l_d,h_d]$.

We assume the existence of two functions: *BPrangeUpdate(u, a, b, bpart, dr)* and *BPrangeQuery(a, b, bpart, dr)*, similar to those defined in subsection 4.3.3. *bpart* is a block partition (with dimension *bpart.dim*) and *[a,b]* is the range in the dimension *bpart.dim*. The necessary functions are presented in Pseudocode 4-20.

4.5. Practical Scenarios for Scheduling Data Transfer Requests on Network Links and Paths

In this section we discuss the practical applicability of the data structures presented in the previous sections. We will present several scenarios, in which using their capabilities will be beneficial.

In the first scenario, the data transfer scheduler receives requests which ask for a fixed amount of bandwidth B during a fixed time interval $[t_1,t_2]$. By dividing the time horizon into equally sized time slots, this problem has two components:

- find the minimum available bandwidth among all the time slots in the interval $[s_1,s_2]$, where s_1 and s_2 are the time slots in which t_1 and t_2 are located.
- decrease the available bandwidth with the same value B for each time slot in $[s_1,s_2]$.

These two operations are equivalent to a range addition update and a range minimum query and can be efficiently handled by the proposed data structures.

In the second scenario, clients need to send multimedia data at a minimum rate R in a wireless network. The scheduler manages a set of N frequencies, numbered consecutively from *1* to N. For each frequency i, the scheduler keeps track of the maximum rate r_i at which data can be sent on that frequency. We can consider the values $r_i<R$ as negative and the others positive (by setting $r_i=r_i-R$). A request asks for an allocation of consecutive frequency numbers completely located inside an interval $[f_{min}, f_{max}]$, such that the sum of the transfer rates on the frequency interval is maximum. Occasionally, the values r_i may need to be updated. This problem is the range maximum segment sum that we presented.

In the third scenario, the scheduler manages two resources: frequency numbers and time. For each pair *(i,j)*, the available bandwidth $B_{i,j}$ for frequency i during time slot j is known. Clients need to send as much data as possible during a fixed time slot interval $[s_1,s_2]$ and using a fixed interval of frequency numbers $[f_1,f_2]$. The total amount of data that can be sent is $\sum_{i=s1}^{s2}\sum_{j=f1}^{f2} B_{i,j} \cdot slot\_duration$, where *slot_duration* is the duration of a time slot. A client may want to query several time and frequency intervals until it finds enough available bandwidth. Occasionally, the values $B_{i,j}$ need to be updated. This problem can be handled efficiently by a multidimensional data structure.

4.6. Scheduling Non-Preemptive Data Transfer Requests in Tree Networks

In this section we consider the problem of scheduling data transfers in real-time in a tree network. We present algorithmic techniques for this problem, some of which can be implemented in a centralized data transfer scheduling framework which has full control over the network. We consider only non-preemptive data transfers in this section.

Trees are some of the simplest non-trivial topologies which appear in real-life situations. Many of the existing networks have a hierarchical structure (a tree or tree-like graph), with user devices at the edge of the network and router backbones at its core. Furthermore, many graph topologies can be reduced to tree topologies, by choosing a spanning tree or by covering the graph's edges with edge disjoint spanning trees [Roskind and Tarjan, 1985]. In a tree network, there exists a unique path between every two nodes. Thus, the scheduling techniques do not need to compute an optimal path (as there is only one). In the case of multicast transfers (e.g. multimedia live streaming to many clients), the multicast tree is a subtree of the tree network and is uniquely defined by the source node and the destination nodes. Under these conditions, only time scheduling techniques are employed.

4.6.1. The Real-Time Data Transfer Scheduling Model

A (centralized) scheduler has full control over the communication network (i.e. the background traffic is non-existent or very low compared to the bandwidth of the links). The communication topology is a tree with n nodes; each node can issue data transfer requests to the scheduler, containing several parameters, like: source node, destination node(s), earliest start time, latest finish time, duration and minimum required bandwidth (in the case of non-preemptive data transfers), total size of the transferred data (in the case of preemptive data transfers), profit (obtained if the request is scheduled and all its constraints are satisfied). The scheduler handles these requests in batches of at most $R \geq 1$ requests at a time (i.e. the requests are not necessarily handled immediately – the scheduler waits until $m=R$ requests are received or until a short time limit is exceeded, if $m<R$). Once a batch of requests is constructed, the scheduler runs an optimization algorithm, considering the $m \leq R$ requests in the batch, as well as the previously scheduled data transfers. Note that the case $R=1$ means that the scheduler handles the requests one at a time and corresponds to the usual meaning of real-time processing.

When scheduling the new batch of requests, the scheduler may choose to interrupt some of the previously scheduled data transfers and resume them later or modify their parameters (e.g. allotted bandwidth), if such actions are permitted. We consider two types of scheduling behavior for the requests in the current batch. The most general type consists of choosing the starting time (as soon as possible or some time in the future) of each request in the batch or rejecting it (for good).

The second type is applicable only when the requests do not have an earliest start time parameter (i.e. they can be started any time after they are submitted). The scheduler divides the time into T equal time slots (note that time may also be divided into time slots in the case of the first type) and data transfers are started only at the beginning of a time slot. The scheduler maintains the value of the next time slot during which requests can be started. Then, the scheduler chooses a subset of requests from the current batch which will be started at the beginning of the next time slot. The other requests are either rejected or delayed until the next batch. After the batch is processed, the value of the next time slot when transfers can be started is increased. If a request is delayed for more than an upper limit of UB batches (or if it is not scheduled until its deadline is exceeded), then it will be rejected.

4.6.2. Non-Preemptive Data Transfers with Unit Durations and Full Link Usage

We will consider that the time horizon is divided into T equally-sized time slots and that every data transfer request has the same duration, equal to one time slot (unit duration). We consider first that the requests do not have an earliest start time (i.e. they can start immediately after

the request is submitted to the scheduler), but they may have a deadline (latest finish time). The case where an earliest start time is also given could be easily handled by assigning a priority to each request in the batch, sort them according to the priority and then consider the requests in this order, one at a time, as if the size of the batch was equal to one request. Moreover, every transfer requires the full bandwidth of the links on its path (subtree).

The starting times of the data transfers correspond to the beginning of the time slots and the requests in each batch are scheduled to start later than the requests in the previous batches. Because each data transfers lasts for only one time slot, the requests in each batch can be scheduled without considering the requests in the previous batches (because those data transfers will already be finished when the scheduled requests from the current batch will start). Thus, the optimization algorithm only needs to take care of "conflicts" among the requests in the same batch.

4.6.2.1. Edge- and Vertex-Disjoint Data Transfers

A request may correspond to a unicast or multicast data transfer. Since the communication topology is a tree, a unicast transfer corresponds to a unique path in the tree (from the source to the destination) and a multicast transfer corresponds to a unique subtree spanning the source node and all the destination nodes. The unicast path (multicast subtree) will be called the *subgraph* of the request. Because of the full bandwidth requirements, the scheduled requests must have edge-disjoint subgraphs. If, moreover, the requests require the full processing power of the nodes in their subgraphs, then the subgraphs of the scheduled requests must be vertex-disjoint (which also implies edge-disjointness in tree networks). If the subgraphs of two requests r_1 and r_2 are not (edge-) vertex-disjoint, we say that r_1 and r_2 are incompatible (they mutually exclude each other). Using the incompatibility relations, we can construct a mutual exclusion graph, consisting of the requests as vertices and every pair of vertices connected by an edge correspond to a pair of incompatible requests.

We will first consider the case of vertex-disjoint data transfers. It is well known that the vertex intersection graph of subtrees of a tree is a chordal graph. In our case, the mutual exclusion graph of the requests is chordal. We will use the following notations: V=the number of vertices of the mutual exclusion graph ($V=m$, the number of requests in the batch) and E=the number of edges of the mutual exclusion graph ($E=O(V^2)$).

If we are interested in maximizing the total profit of the scheduled requests (from the current batch), then we need to find a maximum weighted independent set in the chordal mutual exclusion graph, where the weight of a vertex j ($w(j)$) is the profit of the corresponding request (we will consider the graph to be connected; otherwise, we run the algorithm for each of its connected components). We will present here a new algorithm, based on dynamic programming on the clique tree of the chordal graph. Every chordal graph has a perfect elimination ordering (*PEO*), based on which the associated clique tree (tree of maximal cliques) can be computed. A clique tree has $O(V)$ vertices. Both the *PEO* and the clique tree can be computed in linear time ($O(V+E)$) [Galinier, Habib and Paul, 1995]. We will root the clique tree at an arbitrary clique C_r and then perform a bottom-up dynamic programming algorithm.

For each clique C_i, we will compute the value $W_{max}(i,j)$=the maximum weight of an independent set in the subset of graph vertices contained in C_i and all of its descendants, such that j is a vertex contained in the set (if $j=0$, we consider that no vertex of C_i is part of the set). Obviously, since C_i is a clique, at most one vertex of C_i can be part of the independent set. We can compute W_{max} in the following way:

$$W_{max}(i,0) = \sum_{C_k \text{ son of } C_i} \max\left\{ \begin{matrix} W_{max}(k,0) \\ W_{max}(k,j),\ j \in (C_k \setminus C_i) \end{matrix} \right\} \quad \textbf{(4-6)}$$

$$W_{max}(i,j>0) = w(j) + \sum_{C_k \text{ son of } C_i} \left\{ \begin{matrix} W_{max}(k,j) - w(j), \text{if } j \in C_k \\ W_{excl}(k) = \max\left\{ \begin{matrix} W_{max}(k,0) \\ W_{max}(k,p),\ p \in (C_k \setminus C_i) \end{matrix} \right\}, \text{if } j \notin C_k \end{matrix} \right. \quad \textbf{(4-7)}$$

A trivial implementation takes $O(V^3)$ time. By computing for each clique C_k the value

$W_{excl}(k) = max\{W_{max}(k,0), W_{max}(k,p)$ with p in C_k, but not in its parent $C_i\}$, we can reduce the time complexity to $O(V^2)$ ($W_{excl}(k)$ can be computed in $O(V)$ time, by considering every graph vertex in C_k and testing in $O(1)$ time whether it also belongs to C_k's parent C_i).

In the case of edge-disjoint data transfers, we were not able to find useful properties of the mutual exclusion graph. Edge intersection graphs of paths in a tree (corresponding to unicast data transfers) were studied in [Golumbic, Lipshteyn and Stern, 2005]. For this case, we suggest using some of the approximation algorithms for the maximum weight independent set presented in the literature (e.g. [Kako, Ono, Hirata and Haldorsson, 2005]), where the weights of the vertices are the profits of the corresponding transfers.

4.6.2.2. Batches of Size One (one request)

Setting the batch size to one request makes things easier, because no conflicts need to be considered. In this case, we can allow the requests to have an earliest start time, too. We would like to find for each request a time slot in which all the links in the request's subgraph are available. We could consider the data structures presented in [Andreica and Ţăpuş, 2008f] and assign such a data structure (enhanced time slot array, segment tree or block partition) to every link of the tree. Then, for each time slot in the *[earliest start time, latest finish time]* range, we would query every link in the request's subtree to check if it is available.

When all the data transfers are unicast, we can use a multi-dimensional data structure [Andreica and Ţăpuş, 2008f], together with a heavy path decomposition of the tree network, which splits the tree into paths, based on light and heavy edges. This decomposition has the property that there are $O(log(n))$ edges to be traversed between any two paths of the decomposition. We can associate a multi-dimensional segment tree (or block partition) to each path in the decomposition. Then, when a request's path spans multiple paths of the decomposition, we will query/update all of them appropriately. The request path spans $O(log(n))$ such paths, thus there is only an $O(log(n))$ performance loss over the path network case (which was considered in [Andreica and Ţăpuş, 2008f]). The update and query operations which need to be supported by the data structure are range addition update (set a range of values to *1*; the values were previously equal to *0*) and range sum query (count the number of *1* entries in a multi-dimensional range). As it turns out, this pair of operations (range addition update, range sum query) is one of the few combinations which can be supported efficiently by multi-dimensional segment trees or block partitions. With such a data structure, we can verify in $O(log(n) \cdot log(T))$ time if a time slot t is available for every link of a path.

4.6.2.3. A Model for Rescheduling Data Transfers

So far, we have not considered the possibility of rescheduling some of the data transfers. Once a data transfer was scheduled, the scheduling parameters (e.g. start time, finish time) would remain fixed for that data transfer. We propose here a model for the case when no earliest start times are given, together with an algorithmic technique which supports the model.

Let's consider all the requests in the current batch $r_1, r_2, \ldots, r_m$, sorted according to their deadlines (latest finish times), i.e. $LF_1 \leq LF_2 \leq \ldots \leq LF_m$. Since the time is divided into time slots, the latest finish times are integer numbers. We will construct a bipartite graph with all the requests on one side and all the $nt=LF_m-LF_1+1$ candidate time slots t_j on the other side ($t_1=LF_1, t_2=t_1+1, \ldots, t_i=t_{i-1}+1, t_{nt}=LF_m$). A request r_i will be connected by an edge to every time slot $t_j \leq LF_i$. This graph is a special type of bipartite graph, called *chain bipartite graph*, because the neighboring vertices of a vertex r_i are a superset of the neighbors of r_{i-1}. The restricted situation we consider is when the mutual exclusion graph of the requests r_m is a clique (i.e. a complete graph). Each vertex r_i has a weight p_i, representing the profit of the corresponding request. Each edge (r_i, t_j) has a cost $c_{i,j}=p_i$ minus the sum of the profits of the requests scheduled during time slot t_j, whose subgraphs intersect the subgraph of r_i. Since the mutual exclusion graph is a clique, at most one request r_i can be scheduled during one of the time slots t_j. Thus, a maximum (edge-)cost matching in the graph we have previously introduced represents an optimal way of scheduling some of the requests in the current batch.

```
MaxCostMatching():
HR_match={}; HT_unm={}; HT_match={}
max_cost = 0 ; tlast(0)=0
for i=1 to m do tlast(i)=LF_i-LF_1+1
for i=1 to m do {
  for j=tlast(i-1)+1 to tlast(i) do HT_unm.insert((wts=w_j, ts=j))
  if (HT_unm.size()>0) then {
    (wts, ts)=HT_unm.getMin()
    max_cost=max_cost+p_i-wts
    HT_unm.deleteMin()
    HT_match.insert((wts, ts))
    HR_match.insert((wr=p_i, r=i))
    while (HR_match.getMin().wr≤HT_match.getMax().wts) do {
      max_cost=max_cost-HR_match.getMin().wr+HT_match.getMax().wts
      HR_match.deleteMin()
      HT_match.deleteMax()
    }
  } else // HT_unm.size()=0
}
if (HR_match.getMin().wr<p_i) then {
  max_cost=max_cost+p_i-HR_match.getMin().wr
  HR_match.deleteMin()
  HR_match.insert((wr=p_i, r=i))
}
```

Pseudocode 4-21. Maximum Cost Matching Algorithm.

A maximum (edge-)cost matching in a bipartite graph can be obtained by modifying one of the well-known minimum cost maximum matching algorithms. At every iteration, a maximum cost path (from a virtual source to a virtual sink) is computed in the residual graph (which may have both positive and negative weights on its edges, but does not contain any positive cycles). There are standard algorithms for this path computation, like Bellman-Ford-Moore (which takes $O((m+nt)^3)$ time). When the maximum cost of a path is negative, we stop the algorithm.

The number of iterations of the algorithm described in the previous paragraph is $O(min\{m,nt\})$, leading to an $O((m+nt)^4)$ overall time complexity, which is too large to be used in a real-time systems. Instead, we propose a different model, which is less accurate. Each vertex t_j has an estimated weight w_j, which is computed based on the requests scheduled during the time slot t_j and the requests in the current batch (e.g. it could be equal to the sum of the profits of the requests scheduled during time slot t_j which intersect every request in the current batch, but some more interesting and relevant functions can be defined). We will now define the cost of each edge (r_i, t_j) as $c_{i,j}=p_i-w_j$. Of course, we could use the same maximum cost matching algorithm as before, but due to the particular nature of the edge-cost function and the structure of the bipartite graph, we can do better. For each request r_i, we compute $tlast(i)=j$, if $t_j=LF_i$ (we consider $tlast(0)=0$).

We will traverse the request vertices from r_1 to r_m and maintain a min-heap (priority queue) HR_{match} of the matched requests, a min-heap HT_{unm} of the unmatched time slots and a max-heap HT_{match} of the matched time slots. The maximum cost of a matching will be maintained in the variable *max_cost*. A maximum cost solution has the property that the minimum profit of a matched request is larger than the maximum weight of a matched time slot. The algorithm is sketched in Pseudocode 4-21.

The time complexity of this algorithm is $O((m+nt)\cdot log(m+nt))$, which is a great improvement upon the standard maximum cost matching algorithm. If all the requests or all the time slots have equal weights, we can use a normal queue (or a stack) instead of a priority queue (heap) and obtain an $O(m+nt)$ algorithm.

4.6.2.4. A (Simplified) Model for Matching Requests to Time Slots

We consider in this subsection another restricted model for scheduling requests which do not have an earliest start time and which have the same latest finish time (if they do not have the same latest finish time, they can be split into several groups which are processed independently, such that all the requests in a group have the same latest finish time). Furthermore, the subgraphs of all the vertices are the same (e.g., they could be data transfers from the same source vertex to the same destination). Every request r_i has a minimum bandwidth requirement B_i. However, two requests cannot share the same link at the same time (because every data transfer is capable of using up all of the extra bandwidth available along the path, leading to bandwidth conflicts). Every request r_i has an associated profit function, which is similar for all the requests: if the available bandwidth along the path is larger than B_i, then the obtained profit is p_1; if the available bandwidth along the path is equal to B_i, the obtained profit is p_2 ($p_2 \leq p_1$). If the request is not scheduled, we need to pay a sum equal to p_3 ($-p_3 \leq p_2$). p_1 and p_2 are non-negative numbers (obviously).

In this case, we can model the situation as a bipartite graph containing all the requests on the left side and all the candidate time slots (from the current time slot to the latest finish time) on the right side. Each request r_i has an associated required bandwidth B_i and each time slot t_j has an associate available bandwidth AB_j. At first, we will make the number of requests and the number of time slots equal. If there are more time slots than requests, we can drop the time slots with the lowest amounts of available bandwidth. If we have more requests than time slots, we can drop the requests with the largest amounts of required bandwidth (and consider them rejected).

Thus, we will assume that we have m requests and m time slots. We have an edge for each pair (r_i, t_j) and its cost $c(i,j)$ is equal to the (positive) profit or (negative) penalty obtained if request r_i is scheduled during time slot t_j: p_1, if $AB_j > B_i$; p_2, if $AB_j = B_i$; $-p_3$, if $AB_j < B_i$ (if $AB_j < B_i$, then request r_i is, in fact, rejected). We want to find a maximum-cost matching in this bipartite graph. Of course, we can use a standard maximum-cost matching algorithm (as discussed in the previous subsection), but, due to the special nature of the edge costs, we can do better. We will now present a dynamic programming algorithm with time complexity $O(m^2)$.

We will consider the requests sorted, such that $B_1 \geq B_2 \geq \ldots \geq B_m$; similarly, the time slots are sorted such that $AB_1 \leq AB_2 \leq \ldots \leq AB_m$. We will compute a table $C_{max}(i,j)$=the maximum profit obtained if we considered the requests $r_1, r_2, \ldots, r_{i-1}$ and the requests $r_{j+1}, r_{j+2}, \ldots, r_m$. The strategy is to consider the time slots in order and, for each time slot, decide which request will be matched to that time slot. We have $C_{max}(1,m)=0$. When computing $C_{max}(i,j)$, we have already considered the first $(i-1)+(m-j)$ time slots and, thus, we are interested in time slot t_k, where $k=i+m-j$. We can match the requests r_i or r_j to time slot t_k. We have

$$C_{max}(i,j) = max\{c(i,k)+C_{max}(i+1,j),\ c(j,k)+C_{max}(i,j-1)\}.$$

By computing the values $C_{max}(i,j)$ in increasing order of $(j-i+1)$, we obtain an $O(m^2)$ time complexity. The pseudocode is described below:

```
DPMatching():
for q=1 to m do { // q=1,2,…,m
  for i=1 to m-q+1 do { // i=1,2,…,m-q+1
    compute Cmax(i, i+q-1)
  }
}
```

Pseudocode 4-22. A Simplified Request Matching Algorithm.

There exists another solution, based on a greedy strategy, having the same time complexity. We consider the requests and time slots ordered as before. Then, we consider every circular cp permutation of the requests $r_{cp(1)}, r_{cp(2)}, \ldots, r_{cp(m)}$ and, for each such permutation, we match each request $r_{cp(i)}$ to time slot t_i and compute the total profit. The maximum profit is the best profit obtained for one of the circular permutations.

4.6.3. Non-Preemptive Data Transfers with Multi-Unit Durations and Full Link Usage

We will handle the case of non-preemptive data transfers with multi-unit durations and full link usage by reducing it to the unit-duration case. We will consider first the situation when there are no earliest start times given (i.e. every data transfer can be started at any time after the request is submitted). The supplementary problem that arises when scheduling a batch is that some other data transfers may already take place. If we are not allowed to cancel data transfers and restart them later, then we will remove all the requests from the current batch which are in conflict with some data transfers which are already taking place and then schedule them using some of the techniques presented in the previous section.

However, if we are allowed to cancel and restart (not resume) the data transfers, we could choose to add some of the previously started data transfers to the batch (and remove those requests from the batch which are in conflict with previously started data transfers not added to the batch). With this extended batch, we will again use one of the scheduling techniques from the previous section. If a data transfer which was already taking place is scheduled, then it will continue to run normally; otherwise, it will be cancelled and reconsidered later (when it will have to be restarted).

When a request has an earliest start time *ES* and the time interval between *ES* and the latest finish time *LF* is equal to the transfer's duration (i.e. the transfer can be scheduled only during a fixed time interval), we consider only unit size batches and all the data transfers are unicast, then we can use the same multi-dimensional data structures as in subsection 4.4, obtaining a squared logarithmic time for checking if the request can be granted (if the data transfers can also be multicasts, then we can use a 1D data structure for every network link, but the time complexity increases).

4.6.4. Non-Preemptive Data Transfers with Unit Durations and Partial Link Usage

When each data transfer requires a minimum bandwidth (part of a link), we obtain an even more difficult problem. In the restricted case of a network composed of two nodes and a single link, this problem is equivalent to the well-known knapsack problem (if we ignore the requests' deadlines in the decision process). Thus, efficient algorithms are difficult to find in the general case. For the case when earliest start times are not given, we consider the following simple approach: we will assign to each data transfer in the batch a priority, then sort the data transfers according to their priorities and add them to the network in this order; if a data transfer does not have enough bandwidth on (at least) one of the links of its path (or subtree, for multicast requests), then it will not be scheduled immediately (it will be delayed or rejected). The priority function can be customized and can take into consideration such parameters like the profit of the request, the minimum required bandwidth, the number of links of the transfer's path (subtree), available bandwidth on those links and so on.

Another approach which is feasible when each transfer requests a large fraction of the links of its subgraph and the requests' profits are equal is to compute a mutual exclusion hypergraph and find a large independent set in it. The vertices of the hypergraph are the requests in the batch and an edge is a subset of requests which cannot all be scheduled at the same time (it is desirable that every edge is a small subset). Then, we can use the following greedy heuristic for finding a maximum independent set in a hypergraph: We will maintain a max-heap *H* containing all the vertices of the hypergraph, together with their degree (the number of edges they belong to). Then, we repeatedly remove the vertex *i* with the maximum degree and decrease correspondingly the degrees of all of its neighbors *j*. The time complexity of this algorithm is $O((m+E)\cdot log(m))$, due to the heap operations (*E*=the number of edges in the hypergraph). The pseudocode is shown in Pseudocode 4-23.

The non-removed vertices form an independent set. This heuristic can be used even when the request have different profits. In this case, each vertex will be assigned a weight, based on its profit, degree and possibly other factors. We then remove the vertices from the hypergraph in decreasing order of their weights. When earliest start (*ES*) and latest finish (*LF*) times are given and we

consider only unit size batches, we can maintain a time slot array for every link and easily verify for every time slot t in the range *[ES, LF]* if every link of the request's subgraph has enough available bandwidth during time slot t.

```
HyperGraph-VertexRemoval():
H=empty
for each vertex i do {
  deg(i)=0
  removed(i)=false
 for each edge e, such that i∈ e, do deg(i)=deg(i)+1
 if (deg(i)>0) then H.insert((value=deg(i), vertex=i))
}
while (H.size()>0) do {
 (value=deg(i), vertex=i)=H.extractMin(); removed(i)=true
 for each edge e, such that i∈ e, do {
  for each vertex j≠i, s.t. ((j∈ e) and (not removed(j))), do {
    H.remove((deg(j), j))
    deg(j)=deg(j)-1
    if (deg(j)>0) then H.insert((value=deg(j), vertex=j))
  }
 }
}
```

Pseudocode 4-23. The Hyper-Graph Repeated Vertex Removal Algorithm.

4.6.5. Non-Preemptive Data Transfers with Multi-Unit Durations and Partial Link Usage

We want to handle the case of non-preemptive data transfers by reducing it to the unit duration case. However, when earliest start times are not given, we will need to consider more sophisticated priority functions. For instance, these functions will need to also consider the duration of the data transfer (some simple examples could be: *profit/(required bandwidth* × *duration)* or *profit/duration*). If the requests have a fixed time interval during which they can be executed (i.e. fixed starting and finish times are given) and we use unit size batches, then we can assign a segment tree (or block partition) data structure [Andreica and Ţăpuş, 2008f] to every link to enhance the speed of the procedure which verifies if a request can be granted or not.

4.7. Real-Time Scheduling of Fixed-Data Fixed-Duration (Deadline-Constrained) Out-of-Order Data Transfer Requests

In this section we will consider fixed-data fixed-duration requests (also called deadline-constrained) requests. The parameters of a request are: t_1 (the earliest time moment at which the data transfer may start; the request may be submitted at the time moment t_1 or earlier), t_2 (deadline), s (the identifier of the source node), d (the identifier of the destination node), TD (total data to be transferred), *dataid* (identifier of the data on the source node - e.g. file name, file location, and offset within the file). The request may optionally be assigned a weight w, representing its priority (e.g. it could be proportional to the revenue of the data transfer service provider obtained if the request is satisfied, and/or proportional to TD).

In the case of a *reliable* data transfer, a request is satisfied if all of its data arrives at the destination by the time moment t_2. If there is even one piece of data which has not reached the destination by the time moment t_2, then the request is not satisfied (and all the network resources consumed for transferring pieces of data for this request can be considered wasted). In the case of *unreliable* data transfers, the intent is that most of the data to be transferred should reach the destination by its deadline; thus, a request can be partially satisfied (depending on the amount of

data which reached the destination in time).

We consider two methods of handling the requests. The first one assumes the existence of a central scheduler which is aware of the entire network topology (and its parameters). The scheduler decides how to split the data into packets and how to route these packets through the network, such that every packet arrives at the destination by the specified deadline. Of course, the scheduler could decide that the request cannot be satisfied (e.g. in the case of reliable data transfers) and, thus, could reject the request.

The second method considered is a fully decentralized one. The purpose of also considering decentralized approaches here is that of providing a comparison reference for the centralized approach. A request is passed by the user application to the source node of the request. The source node splits the data into packets (however it sees fit) and then forwards the packets to its neighbors. Whenever a neighbor receives a packet, it checks if it is the packet's final destination. If the packet must be forwarded further, the node decides whether to split the packet further and, for each newly generated packet (possibly only one packet - the same packet that was received), it decides the neighbor to which the packet should be sent further (or delays the decision until a later time). Deadline information is encoded within each packet. Thus, the decisions made by each node regarding the routing of a packet must consider this information.

Both approaches handle the requests *online* (i.e. one at a time, as soon as they are submitted; if multiple requests are submitted at the same time, they are handled sequentially in a first come-first served order). For simplicity, we will consider that the data to be transferred is already split into a given number of (fixed size) packets, which could not be split further; thus, size differences between the data of two different requests are expressed only as possibly different numbers of packets. As before, we consider that time is divided into equally-sized time slots.

The bandwidth of a (directed) connection from a vertex i to a vertex j is a function $B(i,j,t) \geq 1$ and has the meaning that $B(i,j,t)$ packets can be transferred from the time moment t (beginning of the time slot t) to the time moment $t+1$ (beginning of the time slot $t+1$), from vertex i to vertex j along the connection. $B(i,j,t)$ is a natural number. We consider that the size of a packet is sufficiently small such that a packet never needs more than 1 time unit for being transferred along a connection (excluding queuing delays, of course). Note that we implicitly assumed that the latency of each connection is 1 time unit. A more general model assumes that a packet sent along the connection $i->j$ at time moment t takes $lat(i,j,t)$ time units to reach vertex j (in this case, $B(i,j,t)$ is the maximum number of packets that can be sent during the time unit t on the connection $i->j$).

The centralized and decentralized techniques were tested using a simulation study. The simulation was run for a number of *Tsim* time steps. During every time step t, we performed the following actions, in order:

1. for every connection $i->j$, the first $min\{qsize(i,j), B(i,j,t-1)\}$ packets from the head of the queue associated to connection $i->j$ are delivered to peer j (where $qsize(i,j)$=the number of packets located in the queue of the connection $i->j$ at the beginning of the current time step and $B(i,j,-1)=0$);
2. a request generator generates a list of requests to be submitted during the current time step;
3. if we use the centralized method, then each newly generated request is submitted to the central scheduler, which will make decisions regarding its acceptance or rejection and, if accepted, the request will be submitted to its source node;
4. if we use the decentralized method, then every newly generated request is submitted to its source node (in this case, t_1=the submission time moment);
5. every peer P makes decisions regarding the packets received from its adjacent connections and the packets composing newly generated received requests whose source node is P; the decision for a packet consists of whether to forward it to one of P's neighbors (along the corresponding connection) or to store it until the next time step, when a decision for the packet will be made again.

4.7.1. The Centralized Approach

For the centralized approach we considered the following algorithms. The central scheduler constructs a time-expanded graph *TEG* of the network. A vertex of a time-expanded graph is a pair *(v,t)*, where *v* is a vertex of the graph and *t* is a time moment (*0≤t≤Tsim*). For every directed connection *i->j* we add the following edges in the time-expanded graph: *(i,t)->(j,t+1)* with capacity *B(i,j,t)* (*0≤t≤Tsim-1*). The meaning is obvious: at most *B(i,j,t)* packets which are located at vertex *i* at time moment *t* can arrive at vertex *j* at time moment *t+1*. We also add the edges *(v,t)->(v,t+1)* with capacity *bufsize(v,t)*, for every vertex *v* and every time moment *t* (*0≤t≤Tsim-1*). *bufsize(v,t)* is a limit on the buffer size of the vertex *v*, representing the maximum number of packets which can be stored at vertex *v* from time moment *t* to the next time moment *t+1*. In our experiments, we always used *bufsize(v,*)=+∞*.

In the more general case, in which every connection *i->j* had a latency *lat(i,j,t)*, then *TEG* would consist of the edges *(v,t)->(u,t+lat(v,u,t))*, with capacity *B(v,u,t)*, and for which *0≤t≤t+lat(v,u,t)≤Tsim* (for every connection *v->u*); then, we would also add the same edges *(v,t)->(v,t+1)* as before. However, we can always reduce this more general case to the unit duration case. For every connection *v->u* and every time moment *t* such that *lat(v,u,t)>1*, we create the vertices $(w_{\{v,u,t,i\}}, t+i)$ (*1≤i≤lat(v,u,t)-1*); then, we add the directed edges $(v,t)->(w_{\{v,u,t,1\}}, t+1)$, $(w_{\{v,u,t,i\}}, t+i)->(w_{\{v,u,t,i+1\}}, t+i+1)$ (for *1≤i≤lat(v,u,t)-2*), and $(w_{\{v,u,t,lat(v,u,t)-1\}}, t+lat(v,u,t)-1)->(u,t+lat(v,u,t))$, each of them having capacity *B(v,u,t)*. The extra vertices $w_{\{v,u,t,i\}}$ are fictitious vertices. These vertices do not make packet routing decisions; however, they do check if they have a matching reservation for every received packet, or drop the packet otherwise. We will discuss about reservations later in this section. Every edge of *TEG* also has a flow value, which is initially *0*.

When the central scheduler receives a request with identifier *rid* with the relevant parameters: *s* (source node identifier), *d* (destination node identifier), *np* (number of packets), t_1 (earliest time moment at which the transfer may start) and t_2 (the deadline), it performs the following actions. The central scheduler will compute a flow of value at most *np* in the time-expanded graph from the vertex (s,t_1) to the vertex (d,t_2) (if possible), optionally considering a load balancing criterion. We considered three load balancing criteria:

- the maximum extra flow on each edge of the time-expanded graph is minimum.
- the maximum flow on each edge of the time-expanded graph is minimum.
- the minimum available capacity on every edge of the time-expanded graph is maximum.

The scheduler constructs the following modified version *TEG'* of the time-expanded graph, which contains all the vertices *(v,tm)* of *TEG*, with $t_1≤tm≤t_2$, together with all the edges between them. It adds a virtual source node *vs* with an outgoing edge of capacity *np* to the node (s,t_1). Let's assume that *cap(a,b,c,d)* ($t_1≤b<d≤t_2$) is the capacity of the edge *(a,b)->(c,d)* in the original time-expanded graph and *flow(a,b,c,d)* is the current flow on the same edge. The capacity of an edge *(a,b)->(c,d)* in the modified version of the time-expanded graph will be *cap(a,b,c,d)-flow(a,b,c,d)* (when no load balancing criterion is used).

In order to enforce any of the three load balancing criteria, we can use binary search, combined with a slightly modified maximum network flow algorithm. The capacity of an edge *(a,b)->(c,d)* in the modified version *TEG'* of the time-expanded graph will be the minimum value between: (*cap(a,b,c,d)-flow(a,b,c,d)*) and:

- in the case of criterion *1*: *X*
- in the case of criterion *2*: *max{X-flow(a,b,c,d),0}*
- in the case of criterion *3*: *max{cap(a,b,c,d)-X-flow(a,b,c,d), 0}*

X is the value which is binary searched. For criteria *1* and *2* (*3*), we will find the minimum (maximum) value of *X* such that the maximum flow (in *TEG'*) from *vs* to the set of sinks *(d,tm)* ($t_1≤tm≤t_2$) is the same as in the case *X=+∞* (*X=0*). Thus, we first compute the maximum flow for *X=+∞* (*X=0*) and then we compute it for every value of *X* chosen by the binary search. For a fixed value of *X*, the maximum flow is computed using the Edmonds-Karp maximum flow algorithm.

When no load balancing criterion is used, we compute the maximum flow once in *TEG'*. When one of the load balancing criteria is used, we perform *O(log(max(cap(*,*,*,*))))* maximum flow computations in *TEG'* (with possibly different edge capacities each time).

The Edmonds-Karp algorithm is based on iteratively finding a path from the source *vs* to one of the sinks *(d,tm)* and increasing the flow on that path (initially, the flow on every edge of *TEG'* is *0*). We implemented every iteration of the algorithm as follows. We perform a BFS traversal of (the residual graph of) *TEG'*, considering the flow values from the previous iterations. Then, we choose the minimum value of *tm* ($t_1 \leq tm \leq t_2$) such that *(d,tm)* is reachable from *vs* (meaning that there is a path from *vs* to *(d,tm)* on which the flow can be increased) - if no such vertex is reachable from *vs*, then the algorithm stops. We increase the flow on the path found by BFS from *vs* to *(d,tm)*. This way, although the deadline is t_2, we try to complete the data transfer as soon as possible. Alternatively, we could have chosen the k^{th} ($k \geq 1$) smallest (or largest) value of *tm* ($t_1 \leq tm \leq t_2$) such that *(d,tm)* was reachable from *vs*.

After computing the maximum flow *F* (for the value of *X* found by the binary search, in the case of using a load balancing criterion; or simply in *TEG'*, when no load balancing criterion is used), if the request was for a reliable data transfer and *F=np*, then the request is accepted; otherwise it is rejected. If the requested data transfer was unreliable, the request is accepted as it is (meaning that only *F* of the *np* packets would reach the destination, and the others would not even be sent at all). If the request is accepted, we traverse all the edges of the modified time-expanded graph (except for *vs->(s,t1)*). Let's assume that we had a computed flow value *flow'(a,b,c,d)>0* on the edge *(a,b)->(c,d)* of this graph. Then, we increase *flow(a,b,c,d)* (the flow on the edge *(a,b)->(c,d)* in *TEG*) by *flow'(a,b,c,d)* and then, if *a≠c*, we make a reservation at node *a* with the following meaning: at time moment *b*, *flow'(a,b,c,d)* packets of the request *rid* which are then at vertex *a* will be sent along the connection *a->c* to vertex *c*.

4.7.2. The Decentralized Approach

Decentralized processing is performed even when a centralized scheduler is used. In this case, at every time step *t*, every peer *q* analyzes the set of packets *pkt(q,t)* composed of all the packets that were just received, were just submitted (as part of a new request during the current time step), or were stored from previous time steps. For each such packet (belonging to a request *rid*), the peer considers all of its neighbors *p* and verifies if any reservation was made for the request *rid*, the current time moment, and the neighbor *p* as the next node for the packet. If it finds such a reservation for *np>0* packets, it decreases *np* by *1* (in the reservation) and then sends the packet on the connection to peer *p* (and no further neighbors are considered); otherwise, we just proceed to the next neighbor. If the packet is not sent to any neighbor (because no matching reservation was found or because all the matching reservations were completely used for sending other packets considered before the current packet), then the packet is stored until the next time step, when it will be considered again, or is dropped if its deadline will be expired by the next time step.

Sending a packet from a peer *q* to a neighbor *p* means placing the packet at the end of the queue of the connection *q->p*. At the beginning of each time step *t*, the first *B(i,j,t-1)* packets (or less if there aren't that many packets) are removed from the head of the queue of every connection *i->j* and delivered to peer *j*.

If we do not use a centralized scheduler, then at every time step, every peer must decide what to do with the packets mentioned in the previous paragraph (previously, it only looked for matching reservations; now, it has to make a decision). Any decentralized decision making method is vulnerable mostly due to the lack of information. Because of this, we tried to offer as much information as possible to every node. First, we considered the data transfer duration *dtd(i,j)* of a packet along any directed connection *i->j* (in our case, it is always *1*). It is well-known that, using gossiping protocols, every peer *i* can compute the shortest path from itself to every other peer *j* in the network.

Such a protocol works in rounds. Initially, every peer *i* maintains an estimation *shp(i,j)=dtd(i,j)* for every neighbor *j* (for which a directed connection *i->j* exists), plus the

estimation *shp(i,i)=0*. Then, at every gossiping round, every peer *i* sends all of its estimations *shp(i,*)* to all of its neighbors *j* (for which a connection *i->j* exists). Thus, peer *i* receives all the estimations *shp(j,k)* that every neighbor *j* of *i* has. If peer *i* had no previous estimation *shp(i,k)*, then it sets *shp(i,k)=dtd(i,j)+shp(j,k)*; otherwise, it sets *shp(i,k)=min{shp(i,k), dtd(i,j)+shp(j,k)}*. After a number of rounds proportional to the diameter of the network, the estimations *shp(i,*)* of every peer *i* converge to the shortest path values. Since the network does not change in time, we can assume that the peers performed the total number of rounds of gossiping required for the estimations to be final before the first data transfer request is submitted. We made the implicit assumption that the total number of nodes in the network is not too large and, as such, every peer can learn about every other peer.

Note that gossiping is a very useful method of exchanging information, particularly in a network whose structure does not change. Through gossiping, every peer could build its own view of the entire network. Initially, the network view of every peer *i* consists of itself, its neighbors and the directed connections *i->j* to its neighbors *j*. At every gossiping round, every peer *i* sends its network view to all of its neighbors. Then, after peer *i* receives the network view of a neighboring peer *j*, it will add the network view of *j* to its own view: it will add all the new peers and edges that were not in the network view of vertex *i* before this. Thus, we will assume that every peer has a perfect view of the entire network in which it is located and it is able to obtain this view before the first request is submitted. At this point, every peer has all the static information that is available in the network.

We considered several simple decision making procedures. The first one (*DDM1*) sorts all the packets from *pkt(q,t)* in increasing order of their deadlines (excluding those packets with missed deadlines). Then, it considers every packet in this order (at the current time step *t*) and tries to send it randomly to one of the neighbors *p* whose distance to the destination is the smallest (only if the packet's deadline would not be exceeded by the time it reached its destination, in which case, the packet is dropped). If the connection towards every such neighbor *p* already has *B(q,p,t)* packets waiting to be sent in its queue, then the packet is stored until the next time moment.

The second procedure (*DDM2*) considers that every peer has a self-generated identifier (possibly a point in a multidimensional space; see, for instance, section 3.1). We sort the packets like before and, for each packet, we consider the set of neighbors *p* whose identifiers are closer (using an appropriate distance function) to the destination ID than the ID of the current peer *q* and, if we sent the packet to *p* and then on the shortest path from *p* to the destination, the packet's deadline would not be missed, and such that fewer than *B(q,p,t)* packets are already in the queue of the connection *q->p*. From this set, a peer *p* to which the packet is sent is chosen randomly. If the set is empty, the packet is stored until the next time moment.

The third approach (*DDM3*) is the following. When a source peer receives a request, it computes a deadline-constrained flow from the source to the destination, considering that all the network connections are available (i.e. there are no other packets in the network). The flow is computed by considering the same (modified) time-expanded graph that was used in the case of the central scheduler, except that the capacity of an edge *(i,t)->(j,t+1)* is always equal to *B(i,j,t)*. This is possible because, as we said earlier, every peer can find, through gossiping, the structure of the entire network.

In the case of reliable requests, the flow must be equal to the number of packets of the request; otherwise, the request is rejected. In case of unreliable data transfer requests, the request is accepted no matter what. If the request is accepted, the computed flow is decomposed into several (not necessarily disjoint) paths. On each path, a certain amount of packets can be sent. Let the vertices of the path be *(v(1),t(1)), ..., (v(k),t(k))* (*v(1)*=the source) in the order in which they occur on the path from the source to the destination. We compute the sequence *v'(1), ..., v'(k')* consisting of the sequence of vertices *v(1), ..., v(k)*, from which multiple consecutive occurrences of the same node were replaced by just one occurrence. Let *np'* be the number of packets that could be sent on the path. We assign to *np'* packets of the request the sequence *v'(2), ..., v'(k')*, representing the path on which the packet has to be routed (from which we excluded the source node). Then, these

packets are considered for routing during the current time step by the source node. This procedure is called *source routing*.

At every time step t, every peer q sorts the packets in $pkt(q,t)$ in increasing order of their deadlines and considers them in this order. For every packet p, if q is the destination of the packet (i.e. packet p's sequence of vertices is empty), then it sends the packet to the upper layer application. Otherwise, if t plus the number of vertices in packet p's sequence is greater than p's deadline, then p is dropped; otherwise, p is sent to the neighbor v which is at the head of packet p's sequence of vertices (only if the queue of the connection towards v does not already contain $B(q,v,t)$ packets; if it does, then the packet is stored until the next time moment); if the peer decides to send the packet, then the vertex v is removed from packet p's sequence of vertices.

4.7.3. Simulation Results

We developed a simulation framework in the Python programming language. No load balancing criteria were used in the simulation, because of the long duration of the simulation in this case.

We first ran *5* tests for reliable data transfers. Each simulation test was run for *99* time steps and *1000* requests of *500* packets each were generated, using several distributions. We considered the *Uniform*, *Gauss*, *Cauchy*, *Pareto* and *Weibull* distributions. Requests were generated on the interval of time steps *[0,Tmax]* only ($Tmax=84$); the t_1 parameter of each request was always equal to its submission time moment. Each request had a deadline which was equal to *15* time steps after the submission time moment. The network was generated randomly and consisted of *50* vertices and *250* connections between them (actually, each connection between two vertices i and j consisted of *2* directed connections, in both directions); the network was connected. The bandwidth of each connection was a randomly generated integer number between *1* and *25*. The two directed connections between the same pair of vertices had the same bandwidth.

The only difference between the *5* distributions consisted in the number of requests generated during each time step t ($0 \le t \le Tmax$). The number of requests generated at each time step for every distribution (except *Pareto*) are shown in Fig. 4-2. The Pareto distribution generated *428* requests during each of the first *2* time steps, and then the number of requests decreased progressively until $t=11$, after which no more requests were generated. The exact distribution parameters used were: $Gauss(average=Tmax/2,\ standard\ deviation=Tmax/6)$, $Cauchy(x_0=Tmax/2,\ gamma=0.6 \cdot x_0)$, $Pareto(x_m=1,\ \alpha=1.5)$, and $Weibull(k=2,\ \lambda=0.36 \cdot Tmax)$. Fig. 4-3 shows the number of requests which were satisfied in each case.

Then, we used the same tests, except that the requests asked for unreliable data transfers. In these cases, we counted the number of packets which reached their destination on time (before or exactly at the deadline): these numbers are shown in Fig. 4-4.

As can be noticed, the centralized approach outperformed the decentralized techniques in the case of reliable data transfer requests. In the case of unreliable data transfer requests, the difference between the centralized and decentralized methods was not as great, but the centralized approach was still better.

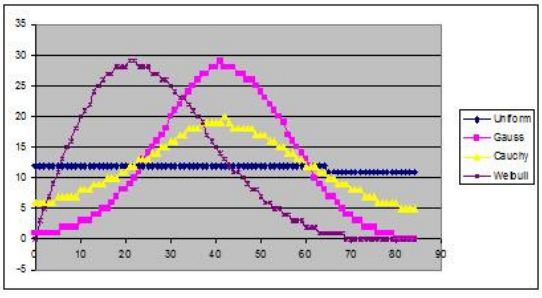


Fig. 4-2. Number of Requests generated at each Time Steps.

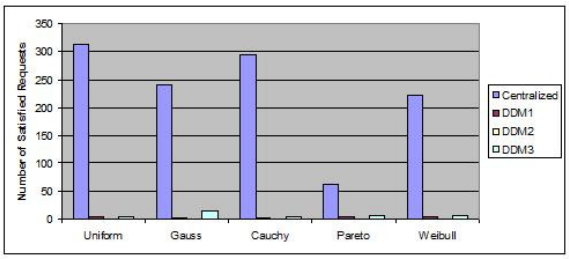


Fig. 4-3. Test Results – Reliable Data Transfers.

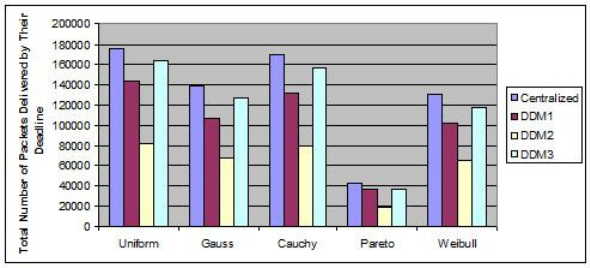


Fig. 4-4. Test Results – Unreliable Data Transfers.

4.8. An Event-Based Real-Time Data Transfer Scheduling Framework

In this section we propose an architecture for a data transfer scheduling framework, which is similar in nature to the one presented in [Cîrstoiu, 2008]. The framework consists of several components:

- the Communication Flow Scheduling and Optimization Component
- the Data and Information Management Component
- the Communication Flow Management Component
- the User and Application Interface
- Interface to a Monitoring System (e.g. MonALISA [Legrand et al., 2004])
- the Prediction and Pattern Detection Component
- the Simulation Component
- the (Self-) Monitoring and (Self-) Evaluation Component
- the (Self-) Reconfiguration Component.

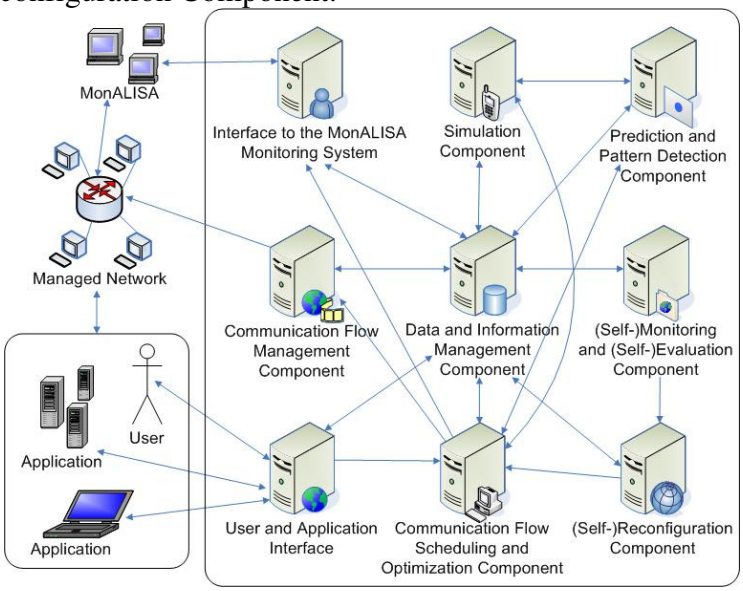


Fig. 4-5. Architecture of the Data Transfer Scheduling Framework.

Fig. 4-5 presents all the components, together with the directions of the command and data flows between them. We intend to use the MonALISA monitoring system [Legrand et al., 2004] to provide monitoring data to the scheduling framework (i.e. information about the relevant network

parameters and about the status of the running data transfers). The core of the framework is the Communication Flow Scheduling and Optimization Component, which runs the optimization algorithms and makes the scheduling decisions. This component may use simulations (the Simulation Component) or pattern detection and data transfer request prediction techniques (the Prediction and Pattern Detection Component) in order to make improved scheduling decisions.

The decisions of this component are transformed into commands for the network nodes by the Communication Flow Management Component. The Data and Information Management Component stores all the data of the framework and, as such, it is connected to all the other components. The (Self-) Monitoring and (Self-) Evaluation Component monitors the quality of the decisions made by the scheduling component. If they are not of sufficient quality, it may use the services of the (Self-) Reconfiguration Component in order to reconfigure the Communication Flow Scheduling and Optimization Component (e.g. change the scheduling algorithm, switch from a time-slot based to an event-based time interpretation).

The Communication Flow Scheduling and Optimization component also contains a rescheduling submodule, which takes care of data transfers which either do not use the requested bandwidth or take more than the allotted time interval. There are two main choices for this component's rescheduling strategy. The first one is to terminate the transfers which take longer than the allotted time interval. However, if transfer preemption is not allowed, then a terminated transfer would need to be restarted and the network resources consumed so far would be wasted. The second choice would be to extend the time interval of the transfers which are taking too long and reschedule all the transfers which have not already been started. This might require a lot of work on the scheduler's part, as there may be many transfers that need to be rescheduled.

In the rest of this section we will present an event-based scheduling algorithm for fixed-bandwidth fixed-duration data transfer requests (the algorithm works on a very different principle than the time slot-based algorithms and data structures presented earlier in this chapter). Then, we will discuss an implementation of the proposed data transfer scheduling framework as a simulation framework.

4.8.1. An Event-Based Scheduling Algorithm

In this section we will present an algorithm for the case when only *fixed bandwidth-fixed duration* requests are considered. The algorithm is based on events (as opposed to time-slot based algorithms). For each event, the time moment and the value by which the bandwidth is modified (a positive or negative value) are stored, i.e. a pair *(t,dB)*. For each network link (and direction), a list of events is maintained. The time moment of an event in the event list of a network link is represented by the start or finish time of a data transfer.

The algorithm starts by generating several candidate paths from *s* to *f*, which satisfy the sum of latencies constraints. Then, for each path, we verify if we can schedule the data transfer request on that path in order to satisfy all the other constraints. We consider a family of greedy algorithms for this case: *First-Fit*, *Last-Fit*, *Best-Fit* and *Worst-Fit*. All the events on all the (directed) network links along the candidate path are initially sorted. Also, for each (original) event *(t,dB)* on a link *l*, an event *(t+D,0)* on the same link is added, where *D* is the duration requested for the data transfer. These events are sorted together with the other events. Then, the sorted events are traversed.

During the traversal we maintain for each network link *l* a deque *DQ(l)* and the current available bandwidth *cb(l)* of the link (*cb(l)* is initially the total bandwidth of the link). The deque maintains sorted *(tm=time moment, ab=available bandwidth)* pairs, similar to the deque presented for the Time Slot Groups data structure in section 4.3.5. Let's assume that we reached an event *(t,dB)* on a link *l*. First, we consider every link *l'* and:

1. while *DQ(l')* contains at least *1* pair and the last pair *lp* from *DQ(l')* has *lp.ab*≥*cb(l')*, we remove *lp* from *DQ(l')*;
2. we insert the pair *(t,cb(l'))* at the end of *DQ(l')*;
3. while the first pair *fp* of *DQ(l')* has *fp.tm*≤*t-D*, we remove *fp* from *DQ(l')*; then, we increment *cb(l)* by *dB*.

After all these, the first pair *fp* of *DQ(l')* of each link *l'* contains the minimum available bandwidth *AB(l')=fp.ab* of that link on the interval *[t-D,t]*. If *AB(l')* is greater than or equal to the required bandwidth *B* for every network link *l'* on the candidate path and the interval *[t-D,t]* is included in $[t_1,t_2]$ (given by the request), then a match is found and we perform the following actions. In the case of the First-Fit algorithm, we return the interval *[t-D,t]* as the solution (the request can be scheduled on the path). For the Last-Fit Algorithm we just store *t* into *tfin*. For the Best-Fit and Worst-Fit algorithms we compute *MAB=min{AB(l')|l' is a link on the candidate path}*. These two algorithms will maintain a bandwidth value *MB*, initially equal to $+\infty$ for Best-Fit and $-\infty$ for Worst-Fit. If $MAB<MB$ for Best-Fit ($MAB>MB$ for Worst-Fit) we set $MB=MAB$ and $tfin=t$ (note that $MAB \geq B$, because otherwise we wouldn't have performed these actions). Then, at the end, if anything was stored in *tfin* (in the case of the Last-Fit, Best-Fit and Worst-Fit algorithms), we return the interval *[tfin-D,tfin]*; otherwise, no solution is found.

If we schedule a request on a path between time moments *ts* and *ts+D*, we insert the events *(ts,-B)* and *(ts+D,+B)* in the event list of every link *l* on the path (for the correct direction). The sorting stage of the events can be performed by maintaining a balanced tree, which contains the next event to occur for each network link. At each step, the event occurring at the earliest time is extracted from the balanced tree (and replaced by the next event on its link). The algorithm stops as soon as the latest finish time of the request is passed by. The overall time complexity is $O(E \cdot (\log(NL)+NL))$, where *E*=the total number of events and *NL*=the total number of links on the candidate path.

4.8.2. The Data Transfer Scheduling Framework Simulator

In order to easily test the performance and behavior of the scheduling algorithms and of the whole framework under controlled test conditions, we developed a simulator for the data transfer scheduling framework. The simulator controls all the actions in the system with a given time step granularity. The simulator was implemented in the Java programming language and follows closely the architecture of the framework. The main modules of the simulator are:

- Graph Manager
- Data Transfer Scheduler
- Simulator Engine
- Testing Framework

4.8.2.1. The Graph Manager

The Graph Manager module contains classes for representing network nodes, network links and network paths. The nodes and links have an associated probability to fail at each time step. The Graph Manager maintains the state of the network and computes shortest paths, using the Bellman-Ford-Moore algorithm.

4.8.2.2. The Data Transfer Scheduler

The Data Transfer Scheduler accepts data transfer requests at each time step. The requests can be given one by one or in batch mode (multiple requests at a time). For the single request mode, the First Fit scheduling algorithm is run and the request is either scheduled or rejected (if the request's constraints cannot be satisfied). For batch submissions, an exponential algorithm is used. The algorithm tries all the permutation orderings of the requests and calls the First Fit algorithm for each request, in the order given by the permutation. Each request also has an associated profit value (which could be its priority). The ordering which maximizes the total profit of the scheduled requests (those that are not rejected) represents an optimal ordering. Although the algorithm is exponential, it is useful for comparing the performance of other scheduling algorithms against the optimal solution.

4.8.2.3. The Simulator Engine

The simulator engine controls the entire simulation. It increments the time step and calls the callback functions of the simulated entities. All the simulation takes place in a single thread. The simulator engine was designed this way in order to have everything under control. There are many realistic existing simulators (e.g. [Dobre and Stratan, 2004], [ns2]), but our purpose was not to develop an all-purpose simulator, but rather a very specific one, for the problem at hand.

4.8.2.4. The Testing Framework

The testing framework consists of test scenarios and request submitters. The test scenarios define the network nodes, links and their parameters (latency, bandwidth, failure probability). The request submitters define the parameters of the requests to be submitted (earliest start time, latest finish time, duration, required bandwidth, start and finish nodes, profit/priority, bandwidth usage distribution) and decide if the requests will be submitted one by one or as a batch.

4.8.2.5. Simulation Results

We used the simulator for comparing the online submission of requests to the batch submission, when the First Fit Algorithm is used. The batch case was handled as follows. Every possible permutation P of the requests in the batch was generated. Then, the First-Fit algorithm was used, as if the requests were submitted online in the order given by the permutation. Each request had an associated profit. The sum of the profits of the accepted requests for a given permutation P is denoted by $Psum(P)$. The permutation P_{opt} with the largest value $Psum(P_{opt})$ was chosen and the requests were scheduled by using the First-Fit algorithm and the (online) order of requests given by P_{opt}.

When only the First-Fit algorithm was used, the same requests submitted in the batch case were also submitted now, but online, in some arbitrary order. We computed the sum of the profits of the accepted requests.

We found situations in which the First Fit algorithm (in the online case) produces schedules which are significantly worse than in the batch case. This was to be expected, as First Fit is a simple algorithm and there exist request orderings for which its performance is very poor. However, the advantage of the algorithm is that it is quite fast and has a good response time.

4.9. Scheduling of Data Transfer Requests with Earliest Start Time, Latest Finish Time and a Tree Mutual Exclusion Graph

We are given a time horizon H, n requests and pairs of incompatible requests forming a tree mutual exclusion graph. For each request i, the duration d_i and the profit p_i (obtained if the request is scheduled) are known. Furthermore, each request has an earliest start time ES_i and a latest finish time LF_i. This means that request i should be scheduled within an interval $[t,t+d_i]$ fully included in $[ES_i,LF_i]$. We will provide a pseudopolynomial dynamic programming algorithm for this problem, for the case when all the time values are integers. We consider that the tree mutual exclusion graph is rooted at an arbitrary vertex r. For each tree vertex i, we will compute two arrays:

- $S_i(t)$=the maximum profit that can be obtained if the time interval available for scheduling vertex i is $[0,t]$
- $F_i(t)$=the maximum profit that can be obtained if the time interval available for scheduling vertex i is $[t,H]$

  For each vertex i we have:
- $S_i(0)=F_i(H)=\sum_{\text{json of i}} S_j(H)$
- $$S_i(0<t\le H)=\max\{S_i(t-1),\begin{cases} p_i+\sum_{\text{json of i}}\max\{S_j(t-d_i),F_j(t)\}, & \text{if } t-d_i\ge ES_i \text{ and } t\le LF_i \\ 0, & \text{otherwise}\end{cases}\}, \quad \textbf{(4-8)}$$

- $$F_i(0 \le t < H) = \max\{F_i(t+1), \begin{cases} p_i + \sum_{\text{json of i}} \max\{S_j(t), F_j(t+d_i)\}, \text{if } t \ge ES_i \text{ and } t + d_i \le LF_i \\ 0 \qquad\qquad\qquad\quad , \text{otherwise} \end{cases}, \qquad \textbf{(4-9)}$$

The maximum profit that can be obtained is $S_r(H)$. The time complexity of an algorithm which implements these equations is $O(n \cdot H)$; the tree is traversed from the leaves towards the root. For each vertex i we compute the values $S_i(t)$ in increasing order of t, and the values $F_i(t)$ in decreasing order of t. We can extend these results to the case when the mutual exclusion graph is a forest. We run the algorithm for each connected component of the mutual exclusion graph (rooted at any vertex in it); the maximum total profit that can be obtained is the sum of the maximum profits that can be obtained for each connected component of the forest mutual exclusion graph.

4.10. Scheduling File Transfers with a Mutual Exclusion Graph – M Intersecting Cliques

We consider n file transfer requests. Each of them has a pre-assigned source, destination(s) and network path (multicast tree) on which the transfer must be performed. For each request i, the transfer duration d_i and profit p_i are also known. The transfers must be scheduled non-preemptively, i.e. during a continuous time interval, without interruptions. Because the network paths (trees) of some pairs of transfers may cross, the two transfers must not be scheduled during overlapping time intervals.

We define the mutual exclusion graph as a graph containing the file transfer requests as vertices and there is an edge between two vertices i and j if the corresponding requests are in conflict. Given a deadline T, we want to schedule a subset of requests whose total profit is maximum, such that no two conflicting requests are scheduled at the same time. The mutual exclusion scheduling is an NP-hard problem and polynomial time algorithms are known only for some particular situations of the mutual exclusion graph. In this section we consider the case when the mutual exclusion graph consists of $M \ge 2$ intersecting cliques (complete subgraphs) and the durations are integer numbers. Any pair of cliques (C_a, C_b) has the same common intersection CI. We define $X_j = C_j \setminus CI$.

When $|CI|$ is bounded by a constant ct, we present a pseudo-polynomial dynamic programming algorithm, using ideas borrowed from the well-known knapsack problem. Let's assume that $|CI|=k$ and that a vertex i in CI also has an associated earliest start time $ES(i)$ and latest finish time $LF(i)$, i.e. the file transfer corresponding to request i cannot start before $ES(i)$ and cannot finish after $LF(i)$. This problem is equivalent to a multiple knapsack problem.

The states S of the problem are defined by a sequence of non-decreasing $2 \cdot k$ numbers: $S=(t_1, t_2, \ldots, t_{2 \cdot k})$. These numbers represent k time intervals: $[t_1,t_2]$, $[t_3,t_4]$, …, $[t_{2 \cdot k-1}, t_{2 \cdot k}]$; the meaning of these intervals is that no request has been scheduled within any of the intervals. We will compute $PM_j(i,S)$=the maximum profit of a subset of the first i requests from the set X_j (considering some arbitrary order $X_j(1), X_j(2), \ldots, X_j(i), \ldots$), scheduled outside the time intervals defined by the state S. By excluding the k intervals defined by a state S from the interval $[0,T]$, we obtain $(k+1)$ intervals into which a request from X_j can be scheduled. We will consider the request to be scheduled either to the left of $t_1, t_3, t_5, \ldots t_{2 \cdot k-1}$ or to the right of $t_2, t_4, \ldots, t_{2 \cdot k}$. Initially, we have $PM_j(0,S)=0$, for all the states S. For $i>0$, we have:

$$PM_j(i, t_1, t_2, \ldots, t_{2 \cdot k}) = \max \begin{Bmatrix} PM_j(i-1, t_1, t_2, \ldots, t_{2 \cdot k}) \\ p_{X_j(i)} + PM_j(i-1, t_1 - d_{X_j(i)}, t_2, \ldots, t_{2 \cdot k}), \text{if } (t_1 - d_{X_j(i)} \ge 0) \\ p_{X_j(i)} + PM_j(i-1, t_1, t_2 + d_{X_j(i)}, \ldots, t_{2 \cdot k}), \text{if } (t_2 + d_{X_j(i)} \le t_3) \\ \ldots \\ p_{X_j(i)} + PM_j(i-1, t_1, t_2, \ldots, t_{2 \cdot k-1} - d_{X_j(i)}, t_{2 \cdot k}), \text{if } (t_{2 \cdot k-1} - d_{X_j(i)} \ge t_{2 \cdot k-2}) \\ p_{X_j(i)} + PM_j(i-1, t_1, t_2, \ldots, t_{2 \cdot k-1}, t_{2 \cdot k} + d_{X_j(i)}), \text{if } (t_{2 \cdot k} + d_{X_j(i)} \le T) \end{Bmatrix}. \qquad \textbf{(4-10)}$$

After computing the tables PM_j $(1 \le j \le M)$, we will consider every subset SCI of CI and for all the vertices in SCI, we will consider all of their permutations. For each permutation pe with q

elements *(pe(1), pe(2), …, pe(q))*, we consider every subset *Spe* of *q* elements of the set *{1, 2, …, k}* and denote its elements by *Spe(1)*, …, *Spe(q)*, such that *Spe(1)<Spe(2)<…<Spe(q)*. For each such permutation *pe* and subset *Spe*, we will consider every state $S=(t_1, t_2, \ldots, t_{2 \cdot k})$.

A state *S* is consistent with a (permutation *pe*, subset *Spe*) pair if every vertex *pe(i)* can be scheduled within the interval $[t_{2 \cdot Spe(i)-1}, t_{2 \cdot Spe(i)}]$. The profit of this subset-permutation-subset-state tuple is equal to $PM_1(|X_1|, S) + PM_2(|X_2|, S) + \ldots + PM_M(|X_M|, S) + p_{pe(1)} + p_{pe(2)} + \ldots + p_{pe(q)}$. The maximum profit scheduling corresponds to the maximum subset-permutation-subset-state tuple. The time complexity of this method is very large and can only be used for cliques whose intersection contains a very small number of vertices. We implemented the method for $|CI|=k=1$, for which the time complexity becomes $O(n \cdot T^2)$, which is quite reasonable.

4.11. Maximum Profit Scheduling of Data Transfer Requests using Conflict Graphs

We consider here the following scheduling problem. We have *N* (multicast or point-to-point) data transfer scheduling requests. The time horizon over which the data transfers can be scheduled is divided into *T* time slots (numbered from *1* to *T*). A request *i* asks for exclusive access to a specific set of network links during a given interval of time slots *[ts(i), tf(i)]* and, if accepted, it brings a profit of *p(i)*. Two requests whose time slot intervals overlap may be in conflict if they ask for exclusive access to at least one common network link. We will model these conflicts by using a conflict graph *CG* in which we have a vertex for every request *i* ($1 \le i \le N$) and an edge between two requests *i* and *j* if their intervals overlap and they are in conflict.

Using this model, we want to find an independent set *IS* within the conflict graph (i.e. there is no edge between any two requests *a* and *b* from *IS*), such that the sum of the profits of the requests in *IS* is maximum. All the requests in *IS* will be accepted and all the other requests will be rejected. The problem of computing a maximum weight independent set in an arbitrary graph is an NP-hard problem. We will present here a solution for a restricted case. We maintain two lists of events for each time slot *t*: a list *LAE(t)* with *activation events* (when a new request becomes active) and a list *LDE(t)* with *deactivation events* (when a request becomes inactive). An activation event for a request *i* is added to the list *LAE(ts(i))* and the deactivation event is added to the list *LDE(tf(i)+1)*. We will traverse the time slots in increasing order and, during the traversal, we will maintain a set *S* of subsets of requests: *S(0), …, S(k-1)* ($k=|S|$). *S(0)* will always exist and will always be void (empty).

For each time slot *t* ($1 \le t \le T$), in increasing order, we will handle all the events in *LDE(t)* first, followed by all the events in *LAE(t)*. For each deactivation event for a request *i*, we find the set *S(j)* which contains the request *i* and remove *i* from *S(j)*; if *S(j)* becomes void, we will remove *S(j)* from *S*. For each activation event for a request *i*, we will consider all the requests *j* such that there exists an edge *(i,j)* in the conflict graph. Let *S(jj(1)), …, S(jj(q))* be the subsets which contain all the requests *j* which are *i*'s neighbors in *CG* (some of these neighbors may not belong to any subset *S(x)*, because their activation events have not been handled, yet; these neighbors will be ignored). We will construct a set *SQ* from the union of *S(jj(1)), …, S(jj(q))*, and then remove all these subsets from *S*. Afterwards, we will add *i* to *SQ* and we will insert *SQ* into *S*. We will present an algorithm which will use the subsets *S(*)* and which is efficient in the following case: at any moment during the execution of the algorithm, the cardinality $|S(j)|$ of any subset *S(j)* is at most *CMAX*, where *CMAX* is a small constant value (i.e. the cardinality is bounded by a constant).

Let the vertices of a subset *S(j)* be *v(j,1), …, v(j,|S(j)|)*. For each subset *S(j)* we will maintain a table $T_j(state)$, where *state* is a tuple with $|S(j)|$ binary values (i.e. *0* or *1*); we denote the i^{th} of these values by *state(i)*. If *state(i)=1*, then we consider that *v(j,i)* belongs to *IS*; otherwise, *v(j,i)* does not belong to *IS*. There are $2^{|S(j)|}$ such states. Every vertex *x* of *CG* can be assigned a label *label(x)=p* which means that the activation event of vertex *x* was/will be the p^{th} such event processed during the algorithm ($p \ge 1$). The value $T_j(state)$ is equal to the maximum profit which can be achieved if the vertices *v(j,*)* are in the state defined by *state*, and we have already considered all the vertices *x* with $label(x)<min\{label(v(j,*))|1 \le j \le |S(j)|\}$ which are in the same connected

component of *CG* as the vertices *v(j,*)*. If *S(j)* contains no vertices, then we have only one possible state, which is the empty tuple *{}*. If we traverse the time slots all the way to *T+1*, then we will eventually process all the deactivation events and, in the end, the only remaining subset in *S* will be the empty set, *S(0)*. $T_0(\{\})$ will be the maximum weight of an independent set *IS* of *CG*.

We will now describe how the values $T_j(*)$ are maintained by the algorithm after processing every activation and deactivation event. Initially, we only have the set *S(0)*, with $T_0(\{\})=0$. When the algorithm processes the deactivation event of a request *i*, it finds the set *S(j)* which contains the request *i*. Let's assume, w.l.o.g., that, within the set *S(j)*, we have $v(j,|S(j)|)=i$ (we can change the order of the vertices in *S(j)* such that *i* is the last vertex). We will now consider every state *s* with $|S(j)|-1$ binary values and we will compute a new table $T_{new,j}(*)$, where $T_{new,j}(s)=max\{T_j(s(0), \ldots, s(|S(j)|-1), 0), T_j(s(0), \ldots, s(|S(j)|-1), 1)\}$. Afterwards, we remove vertex *i* from *S(j)* and we replace T_j by $T_{new,j}$ (i.e. we set $T_j=T_{new,j}$). If *S(j)* contains no more vertices, then we add $T_j(\{\})$ to $T_0(\{\})$ and, afterwards, we remove *S(j)* from *S*.

When the algorithm processes an activation event for a request *i*, it proceeds as follows. It finds the sets *S(jj(1)), …, S(jj(q))* which contain all the neighbors *j* of *i* with *label(j)<label(i)*. Then, it constructs the set *SQ* as the union of these sets. We will consider that the vertices of *SQ* are ordered as follows: *v(jj(1),1), …, v(jj(1),|S(jj(1))|), v(jj(2), 1), …, v(jj(2), |S(jj(2))|), …, v(jj(q),1), …, v(jj(q), |S(jj(q))|)*.

Then, we will compute a table *Taux*. We consider every combination of states *st(jj(1)), …, st(jj(q))*, corresponding to the sets *S(jj(1)), …, S(jj(q))* and we set *Taux(st(jj(1),1), …, st(jj(1), |S(jj(1))|), st(jj(2), 1), …, st(jj(2), |S(jj(2))|), …, st(jj(q), 1), …, st(jj(q), |S(jj(q))|))=* $T_{jj(1)}(st(jj(1)))+\ldots+T_{jj(q)}(st(jj(q)))$.

If the set *SQ* is empty, then the table *Taux* contains only one entry, corresponding to the empty tuple: *Taux({})=0*. Afterwards, we will construct the set *SQ'*, as the union of *SQ* and *{i}* (*i* will be the last vertex in *SQ'*) and we will compute a table $Taux_2$. For every state *stq* for which an entry exists in *Taux*, we set $Taux_2(stq(1), \ldots, stq(|SQ|), 0)=Taux(stq)$. Then, for each such state *stq*, let's consider the positions *pos* corresponding to the neighbors *j* of *i* with *label(j)<label(i)*. If *stq(pos)=0* for every such position, we set $Taux_2(stq(1), \ldots, stq(|SQ|), 1)=Taux(stq)+p(i)$; otherwise, we set $Taux_2(stq(1), \ldots, stq(|SQ|), 1)=-\infty$. Then, we add the set *SQ'* to *S* (after removing from *S* all the sets *S(jj(1)), …, S(jj(q))*). If *SQ'* is assigned the index *p* (i.e. *SQ'=S(p)*), then we set $T_p=Taux_2$.

The time complexity of the algorithm is $O(T+N\cdot 2^{CMAX})$. By maintaining the tables $T_j(*)$ after each processed event, we can compute the actual solution (which requests are accepted) and not just the maximum profit. We mention that the algorithm also works without dividing the time into time slots. In this case, every request has a time interval *[ts(i), tf(i))* and we construct an activation event *(time=ts(i), type=+1, request=i)* and a deactivation event *(time=tf(i), type=-1, request=i)*. We then sort these requests, first in increasing order of the time moment and, for equal time moments, we place the deactivation events before the activation events occurring at the same time moment. In this case, the complexity is $O(N\cdot log(N)+N\cdot 2^{CMAX})$.

4.12. Scheduling Data Transfer Requests with a Common Deadline and a Tree Mutual Exclusion Graph

This problem can be stated as follows: Given is a deadline *T* and *n* data transfer requests. For each request, its duration d_i and a profit p_i are known. Each request must be scheduled on some interval $[t,t+d_i]$, included in *[0,T]*. Some pairs of requests are incompatible, i.e. their assigned time intervals cannot overlap. Incompatibilities are due to using common network resources (links or paths). The mutual exclusion graph defined by the requests as vertices and by the incompatible pairs as edges forms a tree. We are trying to schedule some of the requests (and reject the others) in such a way that the total profit of the scheduler requests is maximum.

We will provide an *O(n)* dynamic programming algorithm for this problem. First, we will choose a root for the tree, which will the define parent-son relationships between the vertices of the tree. For each vertex *i*, we wil compute two values:

- *A(i)*=the maximum profit obtained by scheduling some of the requests in *i*'s subtree and

letting vertex *i* be one of the scheduled requests

- *B(i)*=the maximum profit obtained by scheduling some of the requests in *i*'s subtree and vertex *i* is not scheduled

We will assume that d_i is at most *T* for all the vertices *i*; otherwise, vertex *i* cannot be scheduled under any circumstance (*A(i)* will be set to -∞). For each leaf *i* of the tree, we have $A(i)=p_i$ and *B(i)=0*. For a vertex *i* which is not a leaf, we have:

- $$A(i) = p_i + \sum_{j \text{ son of } i} \begin{cases} \max\{A(j), B(j)\}, & \text{if } d_j \le T - d_i \\ B(j), & \text{if } d_j > T - d_i \end{cases} \quad \textbf{(4-11)}$$
- $$B(i) = \sum_{j \text{ son of } i} \max\{A(j), B(j)\} \quad \textbf{(4-12)}$$

For the case of *B(i)*, the equation is obvious. If we do not schedule *B(i)*, then there is no restriction for the choices we can make for each son of *i*. If we choose to schedule *i*, then the chosen time interval will be $[0,d_i]$. For each son *j* of *i*, we may choose not to schedule it, in which case vertex *i* will not influence any vertex in *j*'s subtree. If we do choose to schedule it, however, we only have at our disposal the interval $[d_i, T]$. But when we computed *A(j)*, we considered that *j* would be scheduled in the interval $[0,d_j]$, which is obviously not possible now. What we have to do is to realize that the time moments *0* and *T* are equivalent, so the interval $[0,d_j]$ is equivalent to the interval $[T-d_j,T]$. Thus, we can use the value of *A(j)*, considering that *j* would be scheduled at the other end of the time horizon, i.e. in the interval $[T-d_j,T]$.

4.13. Semi-Dynamic Maximum Capacity Path Queries

In [Andreica and Ţăpuş, 2009c] we considered the problem of answering maximum capacity path queries in a static undirected graph. Every edge *e* has a capacity *cap(e)* and we want to answer efficiently queries of the form: *Q(u,v)*=the maximum possible value of the minimum capacity of an edge on a path from *u* to *v*. The answers to these queries are based on using the disjoint sets data structure. We compute the lowest common ancestor *LCA(u,v)* of *u* and *v* in the tree obtained as a result of the *Union* disjoint sets operations called with arguments *(x,y)* for every edge *e* connecting two vertices *x* and *y*, in non-increasing order of the edge capacities.

In this section we extend the results from [Andreica and Ţăpuş, 2009c] to the following semi-dynamic situation. Initially, we have a graph with *n* vertices and *0* edges. Then, we consider a sequence of operations of two types: *1)* insert an edge *e* between *u* and *v* with capacity *cap(e)* ; and *2)* answer *Q(u,v)* considering the current structure of the graph. The constraint is that the edges are inserted in non-increasing order of their capacities. When an edge *e* between *u* and *v* is inserted (encountered), we perform the same *Union* steps as in [Andreica and Ţăpuş, 2009c]. In order to answer a query *Q(u,v)*, we need to find the lowest common ancestor of *u* and *v* in the current disjoint sets tree representation. In order to do this, we start at vertex *u* and mark *u* and all of its ancestors (initially, no vertex is marked). Then, we start at *v* and move up the tree (from a vertex to its parent), until we reach a vertex *a* which is marked. Then, *LCA(u,v)=a* and the time complexity of finding *a* is *O(log(n))* (as the height of the tree is *O(log(n))*). The answer to the query is the minimum *weight* of a tree edge on the paths from *u* to *a*, or from *v* to *a*. Then, we unmark all the previously marked vertices (i.e. all the ancestors of *u*, including *u*).

Chapter 5 – Offline Optimal Scheduling of Constrained Point-to-Point Communication Flows

In this chapter we consider several offline scheduling problems for constrained communication flows, for which we provide novel algorithmic solutions. The results presented in this chapter were published in [Andreica, Pârgaru, Ionescu and Andreica, 2009], [Andreica and Ţăpuş, 2009a], [Andreica and Ţăpuş, 2009c], [Andreica, Andreica and Andreica, 2009], [Andreica and Ţăpuş, 2008c], [Andreica and Ţăpuş, 2008g], [Andreica and Ţăpuş, 2008j], [Andreica, Andreica and Andreica, 2008] and [Andreica, Grigorean and Ţăpuş, 2009].

5.1. High Multiplicity Scheduling of File Transfers with Divisible Sizes on Multiple Classes of Paths

We consider C different file types: for each type, the size of the file (sz_i) and the number of files (nf_i) are given. We consider the file types sorted in decreasing order of their sizes: $sz_1>sz_2>\ldots>sz_C$. The sizes further satisfy the condition that $sz_C|sz_{C-1}|\ldots|sz_1$, i.e. sz_i is a multiple of sz_{i+1}, $1\leq i<C$. We are concerned with scheduling the transfer of all the files using P classes of disjoint paths, subject to minimizing the maximum completion time (makespan). For each class, the number of paths in the class (np_i) and their slowdown factor (sd_i) are given. The transfer of a file of type j on a path of type i will take $sd_i \cdot sz_j$ time units to complete. We consider the path classes sorted in non-decreasing order of their slowdown factor: $sd_1<sd_2<\ldots<sd_P$. On each path, only one file can be transferred at one time. We consider file transfers to be non-preemptive.

We will present a polynomial-time algorithm for the makespan minimization problem, having a time complexity of $O(P \cdot C \cdot \log(Tmax))$, where $Tmax$ is an upper bound for the value of the makespan. In order to minimize the makespan, we will use binary search between $Tmin=sd_1 \cdot sz_1$ and

$$\mathrm{Tmax} = \sum_{i=1}^{C} \mathrm{sd}_1 \cdot \mathrm{sz}_i \cdot \mathrm{nf}_i \quad \textbf{(5-1)}$$

where a potential value T for the makespan will be selected. Then, the algorithm will perform a feasibility test, checking if all the transfers can be scheduled and completed on the P classes of paths within time T. If the values of the file sizes and slowdown factors are integers, then this algorithm finds an exact solution. If they are real numbers, then it will find a solution which is arbitrarily close to the optimum. In the rest of the section we will consider that all the values are integers, particularly because an improved version of the algorithm we will present later only works for integer values. We denote by A *div* B the integer division of A and B and by A *mod* B the remainder of the division of A and B. The function that tests if T is a feasible value for the makespan is based on a greedy strategy; its pseudocode is shown below:

For every path class i and file type j, we compute the values $q_{i,j}$ and $r_{i,j}$, defined like in the pseudocode above. It is obvious that $q_{i,1}$ represents the maximum number of type 1 files that can be transferred on a path of class i within the time T. In general, $q_{i,j}$ represents the maximum number of files of type j that can be transferred on a path of type i, considering that $q_{i,j'}$ files of each type $j'<j$ have already been transferred on that path. With these values, we will iteratively compute another set of values, $maxperiods_j$, representing the maximum number of time periods into which files of type j can be scheduled, considering that all the files of type $j'<j$ have already been scheduled on some paths. For each file type j, $maxperiods_j$ needs to be at least equal to nf_j.

The running time of the algorithm ***ChkT*** is obviously $O(P \cdot C)$. The algorithm can be implemented such that it uses only $O(P+C)$ memory, because the $q_{i,j}$ and $r_{i,j}$ values do not need to be stored after moving to the next class of paths $i+1$.

We will now present an improved feasibility test for the case $C=1$ when the sd_i values form an arithmetic or geometric progression, i.e. $sd_i=sd_{i-1}+K$ or $sd_i=sd_{i-1} \cdot K$, for $1<i\leq C$ and some known

constant K.

ChkT(T,C,sz_1,...,sz_C,nf_1,...,nf_C,P,sd_1,...,sd_P,np_1,…,np_P):
for $j=1$ **to** C **do** $qtotal_j=0$
for $i=1$ **to** P **do** {
  $q_{i,1}=T$ **div** $(sd_i \cdot sz_1)$
  $r_{i,1}=T$ **mod** $(sd_i \cdot sz_1)$
  $qtotal_1 = qtotal_1 + np_i \cdot q_{i,1}$
  for $j=2$ **to** C **do** {
    $q_{i,j}=r_{i,j-1}$ **div** $(sd_i \cdot sz_j)$
    $r_{i,j}=r_{i,j-1}$ **mod** $(sd_i \cdot sz_j)$
    $qtotal_j = qtotal_j + np_i \cdot q_{i,j}$
  }
}
$maxperiods_1=qtotal_1$
for $j=1$ **to** C **do** {
  if $(maxperiods_j<nf_j)$ **then return** *"no"*
  if $(j<C)$ **then** $maxperiods_{j+1}=(maxperiods_j-nf_j) \cdot sz_j / sz_{j+1}+qtotal_{j+1}$
}
return *"yes"*

Pseudocode 5-1. Greedy Feasibility Test.

ChkTC1(T,C=1,sz_1,nf_1,P,sd_1,…,sd_P,np_1,…,np_P,K):
  $G=0$; $firstCls=1$; $qtotal_1=0$; $sumnp_0=0$
  for $i=1$ **to** P **do** $sumnp_i=sumnp_{i-1}+np_i$
  while $(firstCls \le P)$ **do** {
    $q=T$ **div** $(sd_{firstCls} \cdot sz_1)$
    if $(q=0)$ **then break**
    $r=T$ **mod** $(sd_{firstCls} \cdot sz_1)$
    $dif=(r$ **div** $q)$ **div** sz_1

$$extraCls = \begin{cases} \left\lfloor \frac{dif}{K} \right\rfloor, & \text{for an arithmetic progression} \\ \left\lfloor \frac{\log\left(1+dif / sd_{firstCls}\right)}{\log(K)} \right\rfloor, & \text{for a geometric progression} \end{cases}$$

    $qtotal_1=qtotal_1+(sumnp_{firstCls+extraCls} - sumnp_{firstCls-1}) \cdot q$
    $firstCls=firstCls+extraCls+1$
  }
  if $(qtotal_1<nf_1)$ **then return** "no"
  else return "yes"

Pseudocode 5-2. Optimized Greedy Feasibility Test for C=1.

In this case, only the values $q_{i,1}$ will be computed. The P classes of paths can be split into G groups, such that: *1)* group *1* contains the classes $1,\ldots,nc_1$, group i $(1<i\le C)$ contains the classes $nc_{i-1}+1,\ldots,nc_i$; *2)* $nc_G=P$; *3)* for any two classes of paths i and j in the same group, we have $q_{i,1}=q_{j,1}$; *4)* for any two classes of paths i and j in different groups, we have $q_{i,1} \ne q_{j,1}$. G can be at most $2 \cdot sqrt(T)$ (where $sqrt(T)$ denotes the square root of T), because: there are at most $sqrt(T)$ classes i for which $(sd_i \cdot sz_1)$ is at most $sqrt(T)$ and each of these classes can form a group all by itself; all the other classes have $(sd_i \cdot sz_1) > sqrt(T)$ and for each such class i, the value $q_{i,1}$ is less than $sqrt(T)$, so there can only exist $sqrt(T)$ different possible values for $q_{i,1}$, thus forming at most $sqrt(T)$ groups. We also compute the values:

$$\text{sumnp}_i = \sum_{j=1}^{i} \text{np}_j \quad \textbf{(5-2)}$$

The feasibility test presented (and described in Pseudocode 5-2) above has *O(sqrt(T))* time complexity. If the sd_i values of the paths do not form an arithmetic or geometric progression, the algorithm can be changed so that it computes the value of *extraCls* by using a binary search (finding the largest class of paths *i* for which sd_i is not larger than $sd_{firstCls}+dif$), reaching a time complexity of *O(sqrt(T)·log(P))* for the test. There seems to be possible to extend the algorithm to the case *C>1*, but with rapid performance degradation ($O(T^{\frac{1}{2}+\frac{1}{4}+...+\frac{1}{2^C}})$ time complexity for the feasibility test).

5.2. High Multiplicity Scheduling of Two Communication Flows on Multiple Disjoint Packet-Type-Aware Paths

Communication performance in distributed systems may be rather poor when multiple communication flows use the network simultaneously. Because they are not aware of each other, they end up trying to use the same bottleneck resources, although other resources may be available in other places or at other times. The solution to this problem consists of scheduling the communication flows in such a way that a performance metric is optimized. In this section we are interested in optimally scheduling two communication flows from a sender to a receiver, using multiple disjoint paths. Each communication flow *i* is composed of a number of packets which need to be sent sequentially and have the same type *i* (*i=1,2*). The paths have different transmission times for the two packet types. This kind of situation may occur when the two communication flows belong to two distinct traffic classes, like, for instance, multimedia and normal web traffic. Moreover, some of the paths may be more suitable for one of the two traffic types.

We will present an algorithm for minimizing the makespan of the schedule, considering that the packet transmission is non-preemptive and that two packets cannot be in transit at the same time on the same path. The problem is given as a high multiplicity problem, but the algorithm is not necessarily polynomial in the input size. We will offer more details about this later.

5.2.1. Problem Statement

We consider *2* communication flows, composed of *np(i)* identical packets (*i=1,2*). The packets of the communication flow *i* are of type *i* and are identified by a pair of numbers *(i,j)*, $1 \le j \le np(i)$. The packets of the same type must be sent to the destination sequentially, using *P* disjoint paths. Each path *q* ($1 \le q \le P$) has *2* transmission times, *ts(q,1)* and *ts(q,2). ts(q, i)* (*i=1,2*) is the time taken by a packet of type *i* to reach the destination using path *q*. A schedule consists of assigning to each packet *(i,j)* a pair *(path(i,j), tstart(i,j))*, meaning that the packet is scheduled to be sent on path *path(i,j)*, starting from time *tstart(i,j)*. Based on this pair, we also associate with each packet *(i,j)* a time interval *[tstart(i,j), tfinish(i,j))*, where *tfinish(i,j)=tstart(i,j)+ ts(path(i,j),i)*. During this interval, the packet *(i,j)* is in transit on *path(i,j)*, so we will call it *transit interval*.

A schedule is valid if for any two packets *(i,j(1))* and *(i,j(2))*, *j(1)<j(2)*, we have $tfinish(i,j(1)) \le tstart(i,j(2))$, and if any two packets scheduled on the same path (disregarding their type) are assigned disjoint transit intervals. The first condition makes sure that each flow's packets are sent sequentially (in the order given by the starting time of the packets' transit intervals) and the second one makes sure that the packets scheduled on the same path are sent one at a time. The makespan C_{max} of the schedule is the maximum time at which a packet's transmission ends and we want to find a schedule which minimizes C_{max}:

$$C_{\max} = \max_{i=1,2} \{tfinish(i, np(i))\}. \quad \textbf{(5-3)}$$

We are interested in finding the schedule with the minimum makespan. The problem is a high multiplicity problem, because it has a very compact input, which makes it difficult to develop a polynomial time algorithm. For instance, an algorithm with time complexity $O(np_i)$ (*i=1,2*) would

be considered exponential. Unfortunately, the algorithm we will present has time complexity $O(np_i)$. Although the algorithm is quite efficient and developing it required many non-trivial proofs, it is not considered polynomial in the input size of the problem.

5.2.2. The Characteristics of Optimal Schedules

In this section we show that an optimal schedule (which minimizes the makespan) must have a particular structure, chosen from a small set of such structures. As a first step, we show that in an optimal schedule, each flow's packets are scheduled on at most *3* distinct paths. In order to do this, we will present and prove several theorems. The main technique lying at the basis of all the proofs is choosing an arbitrary valid schedule and changing it into a schedule which is not worse, but has all the properties mentioned by the theorem. For each flow i, we define an ordering of the paths: $po(i,1), po(i,2), \ldots, po(i,P)$, such that $ts(po(i,1),i) \leq ts(po(i,2),i) \leq \ldots \leq ts(po(i,P),i)$. In the proofs of the following theorems we will frequently reassign a packet from a path $po(i,q(1))$ to a path $po(i,q(2))$, with $q(2)<q(1)$. Such a reassignment does not change the starting time of the packet, but may decrease its ending time. The makespan of the schedule will not increase as a result of these operations.

Theorem 1. *Let k be the first position where the path orderings of the two flows differ, i.e. $po(1,q)=po(2,q)$, for $1 \leq q<k$ and $po(1,k) \neq po(2,k)$. If such a position exists, then in an optimal schedule, no packets are sent on any of the paths $po(i,q)$ $(i=1,2)$, with $q>k$.*

Proof. We will choose an arbitrary valid schedule. All the packets (i,j) which are assigned to paths $path(i,j)$ such that $path(i,j)=po(i,q)$, $q>k$, will be reassigned to path $po(i,k)$. After this reassignment, we obtain a new schedule. We will analyze the validity of this new schedule. The reassignment does not change the starting time of any packet, only the finish time, which may decrease. Therefore, the transit intervals of packets of the same type do not overlap. Let's see if the transit intervals of two packets scheduled on the same path might intersect. If the two packets are of the same type, we showed previously that this cannot happen, because their transit intervals are disjoint.

Let's assume that the transit intervals of two packets of different types, $(1,j(1))$ and $(2,j(2))$, scheduled on the same path p, intersect. This path cannot be one of the first $k-1$ paths (for any of the two flows), because no packet was reassigned to such a path. So path p must be the k^{th} path of one of the flows. W.l.o.g., we will assume that $p=po(1,k)$. But no type *2* packet is assigned to path $po(1,k)$, thus invalidating our initial assumption. In conclusion, the new schedule is valid and this holds for any valid schedule, including the optimal one.

Theorem 2. *In an optimal schedule, no packet (i,j) is sent on a path $po(i,q)$, with $q>4$.*

Proof. We will choose an arbitrary valid schedule, where at least one packet $(i(1),j)$ is scheduled on a path $po(i(1),q)$, with $q>4$ (using Theorem *1*, we also have $q \leq k$, if k exists). The packets of the other type $i(2)$ can be classified into *3* categories, according to the relationship between their transit interval and packet $(i(1),j)$'s transit interval:

- category *1*: their transit interval is included inside $[tstart(i(1),j), tfinish(i(1),j))$.
- category *2*: their transit interval intersects $[tstart(i(1),j), tfinish(i(1),j))$, but is not included in it.
- category *3*: their transit interval does not intersect $[tstart(i(1),j), tfinish(i(1),j))$.

All the category *1* packets can be reassigned to path $po(i(2),1)$, because no other packet of type $i(1)$ is scheduled on that path during the interval $[tstart(i(1),j), tfinish(i(1),j))$. The transit intervals of these packets do not increase. Category *2* packets cannot be reassigned to a different path, because they might be in conflict with other type $i(1)$ packets. However, it is easy to see that there can be at most two packets belonging to category *2*. One of the packets may cross $[tstart(i(1),j), tfinish(i(1),j))$ at the left end and the other one at the right end. Category *3* packets are of no interest. After performing the reassignment of category *1* packets, the packet $(i(1),j)$'s transit interval intersects with the transit intervals of packets scheduled on at most three distinct paths. So the packet $(i(1),j)$ can be reassigned to at least one of the paths $po(i(1),1)$, …, $po(i(1),4)$, without increasing the makespan and without breaking the validity of the schedule. By using this

reassignment method repeatedly, all the packets *(i(1),j)* assigned to some path *po(i(1),q)*, *q>4*, will be reassigned to some path *po(i(1), q(1))*, *1≤q(1)≤4*. Thus, any valid schedule can be turned into another schedule, where no packet is assigned to a path *po(i,q)*, *q>4*. This holds for an optimal schedule, too.

Theorem 3. *In an optimal schedule, no packet (i,j) is sent on a path po(i,q), with q>3.*

Proof. We will choose an arbitrary valid schedule. We first use Theorem *2* in order to obtain a valid schedule with a makespan which is not worse that the initial schedule and in which no packet is sent on a path *po(i,q)*, with *q>4*. Let's assume that a packet *(i(1),j)* is assigned to the path *po(i(1), 4)*. From Theorem *1*, we must have *k≥4*. We will classify the type *i(2)(≠i(1))* packets in the same three categories as in Theorem *2*'s proof and perform the same reassignments. If there are less than two packets in category *2* or if the two packets in category *2* are assigned to the same path or if one of them is assigned to path *po(i(2),1)* or to path *po(i(2),4)* then packet *(i(1),j)*'s transit interval intersects the transit intervals of packets scheduled on at most two distinct paths from the set *{po(i(1),1), po(i(1),2), po(i(1),3)}*, which allows us to reassign packet *(i(1),j)* to one of the paths in the set.

The more difficult case occurs when there are two packets belonging to category *2*, one of them assigned to the path *po(i(2),2)* and the other one to the path *po(i(2),3)* (see Fig. 5-1). In Fig. 5-1, w.l.o.g., we chose to place the type *i(2)* packet assigned to path *po(i(2),2)* on the left. Because each flow's schedule begins at time *0* and ends at time C_{max}, we can always choose to interpret time as moving from C_{max} towards *0*, so left and right are interchangeable. We will name *(i(2),j(2))* and *(i(2),j(3))* the two type *i(2)* packets assigned to paths *po(i(2),2)* and *po(i(2),3)*. All the type *i(1)* packets whose transit intervals start after the finish time of packet *(i(1),j)* and finish before the finish time of *(i(2),j(3))*, or finish before the starting time of *(i(1),j)* and start after the starting time of *(i(2),j(2))* can be reassigned to path *po(i(1), 1)*. Packet *(i(2),j(3))*'s transit interval must intersect with that of a packet of type *i(1)* assigned to path *po(i(1),2)*; otherwise, packet *(i(2),j(3))* could be reassigned to path *po(i(2),2)* and then packet *(i(1),j)* could be reassigned to path *po(i(1),3)*.

We define the interval $[t_1,t_2)$, where t_1 is the starting time of the first type *i(1)* packet (re)assigned to path *po(i(1),1)* whose transit interval is fully included inside that of packet *(i(2),j(2))* (or, if no such interval exists, the starting time of packet *(i(1),j))* and t_2 is the starting time of the first type *i(1)* packet assigned to *po(i(1),2)* whose transit interval intersects that of the packet *(i(2),j(3))*. We also define t_3 as the finish time of the transit interval of packet *(i(2), j(3))*. We define *l(i(1),1)* the total length of the transit intervals included in $[t_1,t_2)$ of all the type *i(1)* packets (re)assigned to path *po(i(1),1)* and *l(i(1),4)* the length of the transit interval of the packet *(i(1),j)*. Similarly, we define *l(i(2),1)* the total length of the transit intervals included in $[t_1,t_2)$ of all the type *i(2)* packets (re)assigned to path *po(i(2),1)*, *l(i(2),2)* the length of the transit interval of the packet *(i(2),j(2))* and *l(i(2),3)* the length of the transit interval of the packet *(i(2),j(3))*. All the packets whose transit intervals are included inside $[t_1,t_2)$ will be rearranged in such a way that the makespan will not increase and that it will be possible to reassign packet *(i(1),j)* to one of the paths *po(i(1),1),…,po(i(1),3)*.

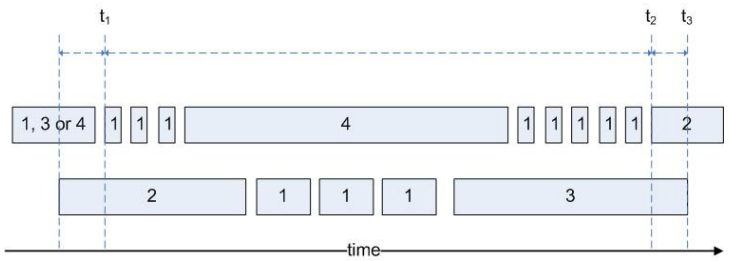


Fig. 5-1. Type i(1) Packets (First Row) and Type i(2) Packets (Second Row). The Left and Right Sides of the Packets are Aligned with Their Starting and Finish Time. The Numbers inside the Packets are the Positions of the Assigned Paths in the Corresponding Path Ordering.

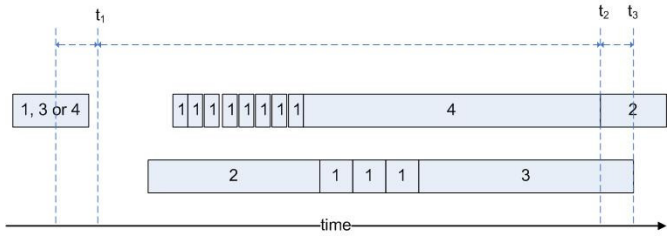


Fig. 5-2. The Case $l(i(2),3)+l(i(2),1)\leq t_3-t_2+l(i(1),4)$. The Rearrangement of Packets. No Path Reassignment has Been Performed, yet.

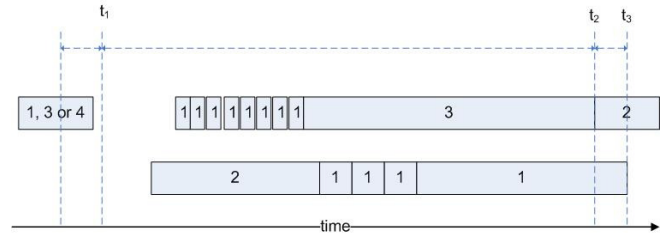


Fig. 5-3. The New Schedule in the Case $l(i(2),3)+l(i(2),1)\leq t_3-t_2+l(i(1),4)$. The Schedule is Valid and the Makespan did not Increase.

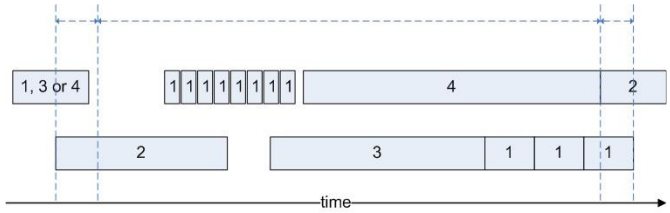


Fig. 5-4. The Case $l(i(2),1)+l(i(2),3)>t_3-t_2+l(i(1),4)$. The Rearrangement of Packets. No Path Reassignment has been Performed, yet.

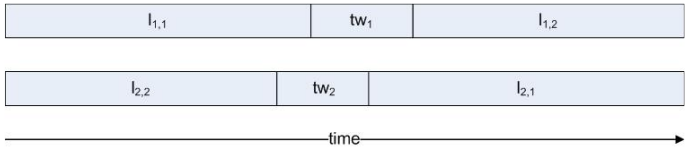


Fig. 5-5. Rearrangement of Packets for the Case k=2, P≥2.

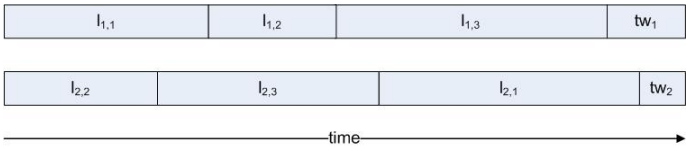


Fig. 5-6. The Case k=3, P≥4. Packet Rearrangement when $l_{1,1}+l_{1,2}\leq l_{2,2}+l_{2,3}$. No Path Reassignment has Been Performed, yet.

If $l(i(2),3)+l(i(2),1)\leq t_3-t_2+l(i(1),4)$, then the packets can be rearranged like in Fig. 5-2. Packet $(i(1),j)$ is placed such that its finish time is equal to t_2. Then, all the other type $i(1)$ packets whose transit intervals were included in $[t_1,t_2)$ will be placed somewhere inside the interval $[t_1,t_2-l(i(1),4))$. This is obviously possible, because $l(i(1),1)\leq t_2-t_1-l(i(1),4)$. After that, the packet $(i(2),j(3))$ will be reassigned to path $po(i(2),\ 1)$. This is now possible, because the only two packets whose transit intervals intersect the transit interval of packet $(i(2),j(3))$ are assigned to the paths $po(i(1),2)$ and $po(i(1),4)$. After packet $(i(2),j(3))$ is reassigned to path $po(i(1),1)$, packet $(i(1),j)$ can be reassigned to path $po(i(1),3)$. The final arrangement is shown in Fig. 5-3.

The case $l(i(2),3)+l(i(2),1)>t_3-t_2+l(i(1),4)$ is handled in a similar manner. The type $i(1)$ packets whose transit interval is included inside $[t_1,t_2)$ are placed like in the previous case. The type $i(2)$ packets (re)assigned to path $po(i(1),1)$ whose transit intervals are included inside $[t_1,t_2)$ are rescheduled consecutively, such that the last one finishes at time t_3. Because $l(i(2),1)<l(i(1),4)$ (initially, all of these packets' intervals were included inside packet $(i(1),j)$'s transit interval), the new starting time of the first of these packets, $t_{newstart}=t_3-l(i(2),1)$, will be greater than the new

starting time of the packet *(i(1),j)*. The packet *(i(2),j(3))* will be rescheduled such that its finish time is equal to $t_{newstart}$. Because $l(i(2),1)+l(i(2),3)>t_3-t_2+l(i(1),4)$, the starting time of the packet *(i(2),j(3))* will be smaller than the starting time of packet *(i(1),j)*. Fig. 5-4 shows the new arrangement.

The new transit interval of packet *(i(1),j)* intersects only type *i(2)* packets assigned to the paths *po(i(2),1)* and *po(i(2),3)*, so packet *(i(1),j)* can be reassigned to path *po(i(1),2)*. This was the last case to be considered. In every case, packet *(i(1),j)* could be reassigned to a path *po(i(1),q(1))*, with $1 \le q(1) \le 3$, without assigning any packet to a path located on a larger position in the corresponding path ordering and without increasing the makespan. Thus, any valid schedule (including an optimal one) can be changed into another valid schedule where no packet is assigned to a path *po(i,q)*, $q>3$.

We will characterize next all the cases of interest that may occur, according to the total number of paths *P* and the parameter *k* defined in Theorem *1*. We will use *A div B* to denote the integer division of *A* and *B*, i.e. the integer number *C*, such that $C \cdot B \le A < (C+1) \cdot B$. *A* and *B* do not necessarily have to be integer numbers.

5.2.2.1. k=1, P≥1

If $po(1,1) \neq po(2,1)$, then all the type *1* packets will be scheduled on path *po(1,1)* and all the type *2* packets on the path *po(2,1)*. The makespan will be:

$$C_{\max} = \max\{np(1) \cdot ts(po(1,1),1), np(2) \cdot ts(po(2,1),2)\}. \quad \textbf{(5-4)}$$

5.2.2.2. k does not exist, P=1

If *P=1* and *po(1,1)=po(2,1)*, then all the packets of both types will be scheduled on the first (and only) path. The makespan will be

$$C_{\max} = np(1) \cdot ts(1,1) + np(2) \cdot ts(1,2). \quad \textbf{(5-5)}$$

5.2.2.3. k=2, P ≥2

We choose an arbitrary valid schedule and denote its makespan by *C*. We denote by $l_{i,j}$ the total length of the transit intervals of type *i* packets scheduled on path *po(i,j)* ($1 \le j \le 2$) and by tw_i the total waiting time $tw_i = C - l_{i,1} - l_{i,2}$.

We have that $l_{1,1} \le l_{2,2} + tw_2$, because each transit interval of a type *1* packet scheduled on the path *po(1,1)* overlaps some part of a type *2* packet scheduled on path *po(2,2)* or some part of the waiting time tw_2. Because of this, the schedule can be changed such that all the type *1* packets assigned to path *po(1,1)* are scheduled first, followed by the waiting time tw_1 and then by all the type *1* packets scheduled on path *po(1,2)*. For the 2^{nd} flow, all the packets assigned to path *po(2,2)* are scheduled first, followed by the waiting time tw_2 and by the packets assigned to path *po(2,1)*. Fig. 5-5 presents the transformed schedule.

No transit interval of a type *1* packet scheduled on path *po(1,1)* overlaps with the transit interval of a type *2* packet scheduled on path *po(2,1)*. The schedule can be further refined by moving part of the waiting time tw_2 at the end and moving the type *2* packets assigned to path *po(2,1)* forward, so that their starting time is $max\{l_{1,1}, l_{2,2}\}$. Similarly, part of tw_1 can be moved at the end, so that type *1* packets assigned to path *po(1,2)* are sent starting from $max\{l_{1,1}, l_{2,2}\}$. Obviously, the new schedule is valid and its makespan is not larger than that of the original schedule.

An optimal schedule is properly defined by the number *u* of type *1* packets assigned to path *po(1,1)*. For the type *1* packets, the schedule can be written as $1^u, 2^{np(1)-u}$, meaning that the first *u* packets are assigned to path *po(1,1)* and the last *np(1)-u* packets are assigned to path *po(1,2)* (a term of the form a^b in the schedule of flow *i* represents *b* consecutive type *i* packets sent on path *po(i,a)*). If the number *u* of packets is fixed, the schedule for the type *2* packets has one of the following two forms:

- $2^v, tw, 1^{np(2)-v}$, where $v = min\{(l_{1,1} \text{ div } ts(po(2,2),2)), np(2)\}$ and $tw = l_{1,1} - v \cdot ts(po(2,2),2)$. This means that the first *v* packets of type *2* are sent consecutively on the path *po(2,2)*, then a waiting time

tw follows and then the last *np(2)-v* type *2* packets are sent on the path *po(2,1)*.

- $2^{v},1^{np(2)-v}$, where $v=min\{np(2),(l_{1,1}\ div\ ts(po(2,2),2))+1\}$

In order to find the optimal schedule, we need to find the value of *u* which minimizes the makespan.

5.2.2.4. k does not exist, P=2

This case is similar to the previous one. We will use the same notations as before. We have that $l_{1,1} \le l_{2,2}+tw_2$ and $l_{2,2} \le l_{1,1}+tw_1$ (by the same argument). Therefore, the schedule shown in Fig. 5-5 is valid in this case, too. Like in the previous case, an optimal schedule is properly defined by the number *u* of type *1* packets assigned to path *po(1,1)*. These packets will be sent first. In parallel, we will send as many type *2* packets as possible on path *po(2,2)*; we have two choices:

- send $v=min\{(l_{1,1}\ div\ ts(po(2,2),2)),np(2)\}$ type *2* packets on path *po(2,2)*, then wait a duration $tw=l_{1,1}-v\cdot ts(po(2,2),2)$ and then send the remaining type *2* packets on path *po(2,1)*. The type *1* packets assigned to path *po(1,2)* will be sent starting from time $l_{1,1}$. The schedule for the type *1* packets is $1^{u},2^{np(1)-u}$ and the schedule for type *2* packets is $2^{v},tw,1^{np(2)-v}$.
- send $v=min\{(l_{1,1}\ div\ ts(po(2,2),2))+1,np(2)\}$ type *2* packets on path *po(2,2)*, then send immediately the remaining type *2* packets on path *po(2,1)*. The type *1* packets assigned to path *po(1,2)* will have to wait a duration $tw=max\{v\cdot ts(po(2,2),2)-l_{1,1},0\}$ before starting to send them. The schedule for the type *1* packets is $1^{u},tw,2^{np(1)-u}$ and the one for type *2* packets is $2^{v},1^{np(2)-v}$. In this case, it would be better to choose the value of *v* and derive the value of *u* based on *v*.

Like in the previous case, finding the optimal schedule means finding the value of *u* which minimizes the makespan.

5.2.2.5. k=3, P≥4

If no packet *(i,j)* is assigned to the path *po(i,3)*, then this case is identical to the previous one. So we will restrict our attention to the case in which at least one packet *(i,j)* is assigned to the path *po(i,3)*. We will choose an arbitrary valid schedule with makespan *C*. We will define $l_{1,1}$, $l_{1,2}$, $l_{2,1}$, $l_{2,2}$ as before. Furthermore, we define $l_{i,3}$ the total length of the transit intervals of the type *i* packets assigned to path *po(i,3)* ($1 \le i \le 2$). The waiting times are now equal to $tw_i=C-l_{i,1}-l_{i,2}-l_{i,3}$. If $l_{1,1}+l_{1,2} \le l_{2,2}+l_{2,3}$, the packets can be rearranged like in Fig. 5-6 (temporarily, packets of both types sent on the path *po(1,2)=po(2,2)* may intersect). All type *1* packets assigned to path *po(1,1)* will be sent first, followed by all the type *1* packets assigned to path *po(1,2)* and by all the type *1* packets assigned to path *po(1,3)*. In parallel, we will send all the type *2* packets assigned to path *po(2,2)*, followed by all the type *2* packets assigned to path *po(2,3)* and then followed by those assigned to path *po(2,1)*. The waiting times are moved at the end of the schedule.

The type *1* packets assigned to path *po(1,2)* will be reassigned to path *po(1,1)*. The type *2* packets assigned to path *po(2,3)* will be reassigned to path *po(2,2)*. At this point, the type *1* packets are assigned only to the paths *po(1,1)* and *po(1,3)* and the type *2* packets are assigned only to the paths *po(2,1)* and *po(2,2)*. However, more reassignments are possible. All type *1* packets assigned to path *po(1,3)* whose finish time is smaller than or equal to $l_{2,2}+l_{2,3}$ can be reassigned to path *po(1,1)*. All type *1* packets assigned to path *po(1,3)* whose starting time is greater than or equal to $l_{2,2}+l_{2,3}$ can be reassigned to path *po(1,2)*. All these reassignments do not increase the lengths of the transit intervals, so they do not increase the makespan. In the end, there will be at most one type *1* packet assigned to path *po(1,3)* and no type *2* packet assigned to path *po(2,3)*. The schedule for the type *1* packets has the form $1^{u},3^{1},2^{np(1)-u-1}$ and the one for type *2* packets has the form $2^{v},1^{np(2)-v}$.

If $l_{1,1}+l_{1,2} > l_{2,2}+l_{2,3}$ and $l_{1,1} \ge l_{2,2}+l_{2,3}$, we can change the schedule in a similar manner. We will send the first type *1* packets assigned to path *po(1,1)*, followed by the type *1* packets assigned to path *po(1,2)* and then *po(1,3)*. In parallel, the type *2* packets assigned to path *po(2,2)* will be sent, followed immediately by the packets assigned to path *po(2,3)*. Because we have $l_{1,1} \le l_{2,2}+l_{2,3}+tw_2$ (since any transit interval of a type *1* packet assigned to path *po(1,1)* overlaps parts of transit intervals of type *2* packets assigned to paths *po(2,2)* or *po(2,3)*, or parts of tw_2), we can insert the waiting time tw_2 before sending the type *2* packets assigned to path *po(2,1)*. This way, the makespan does not increase and the schedule remains valid. Further reassignments are possible. All

type *2* packets assigned to path *po(2,3)* will be reassigned to path *po(2,2)* and all type *1* packets assigned to path *po(1,3)* will be reassigned to path *po(1,2)*. This way, no packet *(i,j)* remains assigned to the path *po(i,3)*, so we are in the case presented in the previous subsection. If $l_{1,1}+l_{1,2}>l_{2,2}+l_{2,3}$ and $l_{1,1}<l_{2,2}+l_{2,3}$, we need to make a difference between the following subcases: $l_{1,1}\geq l_{2,2}$ and $l_{1,1}<l_{2,2}$.

Subcase *1*: $l_{1,1}\geq l_{2,2}$. The packets will be rearranged the same way as before: for type *1* - the packets assigned to the path *po(1,1)*, then those assigned to path *po(1,2)* and then those assigned to path *po(1,3)*; for type *2* - the packets assigned to path *po(2,2)*, then those assigned to path *po(2,3)* and then those assigned to path *po(2,1)*. Because $l_{1,1}\geq l_{2,2}$, the transit interval of no type *1* packet assigned to paths *po(1,2)* or *po(1,3)* overlaps the transit interval of a type *2* packet assigned to path *po(2,2)*. Thus, all the type *1* packets assigned to path *po(1,3)* can be reassigned to path *po(1,2)* and the schedule is valid. The type *2* packets assigned to path *po(2,3)* whose finish time is smaller than or equal to $l_{1,1}$ will be reassigned to path *po(2,2)* and those whose starting time is greater than or equal to $l_{1,1}$, will be reassigned to path *po(2,1)*. This leaves at most one type *2* packet still assigned to path *po(2,3)*. The schedule for the type *1* packets has the form $1^u,2^{np(1)-u}$ and the one for type *2* packets has the form $2^v,3^1,1^{np(2)-v-1}$.

Subcase 2: $l_{1,1}<l_{2,2}$. The type *2* packets will be rearranged just like in the previous subcase. Furthermore, all the type *2* packets assigned to path *po(2,3)* will be reassigned to path *po(2,1)*. The type *1* packets assigned to path *po(1,1)* will be sent first, followed by the type *1* packets assigned to path *po(1,3)*. Because $l_{1,1}+l_{1,3}+tw_1\geq l_{2,2}$, we can insert the waiting time tw_1 before sending the type *1* packets assigned to path *po(1,2)*. After the reassignments, the schedule is valid and its makespan did not increase. Furthermore, the type *1* packets assigned to path *po(1,3)* whose finish time is smaller than or equal to $l_{2,2}$ will be reassigned to path *po(1,1)* and those whose starting time is greater than or equal to $l_{2,2}$ will be reassigned to path *po(1,2)*, leaving at most one type *1* packet still assigned to path *po(1,3)*. The schedule for the type *1* packets has the form $1^u,3^1,2^{np(1)-u}$ or $1^u,tw,2^{np(1)-u}$ and the one for type *2* packets has the form $2^v,1^{np(2)-v}$.

5.2.2.6. k does not exist, P=3

Any valid schedule for this case is also a valid schedule for the previous one. Therefore, we can use the same arguments and transformations. The only problem we might encounter is that the schedule obtained after performing the transformations of the previous case might contain two packets *(1,j(1))* and *(2,j(2))*, with overlapping transit intervals and assigned to the same path *po(1,3)=po(2,3)*. However, we can see that this is not the case, because any schedule obtained in the previous case contains at most one packet *(i,j)* assigned to a path *po(i,3)* (either *po(1,3)* or *po(2,3)*).

5.2.2.7. k>3 or non-existent, P>3

According to Theorem *3*, no packet *(i,j)* is sent on a path *po(i,q)*, *q>3*. Thus, we can limit the value of *P* to *3* and the case becomes identical to the previous one.

In this section we characterized the structure of optimal schedules. There are five kinds of non-trivial structures:

- $1^u,2^{np(1)-u}$ for flow *1* and $2^v,tw,1^{np(2)-v}$ for flow *2*, where $v=min\{np(2), ((ts(po(1,1),1)\cdot u)\ div\ ts(po(2,2),2))\}$ and $tw=u\cdot ts(po(1,1),1)-v\cdot ts(po(2,2),2)$.
- $1^u,tw,2^{np(1)-u}$ for flow *1* and $2^v,1^{np(2)-v}$ for flow *2*, where $u=min\{np(1), ((ts(po(2,2),2)\cdot v)\ div\ ts(po(1,1),1))\}$ and $tw=v\cdot ts(po(2,2),2)-u\cdot ts(po(1,1),1)$.
- $1^u,2^{np(1)-u}$ for flow *1* and $2^{v+1},1^{np(2)-v-1}$ for flow *2*, where $v=min\{np(2)-1, ((ts(po(1,1),1)\cdot u)\ div\ ts(po(2,2),2))\}$.
- $1^u,2^{np(1)-u}$ for flow *1* and $2^v,3^1,1^{np(2)-v-1}$ for flow *2*, where $v=min\{np(2)-1,((ts(po(1,1),1)\cdot u)\ div\ ts(po(2,2),2))\}$.
- $1^u,3^1,2^{np(1)-u-1}$ for flow *1* and $2^v,1^{np(2)-v}$ for flow *2*, where $u=min\{np(1)-1,((ts(po(2,2),2)\cdot v)\ div\ ts(po(1,1),1))\}$.

5.2.3. An $O(np_i)$ Scheduling Algorithm

We will present an algorithm with time complexity $O(np(i))$ which determines the optimal schedule for any of the five kinds of non-trivial structures presented in the previous section. The algorithm has time complexity $O(log(np(i)))$ on three of the schedule structures, but two structures are more difficult and we were unable to develop an equally efficient algorithm for them. We will not include in this section the trivial cases $k=1$ and $P=1$, which can easily be solved in $O(1)$ time using equations (2) and (3).

5.2.3.1. Case 1: $1^u,2^{np(1)-u}$ and $2^v,3^1,1^{np(2)-v-1}$

We chose to handle first the case $1^u,2^{np(1)-u}$ and $2^v,3^1,1^{np(2)-v-1}$, because it is easier to solve than the cases where waiting times are involved. We will define two functions, $C_1(u)$ and $C_2(u)$ representing the completion time of flow *1* and flow *2*, respectively, if there are u packets of type *1* assigned to the path $po(1,1)$. Their definitions are:

$$C_1(u)=u\cdot ts(po(1,1),1)+(np(1)-u)\cdot ts(po(1,2),1) \quad \textbf{(5-6)}$$

$$C_2(u)=v\cdot ts(po(2,2),2)+ts(po(2,3),2)+(np(2)-v-1)\cdot ts(po(2,1),2) \text{ , with} \quad \textbf{(5-7)}$$
$$v=min\{(ts(po(1,1),1)\cdot u)\ div\ ts(po(2,2),2)\ ,\ np(2)-1\}$$

The first function is decreasing for $u \in [0, np(1)]$. The difference $C_1(x+1)-C_1(x)=ts(po(1,1),1)-ts(po(1,2),1)$ is constant. The values of the second function are increasing, but not necessarily strictly increasing. This is easily noticeable, because as u increases, so does v. Whenever v increases, the number of packets assigned to path $po(2,2)$ increases and the number of packets assigned to path $po(2,1)$ decreases, so the overall value increases. In order to find the value $u_{opt} \in [0, np(1)]$ for which $max\{C_1(u_{opt}), C_2(u_{opt})\}$ is minimum we have the following three subcases:

- *Subcase 1*: $C_1(0) \le C_2(0)$. In this case, $max\{C_1(u), C_2(u)\}=C_2(u)$ and the minimum value of $C_2(u)$ is $C_2(0)$. So $u_{opt}=0$.
- *Subcase 2*: $C_1(np(1)) \ge C_2(np(1))$. In this case, $max\{C_1(u), C_2(u)\}=C_1(u)$ and the minimum value of $C_1(u)$ is $C_1(np(1))$. So $u_{opt}=np(1)$.
- *Subcase 3*: $C_1(w) \ge C_2(w)$, for $0 \le w \le w_m$ and $C_1(w) < C_2(w)$ for $w_m < w \le np(1)$. We can find the value of w_m using binary search. The value of u_{opt} is either w_m or w_m+1.

The time complexity of the algorithm is $O(log(np_i))$. The cases $((1^u,3^1,2^{np(1)-u-1}), (2^v,1^{np(2)-v}))$ and $((1^u,2^{np(1)-u}), (2^{v+1},1^{np(2)-v-1}))$ are similar. We define the two functions $C_1(v)$ and $C_2(v)$ having the same meaning, which are decreasing, respectively strictly increasing. We have the same three situations and use binary search to find the optimal value u_{opt} in the third situation.

5.2.3.2. Case 2: $1^u,2^{np(1)-u}$ and $2^v,tw,1^{np(2)-v}$

We will define the two functions $C_1(u)$ and $C_2(u)$, representing the completion time of the first, respectively, second flow, if u packets of type *1* are assigned to path $po(1,1)$. C_1 is defined as before, while C_2's definition is:

$$C_2(u)=u\cdot ts(po(1,1),1)+(np(2)-v)\cdot ts(po(2,1),2) \text{ , with} \quad \textbf{(5-8)}$$
$$v=min\{(ts(po(1,1),1)\cdot u)\ div\ ts(po(2,2),2),\ np(2)\}$$

This case is more difficult, because although C_1 is strictly decreasing, C_2's values are not increasing. The only algorithm we could find was to try out all the $np(1)$ possible values of u and choose the one which minimizes the makespan. A similar situation occurs for the case $((1^u,tw,2^{np(1)-u}), (2^v,1^{np(2)-v}))$.

5.3. Minimum Weighted Sum of Completion Times and Minimum Makespan Scheduling of Two Communication Flows with Packet Ordering Constraints

In this section we consider two communication flows, which need to send $np(1)$ and, respectively, $np(2)$ packets from the same source to the same destination using some of the P

disjoint paths available. The packets are not necessarily identical and their order within the same flow is fixed. For each packet *(i,j)*, the path on which it must be sent is already defined. We will present dynamic programming algorithms for minimizing the following two objectives: (o_1) minimum sum of completion times (see eq. (5-9)) and (o_2) minimum weighted makespan (see eq. (5-10)). Each flow *i* has a weight *w(i)* (*i=1,2*). We will use the same notations as in the previous section. What will be different is that we will have *ts(q,i,j)*=the duration for sending the j^{th} packet of flow *i* on the path *q* (instead of only *ts(q,i)* for the case when all the packets of the same flow were identical).

$$ST = w(1) \cdot tfinish(1, np(1)) + w(2) \cdot tfinish(2, np(2)) . \quad \textbf{(5-9)}$$

$$ST = \max\{w(1) \cdot tfinish(1, np(1)), w(2) \cdot tfinish(2, np(2))\} . \quad \textbf{(5-10)}$$

The minimum (weighted) sum of completion times is at least equal to:

$$ST_{low} = w(1) \cdot \sum_{j=1}^{np(1)} ts(path(1, j),1) + w(2) \cdot \sum_{j=1}^{np(2)} ts(path(2, j),2) . \quad \textbf{(5-11)}$$

Min-WST-WMakeSpan-Case1:

```
ST=+∞ ; initialize T_wm(*,*)
(o1) compute ST_low
for a=0 to np(1) do {
 for b=0 to np(2) do {
  if (T_wm(a,b)<+∞) then {
   a'=a+1 ; b'=b+1 ; tsa'=0 ; tsb'=0
   while ((a'≤np(1)) and (b'≤np(2))) do {
    if (path(1,a')=path(2,b')) then break
    if (tsa'+ts(path(1,a'),1,a')<tsb'+ts(path(2,b'),2,b')) then {
     tsa'= tsa'+ts(path(1,a'),1,a') ; a'=a'+1
    } else if (tsa'+ts(path(1,a'),1,a')>tsb'+ts(path(2,b'),2,b')) then {
     tsb'=tsb'+ts(path(2,b'),2,b') ; b'=b'+1
    } else {
     tsa'= tsa'+ts(path(1,a'),1,a') ; a'=a'+1
     tsb'=tsb'+ts(path(2,b'),2,b') ; b'=b'+1
    }
   }
   if ((a'>np(1)) or (b'>np(2))) then {
    while (a'≤np(1)) do {
     tsa'= tsa'+ts(path(1,a'),1,a')
     a'=a'+1
    }
    while (b'≤np(2)) do {
     tsb'=tsb'+ts(path(2,b'),2,b')
     b'=b'+1
    }
    (o1) ST=min{ST,ST_low+T_wm(a,b)}
    (o2) ST=min{ST, max{w(1)·(T_wm(a,b)+tsa'), w(2)·(T_wm(a,b)+tsb')}}
   } else {
    (o1) T_wm(a'-1,b')=min{ T_wm(a'-1,b'), T_wm(a,b)+w(1)·(tsb'+ts(path(2,b'),2,b')-tsa')}
    (o1) T_wm(a',b'-1)=min{ T_wm(a',b'-1), T_wm(a,b)+w(2)·(tsa'+ts(path(1,a'),1,a')-tsb')}
    (o2) T_wm(a'-1,b')=min{ T_wm(a'-1,b'), T_wm(a,b)+(tsb'+ts(path(2,b'),2,b')}
    (o2) T_wm(a',b'-1)=min{ T_wm(a',b'-1), T_wm(a,b)+tsa'+ts(path(1,a'),1,a')}
   }
  }
 }
}
```

Pseudocode 5-3. Optimization Algorithm – Case 1.

In this case, all we need to do is minimize the total weighted waiting time of the packets - caused by pairs of packets *(1,j(1))* and *(2,j(2))* scheduled on the same path and whose transit intervals might overlap. We will compute a table $T_{wm}(a,b)$=the minimum total weighted waiting time required for sending the first *a* packets of the first flow, the first *b* packets of the second flow and the packets *(1,a+1)* and *(2,b+1)* are scheduled to be sent at the same time moment. Initially, we have $T_{wm}(0,0)=0$ and $T_{wm}(a,b)=+\infty$ (for $a>0$ or $b>0$). We will use a forward type of dynamic programming. The pairs *(a,b)* ($0\leq a\leq np(1)$, $0\leq b\leq np(2)$) will be traversed in lexicographic order. If $T_{wm}(a,b)<+\infty$, then we will perform the following actions: we will advance forward in time, until all the packets of one of the two flows are sent or until a conflict occurs (packets $a'>a$ and $b'>b$ are scheduled on the same path and during overlapping time intervals). In the first situation, we will consider updating the minimum weighted sum of completion times by the value $ST_{low}+T_{wait}(a,b)$. In the second case, we will update the values $T_{wm}(a'-1,b')$ and $T_{wm}(a',b'-1)$.

In the minimum weighted makespan case, we will compute a similar dynamic programming table: $T_{wm}(a,b)$=the minimum time duration required for the packets *(1,a+1)* and *(2,b+1)* to be scheduled for being sent at the same time moment. $T_{wm}(*,*)$ is initialized like in the previous case.

In Pseudocode 5-3 and 5-4, the lines prefixed with (o_1) are executed only when the objective is the minimum sum of weighted completion times, while those with (o_2) are executed only when the objective is the minimum weighted makespan.

The time complexity of the algorithm in Pseudocode 5-3 is $O(np(1)\cdot np(2)\cdot(np(1)+np(2)))$. Note that in this case we only need the $ts(*,*,*)$ values for the tuples $(path(i,j), i, j)$ (i.e. only $np(1)+np(2)$ values). In fact, the values $ts(path(i,j), i, j)$ can be replaced by the $ts'(i,j)$ values.

We will now consider the case when only the packets of the first flow have fixed paths, but we are free to choose the paths on which the packets of the second flow should be sent. The ordering constraints still need to be obeyed. We will use dynamic programming in a similar manner to the previous case and compute a table $T_{min}(a,b)$=the minimum total weighted time (or simply time in the case of the weighted makespan) required for sending all the packets of the first flow, the first *b* packets of the second flow and the packets *(1,a+1)* and *(2,b+1)* are scheduled to be sent at the same time. We have:

$$T_{min}(0,0)=w(1)\cdot(ts(path(1,1),1,1)+ts(path(1,2),1,2)+\ldots+ts(path(1,np(1)),1,np(1))) \quad \textbf{(5-12)}$$

for the weighted sum case and $T_{min}(0,0)=0$ for the weighted makespan case. For the other pairs *(a,b)*, we initially have $T_{min}(a,b)=+\infty$. For each pair *(a,b)* with $T_{min}(a,b)<+\infty$, in lexicographic order, we will generate the longest possible schedule (up to (a',b'), $a'\geq a$, $b'\geq b$), by scheduling the next type *2* packet on the fastest path which does not generate any conflicts. At each step, we also consider scheduling the type *2* packets on faster paths, thus obtaining conflicts – in this case, we try to update values like $T_{min}(a'',b'-1)$ and $T_{min}(a''-1,b')$, $a''\geq a'$. We will denote by $po(2,k,j)$ the path *q* for which $ts(q,2,j)$ is the k^{th} smallest among all the values $ts(*,2,j)$ (i.e. for each pair *(2,j)* we sort the paths *q* in increasing order of their $ts(q,2,j)$ values). The algorithm is presented in Pseudocode 5-4.

The time complexity of the algorithm (Pseudocode 5-4) is $O(np(1)\cdot np(2)\cdot(np(1)+np(2))\cdot Pmax\cdot np(1))$. Running the algorithm with $P_{max}=P$ is correct, but, for the case when all the packets are identical (i.e. ts(q,i,*)=ts(q,i)), we conjecture that we can run it with $P_{max}=min\{P, 4\}$ and it will still always compute an optimal solution.

5.4. Minimum Makespan Packet Scheduling over Multiple Disjoint Paths with Connection Initiation Times

We consider a communication flow which consists of *N* identical packets (which do not necessarily have to be sent in order). The packets must be sent from the source to the destination, using some of the *P* disjoint paths available. Each path *i* ($1\leq i\leq P$) has three parameters: a connection initiation time $CI(i)\geq 0$, a packet sending time $PS(i)\geq 0$ and the maximum number of packets $pmax(i)\geq 1$ that can be sent along the path. If we want to send *k* ($1\leq k\leq pmax(i)$) packets on path *i*, this will take $CI(i)+k\cdot PS(i)$ time units (for $k=0$, it takes *0* time units). We want to distribute the packets over the *P* paths, such that the maximum time moment at which a packet arrives at the destination is minimum (i.e. we want to minimize the makespan of the packet distribution strategy).

```
Min-WST-WMakespan-Case2(P_max(≤P)):
ST=+∞;
(o1) T_min(0,0)=w(1)·(ts(path(1,1),1,1)+...+ts(path(1,np(1)),1,np(1)))
(o2) T_min(0,0)=0
for a=0 to np(1) do {
 for b=0 to np(2) do {
  if (T_min(a,b)<+ ∞) then {
   a'=a+1 ; b'=b+1 ; tsa'=0 ; tsb'=0
   while ((a'≤np(1)) and (b'≤np(2))) do {
    bestp=uninitialized
    for pa=1 to P_max do {
     a''=a'; tsa''=tsa'; noconflict=true
     while ((a''≤np(1)) and (tsa''<tsb'+ts(po(2,pa,b'),2,b')) do {
      if (path(1,a'')=pa) then {
       (o1) T_min(a'',b'-1)=min{T_min(a'',b'-1), T_min(a,b)+w(2)·(tsa''+ts(path(1,a''),1,a''))}
       (o1) T_min(a''-1,b')=min{T_min(a''-1,b'), T_min(a,b)+w(1)·
                          (tsb'+ts(po(2,pa,b'),2,b')-tsa'')+w(2)·(tsb'+ts(po(2,pa,b'),2,b'))}
       (o2) T_min(a'',b'-1)=min{T_min(a'',b'-1), T_min(a,b)+tsa''+ts(path(1,a''),1,a'')}
       (o2) T_min(a''-1,b')=min{T_min(a''-1,b'), T_min(a,b)+tsb'+ts(po(2,pa,b'),2,b')}
       noconflict=false; break
      } else {
       tsa''=tsa''+ts(path(1,a''),1,a'') ; a''=a''+1
      }
     }
     if ((bestp=uninitialized) and (noconflict=true)) then bestp=pa
    }
    if (bestp≠unintialized) then {
     tsb'=tsb'+ts(po(2,bestp,b'),2,b') ; b'=b'+1
     while ((a'≤np(1)) and (tsa'+ts(path(1,a'),1,a')≤tsb')) do {
      tsa'=tsa'+ts(path(1,a'),1,a') ; a'=a'+1
     }
    } else break
   }
   if (b'>np(2)) then {
    while (a'≤np(1)) do {
     tsa'= tsa'+ts(path(1,a'),1,a') ; a'=a'+1
    }
   }
   if (a'>np(1)) then {
    while (b'≤np(2)) do {
     tsb'=tsb'+ts(po(2,1,b'),2,b') ; b'=b'+1
    }
   }
   if (a'>np(1)) then {
    (o1) ST=min{ST, T_min(a,b)+w(2)·tsb'}
    (o2) ST=min{ST, max{w(1)·(T_min(a,b)+tsa'), w(2)·(T_min(a,b)+tsb')}}
   }
  }
 }
}
```

Pseudocode 5-4. Optimization Algorithm – Case 2.

We will first present a solution with $O(N \cdot log(P))$ time complexity. We will maintain a min-

heap H in which we insert every path i, with an initial key $Key(i)=CI(i)$. We will also maintain a counter $pkt(i)$ for every path i (initially, $pkt(i)=0$ for every path $1 \le i \le P$). We will repeatedly extract from H the minimum key N times. Let's assume that we extracted the value $Key(i)$, assigned to path i. We will send a packet on path i at time moment $Key(i)$, which will reach the destination at time $Key(i)+PS(i)$. Then, we remove $Key(i)$ from H, set $Key(i)=Key(i)+PS(i)$ and increment $pkt(i)$ by 1. If $pkt(i)<pmax(i)$ then we will re-insert $Key(i)$ (the new value) into H (after every insertion, we also maintain the path i associated to every value $Key(i)$).

We now consider the restriction that only at most Q of the P available paths can be used for sending the packets. This case is identical to the previous one when $Q=P$. We will present a solution with $O(sort(P) \cdot log(TMAX))$ time complexity, where $TMAX$ is the maximum possible value of the makespan and $sort(P)$ is the time complexity to sort P numbers. We will binary search the makespan. Let's assume that we selected a value T within the binary search. For each path i ($1 \le i \le P$) we will compute a value $np(i)$=the number of packets which can be sent on path i using at most T time units: if $CI(i)>T$ then $np(i)=0$; otherwise, $np(i)=min\{(T-CI(i)) \text{ div } PS(i), pmax(i)\}$ (A div B denotes the integer division of A at B). We then sort the paths such that $np(path(1)) \ge np(path(2)) \ge \ldots \ge np(path(P))$, where $path(1), \ldots, path(P)$ is a permutation of the P paths. Out of these, we will select the first Q paths and compute $sumnp=np(path(1))+\ldots+np(path(Q))$. If $sumnp \ge N$ then T is a feasible value for the makespan and, thus, we will test a smaller value next in the binary search; if, however, $sumnp<N$, then T is too small and we need to test a larger value next in the binary search. $sort(P)$ may be $O(P \cdot log(P))$, or, if the values $np(i)$ are small integer numbers, then $sort(P)$ may be $O(P+VMAX)$, where $VMAX$ is the largest possible value of $np(i)$ (we can sort these values by using a procedure similar to count sort).

5.5. Optimal Offline TCP Sender Buffer Management Strategy

TCP uses a sliding window mechanism in order to enforce flow control and not overwhelm the receiver with too much data. This mechanism is particularly useful when a fast, powerful sender communicates with a slow receiver or with one having limited resources (small amounts of buffer space). However, these situations are rather stressful for the sender, which needs to bring data from application buffers into TCP buffers many times. If the sender is loaded by many applications performing different tasks, copying data between buffers might not always take the same amount of time. Because of this, it might happen that the sender becomes a performance bottleneck, too, not only the receiver, as we would have expected.

In this section we propose a model for characterizing the sender's behavior throughout the life time of a TCP connection. Based on this model, we developed an $O(n \cdot log^2(n))$ algorithm for computing the minimum total processing time for the sender when the window sizes advertised by the receiver and the system load of the sender are known in advance. The algorithm can only be used offline, either when accurate estimates of the required parameters are known, or after a TCP conversation took place, in order to find out what the minimum processing time could have been.

5.5.1. TCP Sender Behavior Model

After the initial three-way handshake, the TCP sender will attempt to send as much data as the last window size advertised by the receiver. We will assume that throughout the TCP conversation, the receiver advertised his window size n times. The window size at the i^{th} advertisement is w_i. After each advertisement i is received, the sender transmits the next w_i bytes of data to the receiver and waits for the next window size advertisement. We will assume that the total amount of transmitted bytes TB is equal to the sum of the windows sizes:

$$TB = \sum_{i=1}^{n} w_i \qquad \textbf{(5-13)}$$

When receiving the i^{th} advertisement, the sender already has in its TCP buffer the next bb_i bytes to be sent. If $bb_i<w_i$, the sender will copy cb_i bytes ($w_i-bb_i \le cb_i$) into its TCP buffer and then send w_i bytes to the receiver. The sender may choose to copy some bytes into the TCP buffer even

when $bb_i \geq w_i$. The time needed to copy x bytes into its TCP buffer at the time of the i^{th} advertisement is

$$t_{copy,i}(x)=t_{setup,i} + t_{byte,i} \cdot x \ . \quad \textbf{(5-14)}$$

There are two components comprised in the copy time. The first one ($t_{setup,i}$) is the setup time needed to initiate the byte transfer. The second one ($t_{byte,i}$) is the time required for transferring a byte from the application buffer into the TCP buffer. If no byte is copied at the time of the i^{th} advertisement, then the copy time is *0*. The two time parameters ($t_{setup,i}$ and $t_{byte,i}$) depend on the system load and may change at any time. We will call the moment that a window size advertisement is received a *time step*. Given the values w_i, $t_{setup,i}$ and $t_{byte,i}$ for each of the n time steps, the processing time depends on the number of bytes cb_i copied between the application buffer and the TCP buffer during each step.

In our model, we will assume that the TCP buffer capacity is very large (infinite) compared to the total number of bytes transferred. That is, it would be possible for the sender to copy all the bytes in the TCP buffer during the first time step, if such a strategy is considered the most convenient. Although buffer space may be large enough for the data transferred on a single TCP connection, we must consider the fact that this buffer might be shared by several connections.

We will take this into consideration by adding another parameter sc_i to every time step. This parameter represents the cost of storing one byte in the TCP buffer from time step i to time step $i+1$. This parameter will also be expressed in time units, because using up one byte of the TCP buffer increases the processing times of other TCP connections, which will have less buffer space at their disposal. Thus, this parameter represents the amount by which the processing times of other TCP conversations increases if one byte is stored in the TCP buffer from one time step i to the next. Under these conditions, the total processing time of the sender is

$$TPT = \sum_{i=1}^{n} t_{copy,i}(cb_i) + \sum_{i=2}^{n} b_i \cdot sc_i \ . \quad \textbf{(5-15)}$$

Given the values of all the parameters at each time step, the total processing time depends only on the number of bytes cb_i copied during each time step i. The values cb_i for which the value of *TPT* is minimum define an optimal sender buffer management strategy. It is interesting that, with other interpretations given to the problem parameters, this problem is equivalent to the economic lot sizing problem, for which optimal $O(n \cdot log(n))$ algorithms are known. In the following section we will present a novel $O(n \cdot log^2(n))$ algorithm which, although it does not have an optimal complexity, it has the advantage of not being difficult to implement.

5.5.2. An Efficient Algorithm For the Minimum Total Processing Time

The algorithm uses a dynamic programming approach and a geometric data structure called segment tree. The segment tree is used in a novel way, for dynamically maintaining half-lines having different slopes and for computing the half-line having the smallest y-coordinate for a given x-coordinate.

Solving the problem starts with an observation which is not necessarily obvious. Let's assume that at time step j, the TCP buffer contains X bytes, after copying the cb_j bytes planned for that time step ($X=b_j+cb_j$). Let's also assume that among these X bytes, X_1 ($X_1>0$) were copied in the buffer at time step i_1 and X_2 ($X_2>0$) were copied in the buffer at time step i_2. The processing time incurred so far by the X_1 (X_2) bytes is:

$$PT_1=t_{setup,i2}+t_{byte,i1} \cdot X_1+(sc_{i1} + sc_{i1+1}+...+sc_{j-1}) \cdot X_1=A_1+B_1 \cdot X_1, \ A_1>0, \ B_1>0 \quad \textbf{(5-16)}$$

$$PT_2=t_{setup,i2} + t_{byte,i2} \cdot X_2+(sc_{i2}+sc_{i2+1}+\ldots+sc_{j-1}) \cdot X_2=A_2 \cdot X_2+B_2, \ A_2>0, \ B_2>0 \quad \textbf{(5-17)}$$

The processing time incurred by the X_1+X_2 bytes is $PT_{12}=PT_1+PT_2$. If all the X_1+X_2 bytes had been copied in the TCP buffer at time step i_1, the processing time would have been

$$PT_A=A_1+B_1 \cdot (X_1+X_2). \quad \textbf{(5-18)}$$

Similarly, if all the X_1+X_2 bytes had been copied in the TCP buffer at time step i_2, the processing time would have been

$$PT_B=A_2+B_2 \cdot (X_1+X_2). \quad \textbf{(5-19)}$$

We will show that either PT_A or PT_B must be less than or equal to PT_{12}. We have

$$PT_A-PT_{12}=B_1\cdot X_2-B_2\cdot X_2-A_2=(B_1-B_2)\cdot X_2-A_2. \quad \textbf{(5-20)}$$

In a similar manner,

$$PT_B-PT_{12}=(B_2-B_1)\cdot X_1-A_1. \quad \textbf{(5-21)}$$

It is obvious that either $B_1-B_2\leq 0$ or $B_2-B_1\leq 0$. If the first one is true, then we have $PT_A\leq PT_{12}$. If the second one is true, then we have $PT_B\leq PT_{12}$. Thus, in an optimal solution, the time steps *1, …, n* can be split in a number *K* of intervals $[1=l_1,r_1]$, $[l_2=r_1+1, r2]$, …, $[l_K=r_{K-1}+1, r_K=N]$, such that at every time step l_i, the number of bytes copied in the TCP buffer is:

$$cb_{l_i}=\sum_{j=l_i}^{r_i} w_j \quad \textbf{(5-22)}$$

At the time steps *j* which are not the first time steps of some interval, no bytes will be copied in the TCP buffer and the required bytes will already be there. We will present three dynamic programming solutions for the problems, each one improving upon the previous one. The last solution is the $O(n\cdot log^2 n)$ algorithm we mentioned.

5.5.2.1. An $O(n^3)$ Dynamic Programming Algorithm

We will compute an array *mintpt*, where *minpt[i]* is the minimum total processing time if i were the last time step. The pseudocode is given below:

```
mintpt[0]=0
for i=1 to n do {
  mintpt[i]=+∞
  for j=1 to i do {
    totalpt=mintpt[j-1]+t_setup,j+t_byte,j·w_j
    storagept=sc_j
    for k=j+1 to i do {
      totalpt=totalpt+(t_byte,j+storagept)·w_k
      storagept=storagept+sc_k
    }
    if (totalpt<mintpt[i]) then mintpt[i]=totalpt
  }
}
```

Pseudocode 5-5. An $O(n^3)$ Dynamic Programming Algorithm.

The algorithm is straightforward. For each time step *i*, considering that *i* is the last time step, all possible time steps *j* are considered, such that *j* is the first time step of the interval ending at *i*. For each value of *j*, the total processing time is computed and the minimum value is maintained.

5.5.2.2. An $O(n^2)$ Dynamic Programming Algorithm

```
mintpt[0]=0
for i=1 to n do {
  mintpt[i]=+∞
  wtotal=0
  storagept=0
  for j=i down to 1 do {
    storagept=storagept+sc_j·wtotal
    wtotal=wtotal+w_j
    totalpt=t_setup,j+t_byte,j·wtotal+mintpt[j-1]+storagept
    if (totalpt<mintpt[i]) then mintpt[i]=totalpt
  }}
```

Pseudocode 5-6. An $O(n^2)$ Dynamic Programming Algorithm.

The algorithm presented in the previous section can easily be optimized such that its time complexity becomes $O(n^2)$ (see Pseudocode 5-6). The optimization consists of maintaining the processing time incurred by storing bytes (*storagept*) from one value of j to another and modifying it locally. Furthermore, the values of j are considered in reverse order.

5.5.2.3. An $O(n \cdot log^2 n)$ Dynamic Programming Algorithm

This solution is based on a data structured, called segment tree, and on several particularities of the problem. Basically, we want to compute the same thing as before, an array *mintpt*, where *mintpt[i]* has the same meaning. When computing *mintpt[i]*, we can choose the time step j, which is the beginning of the time step interval ending at i, from the set $\{1,2,\ldots,i\}$. We will define the family of functions $f_j(x)$, representing the minimum total processing time for the first x time steps, if x is considered the last time step and j is the first time step in the interval ending at x. For each i, we will have to find the function f_j whose $f_j(i)$ value is minimum.

Before going further, we will define two new arrays, *scp* and *wp*, representing the prefix sums of the arrays *sc* and *w*:

$$scp_i=scp[i]=sc_1 + \ldots + sc_i \quad \textbf{(5-23)}$$

$$wp_i=wp[i]=w_1+\ldots+w_i \quad \textbf{(5-24)}$$

We will now define the functions f_j in more detail. A function f_j is defined on the interval $[j-1,n]$. The first value, $f_j(j-1)$ does not have a practical meaning, because a function f_j is considered only at the time steps $j, \ldots, n$. It is introduced in order to simplify the analysis. We have:

$$f_j(j-1)=mintpt[j-1]+t_{setup,j} \quad \textbf{(5-25)}$$

$$f_j(j)=mintpt[j-1] + t_{setup,j} + t_{byte,j} \cdot w_j = f_j(j-1) + t_{byte,j} \cdot w_j \quad \textbf{(5-26)}$$

The difference between two consecutive values of f_j is:

$$df_j(x) = f_j(x) - f_j(x-1) = t_{byte,j} \cdot w_x + (sc_j + sc_{j+1} + \ldots + sc_{i-1}) \cdot w_i \,,\ x \geq j \quad \textbf{(5-27)}$$

This difference contains the processing time of copying w_x bytes in the TCP buffer at time step j and the sum of processing times incurred by storing the w_x bytes in the TCP buffer until time step x. Using the prefix sum arrays introduced earlier, the difference can be rewritten as follows:

$$df_j(x)=t_{byte,j} \cdot w_x + (scp_{x-1} - scp_{j-1}) \cdot w_x = scp_{x-1} \cdot w_x + (t_{byte,j} - scp_{j-1}) \cdot w_x. \quad \textbf{(5-28)}$$

The difference is now composed of two terms: the term $scp_{x-1} \cdot w_x$, which depends only on the point at which the function is evaluated and a term which is composed of two factors, one of which is contant for a given function f_j and the other one depends only on the point where the function is evaluated. The factor which is constant (but possibly different for each function f_j) will be denoted by

$$p_j=t_{byte,j}-scp_{j-1}. \quad \textbf{(5-29)}$$

We will now change the definitions of the functions slightly and remove the term $scp_{x-1} \cdot w_x$. This term does not influence the relative ordering of the values of the functions f_j. After computing *mintpt[n]* using the new definitions of the functions, we will add at the end the sum of all the excluded terms:

$$S_{terms} = \sum_{i=1}^{n} scp_{i-1} \cdot w_i \quad \textbf{(5-30)}$$

With the new definitions of the functions, the equations for the differences of two consecutive values become $df_j(x) = f_j(x)-f_j(x-1)=p_j \cdot w_x$. The initial values $f_j(j-1)$ do not change. If we associate to each time i step an x-coordinate *wp[i]*, we can change the definitions of the functions further and obtain some new functions g_j, defined on the interval $[wp_{j-1},wp_n]$ and $g_j(x)=g_j(wp_{j-1})+p_j \cdot (x-wp_{j-1})$, where $g_j(wp_{j-1})=f_j(j-1)$. It is easy to see that the relationship between the functions g_j and f_j is:

$$g_j(wp[x])=f_j(x) \quad \textbf{(5-31)}$$

The nice thing about the g_j functions is that they are half lines and, thus, have the following useful property: each function g_j has the minimum value among all the functions either on an interval of x-coordinates $[l_j, r_j]$ or none of its values is a minimum value. The proof is quite easy. Let's assume that the function g_j has minimum values on two disjoint intervals $[l_{j1}, r_{j1}]$ and $[l_{j2}, r_{j2}]$, with $l_{j2}>r_{j1}$. There are two possibilities now. The first one is that there exists some function g_k, such

that $g_k(x)>g_j(x)$, for $x\leq r_{j1}-\varepsilon$ and $g_k(x)<g_j(x)$ for $x\geq r_{j1}$. In order for this to happen, the function g_k must have a slope p_k which is smaller than the slope p_j of the function g_j. But if this is the case, then $g_k(x)<g_j(x)$, for any $x\geq r_{j1}$, so the function g_j can never become minimum again, on the interval $[l_{j2}, r_{j2}]$. The second possibility is that a function g_k "started" at $x=r_{j1}$ (that is, r_{j1} is the first point on its domain of definition). But, from the way the functions are defined, the first value of a function g_k is equal to the minimum value of the functions g_p ($p<k$), plus $t_{setup,j}$, which is a positive quantity. So in this case, the function g_k couldn't have caused the function g_j to not be the function with the smallest value at $x=r_{j1}$.

With the observation that each function has minimum values on at most one interval, we can use a segment tree for storing half lines. The nodes on the last level of the segment tree correspond to the coordinates *wp[1], wp[2], …, wp[n]*, because these are the only points of interest. The pseudocode of the algorithm is the following:

```
// compute the arrays scp and wp
scp[0]=0; wp[0]=0
for i=1 to n do {
  scp[i]=scp[i-1]+sc_i
  wp[i]=wp[i-1]+w_i
}
// compute the array mintp
mintpt[0]=0
for i=1 to n do {
  ginit[i]=mintpt[i-1]+t_setup,i
  p[i]=t_byte,i-scp[i-1]
  // find the interval of normalized x coordinates [l_i,r_i] on which the function g_i is minimum
  [l_i,r_i]=find_interval(i)
  if ([l_i, r_i] is not void) then update(l_i, r_i, i, segment_tree_root)
  // find the minimum value among all the functions g_1,…,g_i
  mintpt[i]=get_min(i)
}
sum=0
for i=1 to n do sum=sum+scp[i-1]·w_i
return mintpt[n]+sum
```

Pseudocode 5-7. An $O(n\cdot log^2(n))$ Dynamic Programming Algorithm.

The functions ***find_interval***, ***update*** and ***get_min*** make use of the segment tree data structure. The *update* and *get_min* functions have $O(log(n))$ time complexity, while *find_interval* takes $O(log^2(n))$ time. Each node of the segment tree stores the index j of a half line whose interval $[l_i,r_i]$, at the moment the interval was computed, contained the interval $[left_x, right_x]$ completely, where $left_x$ and $right_x$ are the left and right endpoints of the interval corresponding to node x. Each node x of the tree has a pointer to its parent (*parent[x]*). This pointer is undefined for the root of the tree. The pseudocodes of the *update* and *get_min* functions are shown in Pseudocode 5-8 and 5-9.

The *find_interval* function is more complex than the other two. In this function we binary search for the first time step l_i (between i and n) where the value $g_i(wp[l_i])$ is the smallest value among all the functions' values. In a similar manner, the last time step r_i is binary searched, too. In order to find l_i, we first need to observe how the function g_i's values change relative to the minimum value of the other functions. In general (excluding particular cases), $g_i(wp[i-1])$ is larger than the minimum value. Then the difference between $g_i(x)$ and the minimum value at point x decreases, until $g_i(x)$ becomes smaller than the former minimum value at point x. The function g_i is minimum until $x=r_i$, after which the difference between $g_i(x)$ and the minimum value at point x increases, for $x>r_i$. This type of behavior suggests that a binary search on the differences between two consecutive values of the function $h(x)=g_i(x)-get\_min(x)$ is appropriate. At the end of a

find_interval(i) call, l_i contains the left endpoint of the interval in which g_i is minimum (or $l_i=n+1$ if such an interval does not exist). The value of r_i is computed in a similar manner.

```
update(l_i, r_i, i, node):
if ([left_node, right_node]=[l_i, r_i]) then {
  index[node]=i
} else {
  lson = the left son of node
  rson = the right son of node
  if (rson.left>r_i) then update(l_i,r_i,i,lson)
  else if (lson.right<l_i) then update(l_i,r_i,i,rson)
  else {
    update(l_i, lson.right, i, lson)
    update(rson.left, r_i, i, rson)
  }
}
```

Pseudocode 5-8. The *update* Function.

```
get_min(i):
node = the node in the segment tree corresponding to the interval [i,i]
val_min=+∞
while (node≠undefined) do {
  k=index[nod]
  gki=finit[k]+(wp[i]-wp[k-1])·p[k]
  if (gki<val_min) then val_min=gki
  nod=tata[nod]
}
return val_min
```

Pseudocode 5-9. The *get_min* Function.

```
find_interval(i):
left=i; right=n; l_i=n+1
while (left≤right) do {
  mid = (left+right) div 2
  // compute g_i[wp[mid+1]]
  gi_mid_1 = ginit[i]+(wp[mid+1]-wp[i-1])·p[i]
  // compute g_i[wp[mid]]
  gi_mid_2 = ginit[i]+(wp[mid]-wp[i-1])*p[i]
  // compute get_min(mid+1)
  fmin1=get_min(mid+1)
  // compute get_min(mid)
  fmin2=get_min(mid)
  dg=gi_mid_1-gi_mid_2
  dmin=fmin1-fmin2
  if (dg<dmin) then {
    l_i=mid
    right=mid-1
  } else
    left=mid+1
}
```

Pseudocode 5-10. The *find_interval* Function.

5.5.3. Possible Extensions

Several extensions and special cases of problems which are similar to the one considered in this section have been described in the literature. In this subsection we will adapt some of these extensions (and proposed solutions) to our problem.

The segment tree is used for maintaining the lower envelope of the g_i functions. If the slopes of the half-lines g_i are descending, then, instead of a segment tree, we can use a stack for storing the *x-intervals* defining the lower envelope. Let's assume that the stack has *top* levels. Each level *lev* ($1 \le lev \le top$) of the stack will be assigned an index *idx[lev]* and an x-value *xv[lev]*. We will always have $xv[0]=0$ and $xv[top+1]=+\infty$ (i.e. a sufficiently large value – in our case, we can use $xv[top+1]=wp_n+\varepsilon$, for any $\varepsilon>0$), unless we specifically set *xv[top+1]* at a different value. The x-intervals of the lower envelope will be $[xv(i),xv(i+1)]$ and the function with the minimum value on this interval will be $g_{idx[i]}$. The slopes of the functions in the stack are descending (i.e. $p[idx[lev]]>p[idx[lev+1]]$, for $1 \le lev \le top-1$). Initially, the stack is empty (i.e. $top=0$).

The stack must support the following operation: insert the half-line g_i, with slope *p[i]* and which is defined on the interval $[xstart(i),+\infty]$. If we know that the slopes of the half-lines g_i are sorted descendingly according to the order in which the half-lines are inserted into the stack, then we can use the following insertion algorithm. While ($top>0$) and ($xv[top+1]>xstart[i]$) and ($g_{idx[top]}(xv[top+1])>g_i(xv[top+1])$) do: { *(1)* if ($xv[top] \ge xstart(i)$) then: { *(1.1)* if ($g_{idx[top]}(xv[top])>g_i(xv[top])$) then *(1.1.1)* $top=top-1$ else: { *(1.1.2)* compute the x-coordinate *xcross* of the intersection of the half-lines $g_{idx[top]}$ and g_i ; *(1.1.3)* set $xv[top+1]=xcross$; *(1.1.4)* break the *while* loop } else: { *(1.2)* if ($g_{idx[top]}(xstart(i))>g_i(xstart(i))$) then: { *(1.2.1)* set $xv[top+1]=xstart(i)$; *(1.2.2)* break the *while* loop } else: { *(1.2.3)* compute the x-coordinate *xcross* of the intersection of the half-lines $g_{idx[top]}$ and g_i ; *(1.2.4)* set $xv[top+1]=xcross$; *(1.2.5)* break the *while* loop } }. At the end we set $top=top+1$, $idx[top]=i$ (since the slopes of the half-lines are descending, when inserted, the half-line g_i will always become part of the lower envelope) and $xv[top+1]=+\infty$. This algorithm is very similar to the one for computing the upper envelope of a set of half-lines, presented in section 7.4.1.

Another solution for the case of descending slopes is the following. We will binary search the highest level *lev* such that $xv[lev] \le xstart(i)$. Then, we will binary search the smallest level *lev'* from the interval of levels $[lev,top]$, such that $g_{idx[lev']}(xv[lev'+1])>g_i(xv[lev'+1])$. If ($lev'>lev$) then we compute the x-coordinate *xcross* of the intersection of the half-lines $g_{idx[lev']}$ and g_i. We set $top=lev'+1$, $xv[top]=xcross$, $idx[top]=i$ and $xv[top+1]=+\infty$. If ($lev'=lev$) and ($g_{idx[lev]}(xstart(i)) \ge g_i(xstart(i))$) then we will set $top=lev'+1$, $xv[top]=xstart(i)$, $idx[top]=i$ and $xv[top+1]=+\infty$; otherwise, we proceed as in the case ($lev'>lev$) (i.e. we compute *xcross*, and so on).

The first solution may take $O(n)$ time per insertion, but it will take $O(n)$ time overall (and, thus, $O(1)$ amortized time per insertion). The second solution will take $O(log(n))$ time per insertion.

In order to find the value of the lower envelope at a coordinate x, we will binary search the largest level *lev* for which $xv[lev] \le x$. The value is $g_{idx[lev]}(x)$.

With this extension, the algorithm is the following. For each time step i ($i=1,...,n$), in increasing order, we compute the parameters *ginit[i]* and *p[i]* which define the function g_i (the slope of g_i is *p[i]*, $xstart[i]=wp[i-1]$ and $g_i(xstart[i])=ginit[i]$) and we insert g_i into the stack. Then, *mintpt[i]* is equal to the value of the current lower envelope of the g_* functions at the coordinate $x=wp[i]$. Thus, the time complexity is $O(n \cdot log(n))$.

Because the values *wp[i]* are increasing (as i increases), we can improve the time complexity as follows. Let's assume that *lev(i-1)* is the level in the stack such that ($xv[lev(i-1)] \le wp[i-1]$) and ($xv[lev(i-1)+1]>wp[i-1]$) (at the moment when we want to compute *mintpt[i-1]*). When we want to compute *lev(i)* (for $x=wp[i]$, $i \ge 2$), we can proceed as follows: if ($lev(i-1) \ge top$) then $lev(i)=top$ else: { *(1)* $lev(i)=lev(i-1)$; *(2)* while ($xv[lev(i)+1] \le wp[i]$) do $lev(i)=lev(i)+1$ }. The stack may change when moving from the time step *i-1* to the time step *i*. We can find *lev(1)* easily, with a linear search (because the stack contains only g_1 at $i=1$). Overall, this algorithm based on a linear search takes $O(n)$ time for all the n time steps. Thus, the time complexity of the algorithm is now linear ($O(n)$).

Another extension consists of using a scenario tree, i.e. considering that there may be multiple branching possbilities after every time step. Each node i of the scenario tree (except for the root r) has a parent *parent(i)*: this means that after the advertisement corresponding to the node *parent(i)*, the next advertisement may be i. For each node i we have the same values as before: w_i, $t_{setup,i}$, $t_{byte,i}$ and sc_i. Each path from the root r of the tree to a vertex i defines a sequence *seq(i)* of time steps. We would like to compute the minimum total processing time *mintpt[i]* for every sequence of time steps *seq(i)* ($1 \le i \le n$; n=the number of vertices in the scenario tree). Of course, we could run the previously described algorithm for every sequence of time steps, but this would be very inefficient.

Another possibility consists of traversing the tree using DFS. We will maintain a data structure *DS* for storing the lower envelope of the half-lines on the path from the root to the current vertex i. When we first enter a vertex i, we proceed as follows. If ($i=r$) then we set: $scp[r]=sc_r$, $wp[r]=wp_r$, $ginit[r]=t_{setup,r}$ and $p[r]=t_{byte,r}$. If ($i \ne r$) then we compute: $scp[i]=scp[parent(i)]+sc_i$, $wp[i]=wp[parent(i)]+w_i$, $ginit[i]=mintpt[parent(i)]+t_{setup,i}$ and $p[i]=t_{byte,i}-scp[parent(i)]$. The *ginit[i]* and *p[i]* values define the g_i function. We have $xstart(i \ne r)=wp[parent(i)]$ (and $xstart(r)=0$).

Then, we will insert g_i into *DS*. When moving from a vertex i back to its parent *parent(i)*, we will remove g_i from *DS*. Data structures for fully dynamic online or offline maintenance of the lower envelope of lines exist (note that our case can be considered offline, because the sequence of insertions and deletions can be computed before-hand).

If the slope of g_i is smaller than the slope of $g_{parent(i)}$ (for every vertex i), then *DS* can be a stack. When inserting g_i into *DS*, we will use the second solution presented earlier (with $O(log(n))$ time complexity). Before inserting g_i into *DS*, we will store the value of *top* into *oldtop(i)*. After modifying *top* but before modifying *xv[top]*, *xv[top+1]* and *idx[top]*, we will set *newtop(i)=top*, *oldxv(i)=xv[top]*, *oldxv1(i)=xv[top+1]* and *oldidx(i)=idx[top]*. Then, when moving from the vertex i back to *parent(i)*, we will set *xv[newtop(i)]=oldxv(i)*, *xv[newtop(i)+1]=oldxv1(i)* and *idx[newtop(i)]=oldidx(i)*. Afterwards, we will set *top=oldtop(i)*. This way, removing a half-line from *DS* takes $O(1)$ time.

In order to compute *mintpt[i]* we just find the value of the current lower envelope (after inserting g_i into *DS*) at $x=wp[i]$. The time complexity of the algorithm for the scenario tree is $O(n \cdot log(n))$ (even if the slopes are not descending along a tree path ; but in this case, the data structure *DS* must be significantly more complicated).

5.6. Optimal Deadline-Constrained Packet Transfer Strategy

We consider a directed graph with n vertices and m edges. A packet is sent from the source node s at time 0 and must reach the destination node d by time T. Every directed edge e from a vertex u to a vertex v has an associated start time *tstart(e)* and a finish time *tfinish(e)* (*tfinish(e)>tstart(e)*). The meaning of these parameters is that the packet can be sent along that edge, from u to v, starting only at the moment *tstart(e)* and will only arrive at vertex v at the moment *tfinish(e)*. Thus, the edge e corresponds to a reservation in the underlying network. Moreover, out of the total packet transmission time (equal to *tfinish(e)-tstart(e)*), *twait(e)* is the total time during which the packet has to wait in the waiting queues (e.g. it must wait for some data processing task or must wait until other packets before it are sent along the edge). The time between the moment when the packet arrives at a vertex u and the moment when it is sent to another vertex v (or between the moment when it last arrives at vertex d and the moment T) also counts as waiting time. We want to find a packet transfer strategy minimizing the total waiting time.

For every vertex v of the graph we will sort together the incoming and outgoing edges in increasing order, according to a weight assigned to every edge. For an incoming edge e from a vertex u to vertex v, the weight is *w(v,e)=tfinish(e)*; for an outgoing edge e from vertex v to a vertex u, the weight is *w(v,e)=tstart(e)*. If two edges (an incoming and an outgoing one) have the same weight, then we will place the incoming edge before the outgoing edge in the sorted order. For every edge in the sorted order of a vertex v we will store its type: incoming or outgoing.

Let *deg(v)* be the total number of incoming and outgoing edges adjacent to vertex *v*. We will compute *TWmin(v,i)*=the minimum total waiting time required for the packet to be located at vertex *v* at the time moment *tm(v,i)*=the weight of the i^{th} edge in the sorted order for vertex *v* ($1 \le i \le deg(v)$). We will consider $TWmin(v,0)=+\infty$ and *tm(v,0)=0* for every vertex $v \neq s$ and *tm(s,0)=TWmin(s,0)=0*. We will sort ascendingly all the time moments *tm(v,i)* ($i \ge 1$) in increasing order (e.g. by merging the lists of time moments *tm(*,*)*) and we will store for each moment the associated values *v* and *i*. We will traverse all the time moments *tm(v,i)* in increasing order.

If *tm(v,i)* corresponds to an incoming edge *e* (from a vertex *u* to vertex *v*), then we will first find the index *j* of the edge *e* in the sorted order of the edges adjacent to vertex *u*. We will have *TWmin(v,i)=min{TWmin(v,i-1)+tm(v,i)-tm(v,i-1), TWmin(u,j)+twait(e)}*. If *tm(v,i)* corresponds to an outgoing edge *e* from vertex *v* to a vertex *u* then we set *TWmin(v,i)=TWmin(v,i-1)+tm(v,i)-tm(v,i-1)*.

We can find the index *j* of an edge *e* in the sorted order of a vertex *u* by using a hash table *HT(u)*. After sorting the edges adjacent to vertex *u* we traverse these edges: let the edge *e'* be the i^{th} edge in this order - then we insert the pair *(key=e', value=i)* in *HT(u)*. Thus, in order to find the index *j* associated to an edge *e* in the sorted order of a vertex *u* we just search in *HT(u)* the value associated to the key *e*. Such a lookup takes *O(1)* time.

The overall time complexity is $O(m \cdot log(m))$. The answer (the minimum total waiting time) is $min\{TWmin(d,i)+T-tm(d,i) | 1 \le i \le deg(d), tm(d,i) \le T\}$. If the packet does not have to wait at the vertex *d* until the time moment *T* (i.e. there is no deadline), then the answer is simply $min\{TWmin(d,i) | 1 \le i \le deg(d)\}$.

If all the time moments are integer and not too large, then we can sort all the *O(m)* time moments *tm(*,*)* in *O(m+TMAX)* time, by using a procedure similar to count-sort. *TMAX* is the maximum value of a time moment. We will maintain a list *L(tp)* for every time moment *tp* starting from *0* and up to *TMAX*. We will add the information associated to every edge *e* from *u* to *v* both into *L(tstart(e))* and into *L(tfinish(e))*. Then, we just concatenate all the lists *L(tp)* in increasing order of *tp*. Then, by traversing the list of all the sorted edges, we can construct the list of sorted edges corresponding to every vertex *v* (every edge *e* from *u* to *v* is added at the end of the lists of the vertices *u* and *v*). As a final step, we might need to adjust the lists of sorted edges of the vertices, by placing the incoming edges before the outgoing edges with the same weight (using a procedure similar to the one we just presented, for each group of edges with the same weight; the two categories for each group are *incoming* and *outgoing* edges). Thus, we obtain an *O(m+TMAX)* overall time complexity.

As we can notice, the problem can also be interpreted as a shortest path problem in the graph of the pairs *(v,i)*, where the starting pair is *(s,0)*. We have an edge from each pair *(v,i-1)* to the pair *(v,i)* with cost *tm(v,i)-tm(v,i-1)* ($1 \le i \le deg(v)$). Moreover, we have an edge from each pair *(u,j)* to a pair *(v,i)*, with cost *twait(e)*, if *e* is an edge from *u* to *v* and is the j^{th} edge in the sorted order of the vertex *u* and the i^{th} edge in the sorted order of the vertex *v*. With this interpretation, we can compute a shortest path from *(s,0)* to the pairs *(d,*)*, in $O(m \cdot log(m))$ time (the graph of pairs has *O(m)* vertices and edges). If we denote by *TWmin(v,i)*=the length of the shortest path from *(s,0)* to *(v,i)*, then the answer is computed the same as before.

5.7. Optimal Budget-Constrained Packet Transfer Strategy

We maintain all the assumptions from the previous problem (i.e. the same types of input data). Additionally, every edge *e* also has a cost $cost(e) \ge 0$. A packet is available at the source node *s* at the time moment *0* and must reach the destination node *d* ($d \neq s$). We have no deadline constraints, but we have a budget limit *B*. The total cost of the edges traversed by the packet must not exceed *B*. Moreover, we have the following optimization objective: we want to minimize the maximum interval of waiting time. The packet is waiting for a time interval between the moment when it reaches a vertex *v* (different from *d*) and the moment when it departs to another vertex *u*. Moreover, a packet also waits for several time intervals, whose maximum length is *twait(e)* if it sent along the

edge e; none of the waiting intervals along an edge e contains the time moments $tstart(e)$ and $tfinish(e)$.

The proposed solution is based on binary searching the maximum length of an interval of waiting time. Let's assume that we choose within the binary search a value X. We will then compute the minimum cost path from s to d which contains no waiting time interval of length larger than X. Let C be the total cost of this path. If $C \geq B$ then X is a feasible value and we will consider smaller values in the binary search next; if $C > B$ then X is not feasible and we will consider larger values within the binary search next. The smallest feasible value of X is the minimum maximum delay of a path from s to d whose total cost does not exceed B. We will now focus on testing the feasibility of a value X.

For every vertex v we will sort all of its outgoing edges e (from v to another vertex u) in increasing order of their $tstart(e)$ values. Let $out(v,i)$ be the i^{th} outgoing edge in this order ($1 \leq i \leq outdeg(v)$, where $outdeg(v)$=the number of outgoing edges adjacent to vertex v). Similarly, we sort all the incoming edges e (from another vertex u to the vertex v) adjacent to vertex v in increasing order of their $tfinish(e)$ values. Let $in(v,i)$ be the i^{th} incoming edge in this order ($1 \leq i \leq indeg(v)$; $indeg(v)$=the number of incoming edges adjacent to vertex v).

We will now sort all the incoming and outgoing edges of all the vertices together, according to their weight: the weight of an incoming edge e is $tfinish(e)$ and that of an outgoing edge e is $tstart(e)$. An edge e will occur two times in this order: as an incoming edge to a vertex v and as an outgoing edge from a vertex u. We obtain this sorted order by merging correctly all the lists $out(*)$ and $in(*)$ of all the n vertices. The overall time complexity so far is $O(n+m \cdot log(m))$. These steps are performed before running any feasibility test.

If the time moments $tstart(*)$ and $tfinish(*)$ are not too large, then we can sort all the edges in $O(m+TMAX)$ time, where $TMAX$ is the largest time moment. Based on sorting all the edges, we can construct in $O(n+m)$ overall time all the lists $out(*)$ and $in(*)$.

Let's assume now that we want to compute the minimum cost path from s to d, such that the length of the maximum waiting time interval is at most X. We will compute the values $Cmin_{out}(v,i)$ for every vertex v and $1 \leq i \leq outdeg(v)$, meaning the minimum total cost for leaving vertex v through the edge $out(v,i)$ and without exceeding the maximum waiting time limit, and $Cmin_{in}(v,i)$ for every vertex v and $1 \leq i \leq indeg(v)$, meaning the minimum total cost for reaching vertex v through the edge $in(v,i)$. We will now traverse all the incoming and outgoing edges in the order computed earlier. For every vertex v we will maintain a deque $DQ_{in}(v)$ for the incoming edges and the index $last_{in}(v)$ in the corresponding sorted order of the last edge inserted in the deque (initially, $last_{in}(v)=0$). The deques will store $(time,value)$ pairs, sorted increasingly according to both elements of the pair. The functions $getFirst()$ and $getLast()$ will return the first and the last pair of a deque.

Let's assume that we reached an outgoing edge e, such that $out(v,i)=e$. While ($last_{in}(v)+1 \leq indeg(v)$) and ($tfinish(in(v,last_{in}(v)+1)) \leq tstart(out(v,i))$) we perform the following actions: *(1)* we increment $last_{in}(v)$ by *1*; *(2)* while ($DQ_{in}(v)$ is not empty) and ($DQ_{in}(v).getLast().value \geq Cmin_{in}(v,last_{in}(v))$) we remove the last pair from $DQ_{in}(v)$; *(3)* we add at the end of $DQ_{in}(v)$ the pair $(time=tfinish(in(v,last_{in}(v))), value=Cmin_{in}(v,last_{in}(v)))$.

After this, as long as $DQ_{in}(v)$ is not empty and ($DQ_{in}(v).getFirst().time+X<tstart(e)$) we remove the first pair from $DQ_{in}(v)$.

If ($v=s$) and ($tstart(e) \leq X$) then we set $Cmin_{out}(v,i)=0$. Otherwise, if $DQ_{in}(v)$ is empty, then $Cmin_{out}(v,i)=+\infty$; if $DQ_{in}(v)$ is not empty, then $Cmin_{out}(v,i)=DQ_{in}.getFirst().value$.

Let's assume now that we reached an incoming edge e, such that $in(v,i)=e$. Let j be the index such that $out(u,j)=e$ (where e is an edge directed from u to v). We can find the indices i and j like in the previous problem, by using a hash table $HT_{out}(p)$ for the outgoing edges of every vertex p and a hash table $HT_{in}(q)$ for the incoming edges of every vertex q.

If ($twait(e)>X$) then $Cmin_{in}(v,i)=+\infty$; otherwise, $Cmin_{in}(v,i)=Cmin_{out}(u,j)+cost(e)$.

The minimum cost of a path whose maximum waiting time interval has length at most X is $min\{Cmin_{in}(d,i) | 1 \leq i \leq indeg(d)\}$. If the packet must wait at the vertex d until a fixed time moment T, like in the previous problem, we will set $Cmin_{in}(d,i)=+\infty$ if ($tfinish(in(d,i))+X<T$). Then, the

minimum cost of the path will be $min\{Cmin_{in}(d,i) | 1 \le i \le indeg(d),\ T-X \le tfinish(in(d,i)) \le T\}$. The time complexity of the feasibility test is $O((n+m) \cdot log(max(tfinish(*))))$.

If we have $p \ge 1$ agents working in parallel, then we can replace the binary search by a $(p+1)$-ary search. We start with an interval $[0,XMAX]$ in which the searched value of X is located. Then, let's assume that the current interval of feasible values is $[a,b]$. We choose p values within this interval, e.g. $x_i=a+i \cdot (b-a)/(p+1)$ $(1 \le i \le p)$. Then, every agent i runs a feasibility test for the value x_i. Afterwards, we find the smallest feasible value x_j (based on the decisions of the algorithm) and we set $[a,b]=[x_{j-1},x_j]$; if no feasible value is found, we set $[a,b]=[x_p,b]$. The search stops when the length of the interval $[a,b]$ is smaller than a threshold $\varepsilon > 0$.

5.8. Minimum Cost Path Reservations in Trees

We consider a rooted tree with n vertices (numbered from 1 to n). The root of the tree is the vertex r. Every vertex i (except r) has a parent in the tree: $parent(i)$. Every vertex i has the same amount of buffer space. The buffers of some vertices i are full of data (i.e. $full(i)=true$), while those of other vertices j are empty ($full(j)=false$). We want to establish a reservation from a vertex x to the root r, such that the buffer of every vertex on the unique path from x to r is empty.

In order to accomplish this, we can perform a sequence of the following type of moves: choose a vertex p whose buffer is full and a son q of the vertex p, whose buffer is empty; then, transfer the data from p's buffer to q's buffer (thus, after the move, p's buffer becomes empty, while q's buffer becomes full); the cost of such a move is $c(p,q)$. We are interested in finding a sequence of moves with minimum total cost which frees the buffers of all the vertices on the path from x to r (including the endpoints of the path). We will provide an $O(n \cdot log(n))$ time algorithm.

First, we construct the path $path(1)=x,\ path(2)=parent(x),\ ...,\ path(k)=r$, where $path(i)=parent(path(i-1))$ for $2 \le i \le k$. Then, we compute the „prefix" sums: $spath(1)=0$ and $spath(2 \le i \le k)=spath(i-1)+c(path(i),path(i-1))$. Afterwards, we traverse the path nodes $path(i)$ in ascending order of the index i, from $i=1$ up to $i=k$. Let's assume that we reached the vertex $path(i)$. We remove the vertex $path(i-1)$ from the list of sons of the vertex $path(i)$ (if $i>1$). Then, we will perform a DFS traversal in the subtree of the vertex $path(i)$ (consisting of $path(i)$ and all of its descendants in the tree, except for $path(i-1)$ and $path(i-1)$'s descendants - in the case $i>1$).

For every vertex j visited during this traversal, we will compute $csum(j)$: we have $csum(path(i))=0$ and $csum(j \ne path(i))=csum(parent(j))+c(parent(j),j)$ (where j is a descendant of $path(i)$; since $path(i-1)$ was removed from the list of sons of $path(i)$, for $2 \le i \le k$, the descendants of $path(i-1)$ are not considered here). Then, for every such vertex j, if ($j \ne path(i)$) and ($full(j)=false$), we will insert the value $cval(j)=(csum(j)-spath(i))$ into a min-heap H (which is empty at the beginning of the algorithm, i.e. before considering the first vertex on the path, $path(1)$). After finishing the traversal, we check if $full(path(i))=true$. If it is so, then the data from the buffer of the vertex $path(i)$ must be moved somewhere in its subtree or somewhere in the subtree of one of the vertices $path(j<i)$. We need to find the vertex p in the subtrees of the vertices $path(j \le i)$, such that $full(p)=false$ and the sum of the values of the move costs of the edges on the original tree path from $path(i)$ to p is minimum (and $p \ne path(j)$ for every $1 \le j \le i$).

Then, let $path'(1)=p,\ path'(2)=parent(p),\ ...,\ path'(k')=path(i)$ (where $path'(q)=parent(path'(q-1))$ for $2 \le q \le k'$). We can move data from $path'(q)$ to $path'(q-1)$, in increasing order of q ($2 \le q \le k'$), with a total cost equal to the sum of the values $c(path'(q), path'(q-1))$ ($2 \le q \le k'$). We could find the vertex p in linear time, by considering all the vertices in the subtrees of $path(j \le i)$: the cost of the path from a vertex v in the subtree of a vertex $path(j \le i)$ to the vertex $path(i)$ is equal to $spath(i)+(csum(v)-spath(j))$. Note, however, that the minimum such value is equal to $spath(i)$ plus the minimum value in the min-heap H. Thus, all we need to do is to extract the minimum value val from H and increase the total cost of the strategy (which is 0 at the beginning of the algorithm) by ($spath(i)+val$).

Thus, if we only need to compute the cost, we can do this in $O(n \cdot log(n))$ time. If we also need to perform the moves, then we can associate to every value val from the heap the vertex p to which the value belongs (i.e. $cval(p)=val$). With this extra information, we can reconstruct the

sequence of moves for every vertex *path(i)* (for which *full(path(i))* was originally *true*) in $O(n)$ time. Since we can perform $O(n^2)$ moves overall, the time complexity in this case becomes $O(n^2)$.

Another $O(n \cdot log(n))$ solution for computing the optimal cost is the following. We traverse the tree from top to bottom and we compute for every vertex j the value *croot(j)*: $croot(r)=0$ and $croot(j \neq r)=croot(parent(j))+c(parent(j),j)$. Then, we mark all the vertices on the path from x to r (i.e. the vertices *path(1), ..., path(k)*), leaving the other vertices unmarked. Afterwards, we assign DFS numbers *DFSnum(j)* to all the vertices j of the tree and, for each vertex j, we compute *DFSmax(j)*=the maximum DFS number of a vertex in the subtree of the vertex j (including vertex j).

We construct a segment tree over the sequence of DFS numbers. For each unmarked vertex j, we set the value corresponding to the segment tree leaf *DFSnum(j)* as being equal to *croot(j)*. The values corresponding to the leaves *DFSnum(j)* of the marked vertices j are set to $+\infty$. Then, we traverse the vertices *path(i)* in increasing order of i, from $i=1$ to $i=k$. For each vertex *path(i)*, we query the segment tree for computing the minimum value *val* within the interval of leaves *[DFSnum(path(i)), DFSmax(path(i))]*. If $val=+\infty$, then no solution exists. Otherwise, we increase the total cost (which is initially *0*) by (*val-croot(path(i))*). We also compute the leaf p whose value is equal to *val*. In the segment tree, we set the value of the leaf p to $+\infty$ (so that the same leaf will not be considered for other vertices on the path from x to r). Then, the values of the ancestors of the leaf p are modified accordingly (so that we can perform range minimum queries). Since each segment tree query and update takes $O(log(n))$ time, the overall time complexity is $O(n \cdot log(n))$ (because $k=O(n)$).

5.9. Resource Processing Problems

In this section we consider two resource processing problems. These problems are somewhat generic, and not strictly related to the topic of communication flow scheduling. However, since scheduling data transfers involves the handling of several types of resources, the two considered problems are within the scope of this chapter.

5.9.1. Optimal Resource Gathering Strategy

We have n resource containers: each container i stores $z(i) \geq 1$ resource units in it ($z(i)$ is an integer). We can perform operations consisting of two steps:

1) Choose a container i (containing $z(i)$ resource units) and remove from it $q+1$ resource units (for any number $0 \leq q \leq z(i)-1$) which are distributed into two new containers a and b as follows: container a will store 1 resource unit (we set $z(a)=1$) and container b will store q resource units (we set $z(b)=q$); if $q=0$ then we do not need the container b. $z(i)$ is decremented by $q+1$.
2) Choose one (or two) other container(s) u (and v) such that $u \neq a$ (and $v \neq a$) and put all their contents into the newly created container a, setting $z(a)=z(a)+z(u)$ ($z(a)=z(a)+z(u)+z(v)$) and then $z(u)=(z(v)=)0$.

We want to gather all the resource units into a single container by performing a minimum number of operations. First, it should be obvious that we can gather all the resources into one container after $n-1$ operations. At each operation, in the first step, we choose any container i and split its contents fully into two containers a and b (such that container i remains empty); in the second step we choose another container j, such that the contents of j and of the newly created container b, are placed together in container a. Thus, after every such operation, the containers i and j will have 0 resource units in them.

Nevertheless, we can do a little better. We will sort the containers such that we have $z(1) \leq z(2) \leq .. \leq z(n)$. We will compute the partial sums $sz(i)=z(1)+...+z(i)$ ($sz(0)=0$ and $sz(1 \leq i \leq n)=sz(i-1)+z(i)$). We will compute the largest index k for which $sz(k)<n-k$. During the first $sz(k)$ operations we perform the following actions: at operation t ($1 \leq t \leq sz(k)$) we choose a container $c(t)$ in the first step, such that: if ($t=1$) then $c(t)=1$; otherwise, if ($z(c(t-1))>0$) then $c(t)=c(t-1)$ else $c(t)=c(t-1)+1$. We choose $q=0$, i.e. we place one resource unit from $c(t)$ into the new container $a(t)$.

Then, in the second step, we choose the containers $k+t+1$ and the container g containing the total contents from the containers $k+1,..., k+t$ (at $t=1$, we have $g=k+1$) and the resources removed during the first $t-1$ operations. Then, we place the contents of the containers $k+t+1$ and g into $a(t)$ and we set $g=a(t)$.

After this first stage of the algorithm, there are still $n'=n-k-sz(k)$ non-empty containers left (including the container g which contains $sz(k+sz(k)+1)$ resource units). The contents of these last n' containers can be gathered into a single container in $n'-1$ operations, using the strategy mentioned earlier. Thus, the total number of required operations is $sz(k)+n-k-sz(k)-1=n-k-1$.

5.9.2. Optimal Resource Payment Strategy

We consider a compact block consisting of n resource units (n is an integer), out of which we have to pay 1 resource unit per time unit, as follows. At every time unit we either pay one block of one resource unit, or we pay a block of $q>1$ resource units (for any number q) and we receive change as a sum of blocks containing $q-1$ resource units overall. The change can only be received from the blocks paid during the previous time units.

Before paying during a time unit, we are allowed to break a block of resources, as follows: if the block contains q resource units then it is broken into three blocks, containing 1, p, and $q-p-1$ resource units respectively (where we can choose the value of the integer number p, $0\leq p\leq q-1$). We can perform the breaking operation as many times as we want before paying once. Blocks cannot be glued (back) together and no block that was paid once can be broken in order to get the exact change back (but, if we get the block back, we can break it later when we have to pay). We want to be able to pay the required resources for n time units, and we want to perform the minimum number of breaking operations.

We will compute $S(k)$=the maximum amount of resource units the initial block may have such that we can pay for $S(k)$ time units by performing exactly k breaking operations. In order to compute $S(k)$ we will use the following arguments. We will consider that all the breaking operations are performed in the beginning. Thus, we obtain k blocks of 1 resource unit and (at most) $k+1$ blocks of (possibly) other sizes. During each of the first k time units, we will pay with one block of 1 resource unit. In the $(k+1)^{st}$ time unit we will pay a block of $k+1$ resource units, obtaining back all the k resource units that were paid during the first k time units. Then, for the next k time units we again pay with blocks of 1 resource unit. During the $(2\cdot k+2)^{nd}$ time unit we will pay with a block of $(2\cdot k+2)$ resource units, obtaining back all the blocks paid so far, whose total size is $(2\cdot k+1)$ resource units. Then, during the next $(2\cdot k+1)$ time units we will use the same strategy as for the first $(2\cdot k+1)$ time units, and so on. Thus, $S(k)=k+(k+1)+(2\cdot k+2)+(4\cdot k+4)+...+(2^k\cdot k+2^k)=2^{k+1}\cdot k+2^{k+1}-1=2^{k+1}\cdot(k+1)-1$. Then, we can binary search the minimum value $kmin$ for which $S(kmin)\geq n$ (if $(S(k)<n)$ then $k<kmin$ else $k\geq kmin$). $kmin$ is the minimum number of required breaking operations in order to be able to pay for n time units according to the rules presented above.

5.10. Maximum Cost Bipartition of a Graph

We consider an undirected graph with n vertices and m edges. Every edge (i,j) has a cost $c(i,j)>0$ (e.g. the bandwidth of a network link). We want to split the vertices into two disjoint subsets V_1 and V_2, such that the sum of the costs $c(i,j)$ with $i\in V_1$ and $j\in V_2$ is as large as possible (e.g. the maximum amount of data per time unit that can be transferred form the nodes in V_1 to those in V_2).

We will present a heuristic solution for this problem. We will start by computing a maximum cost spanning tree. Such a tree can be computed by using Prim's algorithm (in which we always choose the maximum cost edge connecting a selected vertex to an unselected vertex) or Kruskal's algorithm (in which we sort the edges in decreasing order of their costs). Let's consider that the computed tree has a root at a vertex r. We will assign the vertices i on odd levels to V_1 (we set $set(i)=1$) and the vertices j on even levels to V_2 (we set $set(j)=2$).

Then, we will try to improve the current solution. We will assign to each vertex i a value $v(i)$, which is equal to: the sum of the values $c(i,j)$ (with j a neighbor of i such that $set(j)=set(i)$) minus

the sum of the values *c(i,p)* (with *p* a neighbor of *i* such that *set(p)≠set(i)*). We will insert the pairs *(v(i),i)* into a max-heap *H*. Then, we will repeatedly extract from *H* the pair *(v(i),i)* with the maximum value of *v(i)*. If (*v(i)≤0*) then the agorithm stops. Otherwise, we move the vertex *i* from the set $V_{set(i)}$ to the set $V_{3-set(i)}$ (we will set *set(i)=3-set(i)*). Afterwards, we will recompute the value *v(i)* and the values *v(j)* of the neighbors *j* of *i*. *v(i)* will be computed from scratch without any penalties. The values *v(j)* of vertex *i*'s neighbors can be recomputed from scratch, or we can simply adjust them: if (*set(i)=set(j)*) (before changing the value of *set(i)* the vertices *i* and *j* were in different sets), then we set *v(j)=v(j)+2·c(i,j)*; if (*set(i)≠set(j)*) (before changing the value of *set(i)* the vertices *i* and *j* were in the same set) then we set *v(j)=v(j)-2·c(i,j)*. After recomputing *v(i)* and the values *v(j)* of all the neighbors *j* of the vertex *i*, we will adjust these values in *H*, too (e.g. we remove the old pairs *(v(i),i)* and *(v(j),j)* from *H* and we insert the pairs with the newly computed values *v(i)* and *v(j)*).

If every vertex has at most a constant number of neighbors *CN*, the time complexity per iteration is *O(log(n))*. The algorithm can be stopped when the number of iterations it performed exceeds some threshold.

5.11. Graceful Labeling of a Cycle

We consider a cycle containing *n* vertices. We want to assign to every node on the cycle a distinct label, from the set *{1, ..., n+1}*. After labeling the nodes, we assign to every edge *(u,v)* on the cycle a label equal to the absolute value of the difference of the labels assigned to *u* and *v*. We want to find a node labeling such that every label from *1* to *n* occurs on the edges of the cycle (i.e. a *graceful labeling*). This problem has applications in frequency assignment in wireless networks.

Since one of the edges of the cycle must have the label n, then the two endpoints of this edge must be labeled with *1* and *n+1*. Let's consider now the cycle without the edge labeled with *n*. We obtain a path, in which the first node has label *1* and the last node has label *n+1*. Let's consider that we want the label of the first edge on the path (between the first and second node on the path) to be *k*. Under these conditions, we will first try to assign to the other edges on the path (from the last edge towards the second one) the labels *n-1, n-2, ..., k+1, k-1, ..., 1*. Let's denote by *dif(i)* the label of the ith edge on the path (when counting from the first node of the path towards the last).

We will consider the following algorithm, which will construct the sequence *x(i)* (*x(i)*=the label of the i^{th} node on the path. We set *x(n)=n+1* and *used(n+1)=true* (and *used(1≤i≤n)=false*). Then, for *i=n-1, n-2, ..., 1*, we will perform the following steps: if (*x(i+1)+dif(i)≤n*) and (*used(x(i+1)+dif(i))=false*), then we set *x(i)=x(i+1)+dif(i)* and *used(x(i))=true*; otherwise, if (*x(i+1)-dif(i)≥1*) and (*used(x(i+1)-dif(i))=false*), then we set *x(i)=x(i+1)-dif(i)* and *used(x(i))=true*; otherwise, no solution is found using this procedure (when *dif(1)=k*). If all the nodes were labeled and (*x(1)=1*), then we found a solution with *dif(1)=k*.

We can then try a second option, as follows: we set *dif(1)=k*, and *dif(2), ..., dif(n)* will be equal to *n-1, n-2, ..., k+1, k-1, ..., 1*. We will start with *x(1)=1* and *used(1)=true* (and *used(2≤i≤n+1)=false*). Then we will compute, in order, from *i=2* to *n*, the values *x(i)*. If (*x(i-1)+dif(i-1)≤n*) and (*used(x(i-1)+dif(i-1))=false*) then we set *x(i)=x(i-1)+dif(i-1)*; otherwise, if (*x(i-1)-dif(i-1)≥1*) and (*used(x(i-1)-dif(i-1))=false*) then we set *x(i)=x(i-1)-dif(i-1)*; otherwise, no solution can be found with this procedure (when *dif(1)=k*). If all the nodes were labeled and (*x(n)=n+1*) then we found a solution.

If we consider all the possible values of *k* and, for each of them, we use the procedures described above, we will certainly find a solution (if it exists). This wat, we obtained an algorithm with an $O(n^2)$ time complexity. If we run this algorithm for moderate values of n, we notice that, if a solution exists, then the value of *k* is close to *n/2*. Thus, it is sufficient to consider only a small number of values for *k*, around the number *n/2* (e.g., the natual numbers from the interval *[n/2-d, n/2+d]*, where *d* is a small value; *d=5* is a good choice, but a detailed analysis can show us that *d=1* is enough). This way we obtained an algorithm with *O(n)* time complexity. We also notice that solutions exist only for those values of n for which *(n mod 4)* is equal to *0* or *3*. By *n/2* we denoted the integer part of the division of *n* by *2*.

Chapter 6 – Multicast Communication Optimization Techniques

As mentioned in Chapter 2, there is almost no multicast support in the Internet. Thus, at the moment, the only feasible solutions consist of application-level multicast routing techniques. In the first part of this chapter we will introduce a novel multicast tree architecture, which maintains a small diameter tree with bounded node degrees. The rest of this chapter is dedicated to offline multicast communication optimization problems (e.g. computing optimal constrained multicast strategies in tree or tree-like networks). The original contributions presented in this chapter were published in [Andreica, Tîrşa and Ţăpuş, 2009b], [Andreica and Ţăpuş, 2008d] and [Andreica and Ţăpuş, 2008i].

6.1. Bounded Degree Small Diameter Multicast Tree

Maintaining a small-diameter multicast tree over all the peers of a distributed system is a desirable feature in several types of applications. For instance, in Internet TV and live streaming applications, it is more bandwidth-efficient to use a multicast tree instead of sending multiple unicast streams. Moreover, by using a self-organizing multicast tree, there is no need for the content source to be aware of all the peers in the group. Some of these content distribution applications require that the latency of each path from the source to a destination be as small as possible. In this respect, it is desirable for the multicast tree to have a small diameter (diameter=the largest distance between any two nodes in the tree). If the tree has a small diameter, then any of the tree nodes can become a content producer and distribute its content (or send content search queries) to all the other peers in the tree efficiently.

Another condition for a good multicast tree is for the traffic load on each node of the tree to be equitably distributed. We can quantify this request in many ways. In this section we consider a simple measure: the degree of every node in the tree must be bounded from above by a (small) fixed value $K \geq 2$. Although it is possible for every peer to use its own value of K, in this section we will consider only the case when all the peers make use of the same value K.

In this section we present an implementation of the multicast tree based on a peer-to-peer topology. The topology maintains bounded degrees for all the nodes, has small diameter and supports node arrivals and departures. The neighbouring peers in the topology periodically exchange information among each other (gossip), which is particularly useful when a peer joins or leaves the topology. We will describe next the gossiping, joining and leaving processes for the peer-to-peer architecture.

6.1.1. Gossiping in the Multicast Tree

Periodically, every peer in the tree sends two types of gossiping messages. The first type is sent to the peers at distance (at most) two in the tree and simply broadcasts its existence to these peers. Thus, every peer X knows all the peers located at distance one (neighbors) and two (2-neighbors) from X in the tree. For every 2-neighbor Z, peer X maintains the neighbor Y which is on the path between X and Z. Since the degree of every peer is at most K, every peer X is aware of at most $K+K \cdot (K-1)=K^2$ other peers. Every peer X maintains two estimated values for every tree neighbor Y: *NumPeers(X,Y)* and *Dmax(X,Y)*. *NumPeers(X,Y)* is an estimate of the total number of peers in *T(X,Y)*=the part of the tree which contains peer Y but does not contain peer X (i.e. if we consider the tree rooted at X, then peer Y is a son of peer X and *T(X,Y)* is the subtree rooted at peer Y); see also Fig. 6-1. *Dmax(X,Y)* is an estimate of the longest path (in terms of peers) from peer Y to the farthest peer in *T(X,Y)*.

The second type of gossiping message is sent by every peer Y to every tree neighbor X and contains the new values *NumPeers(X,Y)* and *Dmax(X,Y)* that peer X should use. Peer Y computes

NumPeers(X,Y) based on its own values *NumPeers(Y,*)*. Let's denote by *SumNumPeers(Y)* the sum of all the values *NumPeers(Y,*)* stored by peer *Y*. Then *NumPeers(X,Y)=SumNumPeers(Y)-NumPeers(Y,X)+1*. Let *DistMax(Y, j)* be the j^{th} largest distance among all the values *Dmax(Y,*)* and let *DistMaxNeigh(Y,j)* be the neighbor *Z* such that *Dmax(Y,Z)=DistMax(Y,j)* and *Z≠DistMaxNeigh(p)* for all *1≤p≤j-1* (*DistMax(Y,j)=0* and *DistMaxNeigh(Y,j)=undefined* if *j* is larger than the number of neighbors peer *Y* has). We will compute *DistMax(Y,j)* and *DistMaxNeigh(Y,j)* only for *j=1,2*. The value *Dmax(X,Y)* sent by peer *Y* to peer *X* is computed as follows: if *DistMaxNeigh(Y,1)≠X*, then *Dmax(X,Y)=1+DistMax(Y,1)*; otherwise, *Dmax(X,Y)= 1+DistMax(Y,2)*.

These values (*NumPeers(X,Y)* and *Dmax(X,Y)*) are only estimates of the total number of peers in *T(X,Y)* and of the longest path in *T(X,Y)* starting at *Y*, because they are not immediately updated whenever a new peer joins the system or an old peer leaves the system. However, we will show that, if no peer joins or leaves the system, these values converge to the actual correct values after a number of gossiping periods which is proportional to the diameter of the tree. In order to present the proof, we will define the concept of *layer of leaves*. A leaf in the tree is a vertex with degree *1*. *L(1)* is the set of all the leaf nodes of the tree. *L(i≥2)* is the i^{th} layer of leaves, composed of those nodes which become leaves in the tree if we remove all the nodes in the sets *L(j)* (*1≤j≤i-1*). We assume that the tree has *LL* layers of leaves. It is well-known that the last layer, *L(LL)*, contains only one or two adjacent nodes (the center or the bi-center of the tree); *LL* is equal to *(D+1)/2*, where *D* is the diameter of the tree (length of the longest path in the tree, expressed in terms of tree edges). If *L(LL)* contains two nodes *A* and *B*, we will add an extra layer *LL+1* and move one of the nodes (*A* or *B*) to that extra layer (and then set *LL=LL+1*). Thus, we will consider that *L(LL)* contains only one node.

The values *NumPeers(*,*)* and *Dmax(*,*)* converge to the corresponding correct values in *O(D)* gossiping periods. We will first show that the values *NumPeers(X,Y)* and *Dmax(X,Y)*, with *X* located on a layer *Q* higher than the layer of *Y*, converge to the correct values in at most *Q-1* gossiping periods. We will prove this by induction on the layer number of the peer *X*. The assumption is true for all the peers *X* in *L(1)*, because they have no neighbor *Y* located on a lower layer. Let's assume now that the proposition is true for all the peers on the layers *1,…,i* and we will prove it for the layer *i+1*. Peer *X* from *L(i+1)* receives the information from a peer *Y* in *L(j)* (*j≤i*). The value *NumPeers(X,Y)* sent by peer *Y* to peer *X* is equal to the sum of the values *NumPeers(Y,W)*, with *W≠X*. Due to the properties of any tree graph, peer *Y* can have only one neighbor on a layer of leaves with an index higher than *j*; this neighbor is *X*. Thus, all the other neighbors *W* are located on layers which are lower than *j* and, by the induction hypothesis, the values *NumPeers(Y,W)* become correct in less than *i* periods. As a consequence, the value *NumPeers(X,Y)* will become correct at the next gossiping period. The same holds for *Dmax(X,Y)*, which is equal to *1+max{Dmax(Y,W)| W≠X* is a neighbor of peer *Y}*, because the values *Dmax(Y,W)* become correct in at most *i-1* periods.

After all the values *NumPeers(X,Y)* and *Dmax(X,Y)* with peer *Y* located on a lower layer of leaves than peer *X* become correct, we will prove that all the values *Dmax(Y,X)* become correct in at most *LL-j* extra periods, where Y belongs to *L(j)*. We will prove this in decreasing order of the index of the layer of leaves of the peer *Y*. For the single peer *A* in *L(LL)* this is true, because it has no neighbors located on a higher layer. Let's assume that the proposition is true for all the peers located on the layers of leaves *LL, LL-1, …, i*. We will now show that the values of all the peers on the layer *i-1* become correct after (at most) *LL-i+1* extra periods. Let's consider a peer *X* from *L(i-1)* and a neighboring peer *Y* from *L(j)* (*j≥i*). Peer *X* receives the values *NumPeers(X,Y)* and *Dmax(X,Y)* from peer *Y*. *NumPeers(X,Y)* (*Dmax(X,Y)*)is computed based on the values *NumPeers(Y,W)* (*Dmax(Y,W)*), with *W≠X*. From the induction hypothesis, all the values *NumPeers(Y,*)* and *Dmax(Y,*)* are correct after *LL-i* extra periods. Thus, *NumPeers(X,Y)* and *Dmax(X,Y)* will be correct at the next gossiping period. This concludes our proof.

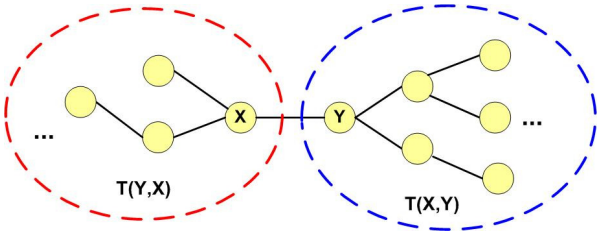


Fig. 6-1. T(X,Y) and T(Y,X) for 2 Neighbouring Peers X and Y.

6.1.2. Joining the Multicast Tree

When a new peer *X* wants to join the multicast tree, it must know how to contact any other peer *Y* which is already part of the tree (the peer *Y* can be any peer already in the tree). During the joining procedure, peer *X* will be gradually redirected to other peers until it reaches a peer to which it will connect in the tree. Whenever peer *X* contacts a new peer *Y* in order to join the tree, it will also tell peer *Y* which other peer *Z* redirected peer *X* to peer *Y* (peer *Z* will be a neighbor of peer *Y*).

At the initial join contact, the previous peer *Z* will be undefined. Let's assume that peer *X* contacted a peer *Y* in order to join the system and was redirected here by peer *Z* (or by nobody if this is the first join attempt, in which case *Z* is undefined). Peer *Y* will consider all of its (at most) *K* tree neighbors *W*, except for *W=Z*. For each neighbor peer *W≠Z*, peer *Y* knows the estimates *NumPeers(Y,W)* and *Dmax(Y,W)*. Then, peer *Y* computes the maximum number of peers *MaxNumPeers(Y,W)* which can be located in *T(Y,W)* such that *Dmax(Y,W)* does not increase. *MaxNumPeers(Y,W)* is equal to $1+(K-1)+(K-1)^2+\ldots+(K-1)^{Dmax(Y,W)-1}$. If *K=2*, then *MaxNumPeers(Y,W)=Dmax(Y,W)*; else, $MaxNumPeers(Y, W)=((K-1)^{Dmax(Y,W)}-1)/(K-2)$. If *MaxNumPeers(Y,W)>NumPeers(Y,W)*, then peer *W* is a *valid* neighbor; otherwise, it is not valid.

Among all of peer *Y*'s valid neighbors *W≠Z*, peer *Y* will choose the peer W_{next} as the one with the smallest value $Dmax(Y,W_{next})$ (if there are multiple such neighbors, one will be chosen arbitrarily). Peer *X* will be redirected to the peer W_{next}. If peer *Y* has no valid neighbors and peer *Y*'s degree is less than *K*, then peer *Y* will connect directly to peer *X*. Peer *X*'s degree will now be *1* (it will be a leaf in the tree) and peer *Y*'s degree increases by *1*. *NumPeers(Y,X)* and *Dmax(Y,X)* will be *1*; *Dmax(X,Y)* and *NumPeers(X,Y)* will be sent immediately to peer *X* (they will be computed as described in subsection 2.1). If peer *Y* has no valid neighbors and its degree is equal to *K*, then it will choose the neighbor $W_{next}≠Z$ with the smallest value $Dmax(Y,W_{next})$ (disregarding the values $NumPeers(Y,W_{next})$ and $MaxNumPeers(Y,W_{next})$). Peer *X* will be redirected to peer W_{next}. If peer *X* was redirected to another peer W_{next}, at the next join request peer *X* will contact peer W_{next} and will tell it that it was redirected there from peer *Y*. We can see that peer *X* may be redirected (at most) a number of times proportional to the diameter of the tree.

6.1.3. Leaving the Multicast Tree

When a peer *X* leaves the multicast tree (gracefully or suddenly), its tree neighbors will detect this event (because every neighbor periodically sends *keep-alive* and *ping* messages to both its neighbors and its 2-neighbors). Because of the first type of gossiping messages, every neighbor *Y* of peer *X* knows every other neighbor of peer *X*. Every neighbor *Y* will compute the value *DistMaxNoX(Y,X)*; if *DistMaxNeigh(Y,1)=X*, then *DistMaxNoX(Y,X)=DistMax(Y, 2)*; otherwise, *DistMaxNoX(Y,X)=DistMax(Y,1)*. Each (former) neighbor *Y* of peer *X* will send the value *DistMaxNoX(Y,X)* to every other (former) neighbor of *X*, as well as a unique, self-generated identifier (e.g. the result of a hash function). The (former) neighbor *W* of peer *X* with the largest value *DistMaxNoX(W,X)* will be chosen by every other (former) neighbor as their representative (if multiple neighbors *Z* have the same largest *DistMaxNoX(Z,X)* value, ties will be broken by considering the unique identifiers of the peers; e.g. the peer with the smallest or largest identifier will be chosen).

From a practical point of view, each (former) neighbor *Y* of peer *X* will wait at most a certain amount of time for receiving the corresponding values from any other (former) neighbor *Y'* of peer *X*. Since 2-neighbors periodically ping each other, peer *Y* can have a good estimate of the

latency *lat* of the network path to any 2-neighbor *Y'*; peer *Y* can wait for the information from *Y'* for at most $C \cdot lat$ time units, where $C \geq 2$ is a constant.

Peer *W* will send a message to the peer *Q* for which the path from *W* to *Q* in the tree contains exactly *DistMax(W,1)* peers (if *DistMax(W,1)* has converged to the correct value). Peer *W* does not need to know peer *Q* before-hand. Peer *W* will forward the message to its neighbor $W' \neq X$ with the largest value *Dmax(W,W')*. Whenever a peer *W'* receives the message from a peer *W''*, it will forward it to the neighbor $W''' \neq W''$ with the largest value *Dmax(W', W''')*. Note that the neighbor $A \neq V$ of a peer *B* with the largest value *Dmax(B,A)* can be computed in *O(1)* time: if $DistMaxNeigh(B,1) \neq V$, then $A = DistMaxNeigh(B,1)$; otherwise, $A = DistMaxNeigh(B,2)$. Eventually, the message will reach a peer *Q* which is a leaf in the tree and, thus, cannot forward the message further. If all the values *Dmax(*,*)* have converged to their stable states, then the path from peer *W* to peer *Q* is the longest path from peer *W* to any peer in its part of the tree (*T(X,W)*); otherwise, this path is only an approximation of the actual longest path (although we may obtain the longest path even if the *Dmax(*,*)* values have not converged, yet).

Peer *Q* will disconnect from its only neighbor in the tree (if the representative peer *W* had no other neighbors except peer *X*, then $Q = W$ and no disconnection is performed) and will replace peer *X*; that is, peer *Q* will connect to all the former neighbors of peer *X*. Thus, after a peer *X* departs from the tree, the tree returns to a correct structure after a number of time steps which is proportional to $K+D$, where *D* is the diameter of the tree. After connecting to all the former neighbors *Y* of peer *X*, peer *Q* receives the values *NumPeers(Q,Y)* and *Dmax(Q,Y)* from these neighbors. As soon as it receives all of these values, peer *Q* will send back the values *NumPeers(Y,Q)* and *Dmax(Y,Q)* to every neighbor *Y* (all these values are computed the way we showed in a previous subsection).

In order to minimize the period of time during which the tree remains disconnected after the departure of a peer *X*, we can use a proactive approach, instead of the reactive approach presented above. Every peer *Y* periodically computes the values *DistMaxNoX(Y,Z)* (as described previously) and $Q_{far}(Y,Z)$=the peer *Q* which would be chosen by the previously described method, if the neighbor *Z* of peer *Y* were to leave the tree. Thus, if peer *Y* is chosen as the representative peer among all the neighbors of a departed peer *X*, then the peer *Q* which will replace *X* is $Q_{far}(Y,X)$. Moreover, every *2* neighbors *Y* and *Z* of a peer *X* could periodically exchange between them the values *DistMaxNoX(Y,X)* and *DistMaxNoX(Z,X)* (together with their identifiers). This way, when a peer *X* leaves the tree, every former neighbor *Y* of peer *X* already knows the values *DistMaxNoX(Z,X)* of all the other former neighbors *Z* of peer *X* and can immediately select the representative (former) neighbor *W*. With this proactive approach, the tree stays disconnected only for a very short time (*O(1)* time steps) whenever a peer *X* leaves the tree.

6.1.4. Experimental Tests

In order to test the multicast tree peer-to-peer topology, we developed a simulation framework, which we implemented in the Python programming language. We performed two types of tests. The first tests were *incremental* tests. *600* peers were added sequentially, at different rates, and considering two values of *K* (*3* and *6*). The rate was measured as the number of newly added peers divided by the number of gossiping periods. We measured the tree's diameter after every peer addition. The lowest rate was *1/D*, where *D* was the (current) diameter of the tree; obviously, this rate was not constant. At this rate, all the *NumPeers(*,*)* and *Dmax(*,*)* values became correct before the next peer addition. The consequence was that the diameter of the obtained tree was always equal to the theoretical optimum (i.e. the diameter of a perfectly balanced tree with the same number of nodes as the multicast tree and with the same upper bound on the node degrees).

We considered both the case when every peer started its joining process from a random peer and the case when all the peers started from the same (first) peer. The same cases were considered for other rates: *2/D*, *1*, *2* and *5*. As expected, the higher the rate, the higher the tree's diameter was (however, there was no difference in the diameters for the rates *1/D* and *2/D*). Fig. 6-2 (right) presents the diameters obtained for $K=3$ and different ratios, as a function of the number of peers,

when every peer joined the tree starting from another random peer. The results for the case when every peer started the joining process from the same peer are similar. Fig. 6-2 (left) shows the obtained tree topology for *K=3* and *100* peers.

The tests of the second type were *decremental*. We started from the tree with *600* peers and optimal diameter and repeatedly removed from the tree the peer *X* whose largest estimate *Dmax(X,*)* was minimum (i.e. the tree's center). The tree recovered gracefully every time and maintained the optimal theoretical diameter after every peer removal. A peer was removed only after the tree recovered correctly from the previous peer removal.

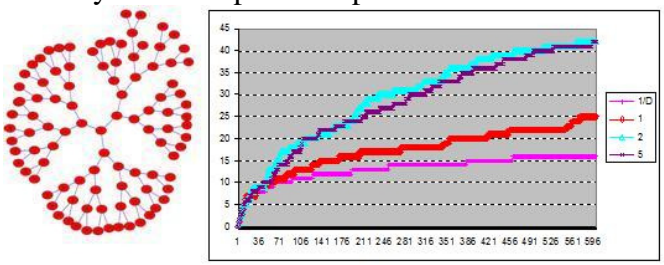


Fig. 6-2. Left - Multicast Tree with 100 Peers (K=3). Right - Tree Diameter after Every Peer Insertion at Different Peer Insertion Ratios.

6.2. Maximum Reliability K-Hop Multicast Strategy in Tree Networks

The reliability of network nodes and links is an important aspect which needs to be considered when developing fault-tolerant distributed algorithms. Usually, the reliability is only a statistical measure, representing the probability that the network node/link will **not** fail. In this section, we consider the reliability of network links in a tree network, in the context of developing a multicast content distribution strategy with the highest reliability, subject to restrictions regarding the number of intermediate hops.

The problem is defined as follows. We are given a directed tree T with n nodes, in which the root of the tree wants to distribute some content to a set of destinations, which are the leaves of the tree. In order to send the content from a node u to a node v located in the subtree of u, the node u establishes a direct connection to node v and sends a message with the content on that connection. The transmission lasts for a fixed amount of time (one time unit). A node may establish any number of simultaneous connections to other nodes in its subtree.

Each edge (u,v) of the tree has an associated reliability $r_{u,v}$. The reliability of transmitting a message on a direct connection from u to v is equal to the product of the reliabilities of the edges on the path from u to v. The reliability of the content distribution strategy is the product of the reliabilities of all the message transmissions performed. We are interested in finding a multicast strategy having the maximum reliability, subject to the constraint that it should not last for more than $k \geq 1$ time units. It is obvious that the root of the tree can send the content to every leaf during a single time unit, but the reliability of this strategy is

$$\prod_{(u,v)\in T} r_{u,v}^{nr\_leaves(v)} \qquad \textbf{(6-1)}$$

where *nr_leaves(v)* is the number of leaves located in the subtree rooted at v.

By using intermediate nodes, the reliability of the strategy can be improved. We will now define the *k-hop multicasting problem*. We will build $k+1$ sets of nodes: $S_0, S_1,...,S_k$. The root is the only node in S_0. In the first time unit, the root sends a message to a subset of nodes S_1 such that each leaf is a descendant of exactly one node X in S_1. During the i^{th} time unit ($2 \leq i \leq k-1$), each node X in S_{i-1} which is not a leaf, sends a message to a subset $S_{i,X}$ of nodes from its subtree, such that each leaf which is a descendant of X either belongs to $S_{i,X}$ or is also a descendant of exactly one node in $S_{i,X}$. The set of nodes S_i is the union of the sets $S_{i,X}$, for each X in S_{i-1}. During the k^{th} time unit, each node X in S_{k-1} which is not a leaf must send a message to each leaf node which is a descendant of X (the leaves receiving the message in the k^{th} time unit form the set S_k). The nodes belonging to the union of the sets S_i are called intermediate nodes. Every intermediate node (except the root) receives the

content from exactly one other intermediate node. Obviously, there can be many multicast strategies and we are interested in the one with the maximum reliability.

By replacing the reliability of each edge *(u,v)* by $cost(u,v)=-log(r_{u,v})$, the requirement to maximize the reliability becomes equivalent to minimizing the total cost of message transmissions, where the cost of sending a message is equal to the sum of the costs of the edges composing the connection along which the message is sent.

6.2.1. An O(k·n^3) Dynamic Programming Algorithm

First, we will transform the directed tree into a binary directed tree. This transformation is quite standard. For each node *i* having $q>2$ sons $s_1,s_2,...,s_q$, we keep his first son s_1 and insert an extra node *x* as his second son. We make $s_2,...,\ s_q$ the sons of *x* and then recursively repeat the procedure for the node *x* and for the son s_1. If $q\leq 2$ we call the procedure for each of the sons $s_1, ..., s_q$. The edge between *i* and *x* will have cost *0* (or equivalently, reliability *1*).

With this modified tree, we will compute the following values in a bottom-up fashion: *C(i,j,p)* = the minimum cost of distributing the content to all the leaves in the subtree of node *i*, using at most *j* ($0\leq j\leq k$) time units and considering only the edges in vertex *i*'s subtree when computing the cost and:

- if $p>1$ then *i* is on the paths between an intermediate node located above *i*, and all the *p* intermediate nodes to which that node sends messages directly.
- if $p=1$, then either *i* is an intermediate node or *i* is on the path between an intermediate node located above it and the only node to which that node sends a message directly (located below *i*).

For all nodes *i*, we define $C(i,-1,p)=+\infty$. If *i* is a leaf, then for all $0\leq j\leq k$ we have $C(i,j,1)=0$ and $C(i,j,p)=+\infty$ (for p>1). If *i* is not a leaf, then it has either one or two sons. If it has only one son *s*, we have the following equations:

- $$C(i,j,p) = p\cdot cost(i,s) + C(s,j,p), p>1 \quad \textbf{(6-2)}$$
- $$C(i,j,1) = \min\{\min_{1\leq p1\leq n}\{p1\cdot cost(i,s) + C(s,j-1,p1)\}, cost(i,s) + C(s,j,1)\}$$

The first case is straightforward: *i* is not an intermediate node, therefore it lies on the paths between *p* intermediate nodes below it and another intermediate node above it. Therefore, *i* will have to forward *p* messages to the *p* intermediate nodes on the edge *(i,s)*. In the second case, either *i* is an intermediate node and consumes one time unit and the number of intermediate nodes to which *i* sends a message directly can be any number p_1, or *i* is not an intermediate node, and we find the same situation as before.

If *i* has two sons s_1 and s_2, we have the equations:

- $$C(i,j,p) = \min_{1\leq p1<p}\{p1\cdot cost(i,s1) + C(s1,j,p1) + (p-p1)\cdot cost(i,s2) + C(s2,j,p-p1)\}, p>1 \quad \textbf{(6-3)}$$
- $$C(i,j,1) = \min_{1\leq p1,\ 1\leq p2,\ p1+p2\leq n}\{p1\cdot cost(i,s1) + C(s1,j-1,p1) + p2\cdot cost(i,s2) + C(s2,j-1,p2)\}$$

In the first case, *i* is not an intermediate node, and all the *p* messages coming from the intermediate node closest to *i* and above it will be forwarded to the *p* intermediate nodes below it. Out of these, p_1 intermediate nodes are located in s_1's subtree and $p-p_1$ are located in s_2's subtree. In the second case, either *i* is an intermediate node and consumes one time unit and there are p_1 intermediate nodes located in s_1's subtree and p_2 in s_2's subtree, to which *i* will send the message directly, or *i* is not an intermediate node and we are in the same situation as in the first case.

If *i* is the root of the tree and has only one son *s*, then the only entry defined is:

$$C(root,k,1) = \min_{1\leq p1\leq n}\{p1\cdot cost(root,s) + C(s,k-1,p1)\}. \quad \textbf{(6-4)}$$

If the root has two sons s_1 and s_2, then the only entry defined is *C(root,k,1)*. This entry represents the minimum cost of the k-hop multicast strategy. In order to actually find the strategy, we can trace the way the *C(i,j,p)* entries were computed and for each intermediate node we can find out to which other intermediate nodes it sends a message directly.

Let's analyze the time complexity of the algorithm. The most complex case is when a node

has two sons, which takes $O(k \cdot n^2)$ time. Thus, the overall complexity is $O(k \cdot n^3)$.

6.2.2. An O(k·n²) Dynamic Programming Algorithm

We will present now a dynamic programming solution with a better time complexity. First, we assign a label from *1* to *M* to each leaf of the tree, where *M* is the total number of leaves. The j^{th} leaf visited by a depth-first traversal of the tree (starting from the root) receives the label *j*. It is obvious that the labels of the leaves located in the subtree of a node *i* (including node *i* itself) form an interval of consecutive values, denoted by *[lmin(i), lmax(i)]*. This interval can be computed in $O(n)$ time for all the nodes, with a simple bottom-up traversal. We will also compute in $O(n^2)$ time the values *dist(i,j)*=the sum of the costs of the edges on the directed path from *i* to *j*.

We will now compute the values *C(i,j,p)*=the minimum cost for distributing the content to the first *p* leaves in node *i*'s subtree (denoted by ST_i), using at most *j* time units, with *i* being an intermediate node. The first *p* leaves are the leaves labeled *lmin(i), …, lmin(i)+p-1*. If *i* is a leaf with label *q*, then *C(i,j,1)=0* for all the values of *j* and *lmin(i)=lmax(i)=q*. If *i* is not a leaf, then we have:

- $C(i,j,0)=0, 0 \le j \le k\,; C(i,0,p)=+\infty, 1 \le p \le nr\_leaves(i)$ **(6-5)**
- $$C(i,j,p) = \min_{\substack{1 \le j \le k,\ p \ge 1 \\ x \in ST_i \\ lmax(x)=lmin(i)+p-1}} \left\{ \begin{array}{c} C(x, j-1, nr\_leaves(x)\} + \\ dist(i,x) + C(i, j, lmin(x) - lmin(i)) \end{array} \right\}$$

The last equation considers the following case: node *i* sends a message to a node *x* in ST_i with *lmax(x)* equal to the label of the p^{th} leaf in ST_i, letting *x* take care of sending the content further to all the leaves in ST_x, using at most *j-1* time units; node *i* takes care of the *lmin(x)-lmin(i)* remaining leaves (with labels in *[lmin(i), lmin(x)-1]*), using at most *j* time units. Each of the $O(n)$ nodes *x* in ST_i must only be considered for only one of the $O(n)$ values of *p*, equal to *lmax(x)-lmin(i)+1*. Thus, as a preprocessing step, for each node *i*, we will compute in $O(n)$ time an array of lists *L(i)*, where *L(i,p)* is a list containing all the nodes *x* in ST_i with *lmax(x)=lmin(i)+p-1*. When computing *C(i,j,p)*, only the nodes *x* in *L(i,p)* will be considered. This way, we can compute the values *C(i,j,p)* in $O(n)$ time for a given pair *(i,j)* and all the values of *p* (and in $O(1)$ amortized time for each tuple *(i,j,p)*, if we also do not add to *L(i,p)* any node *x* in ST_i with exactly one son). The time complexity of the algorithm is $O(k \cdot n^2)$ and the minimum cost is *C(root,k,M)*.

An example of an optimal strategy computed using the previously described algorithms is presented in Fig. 6-3.

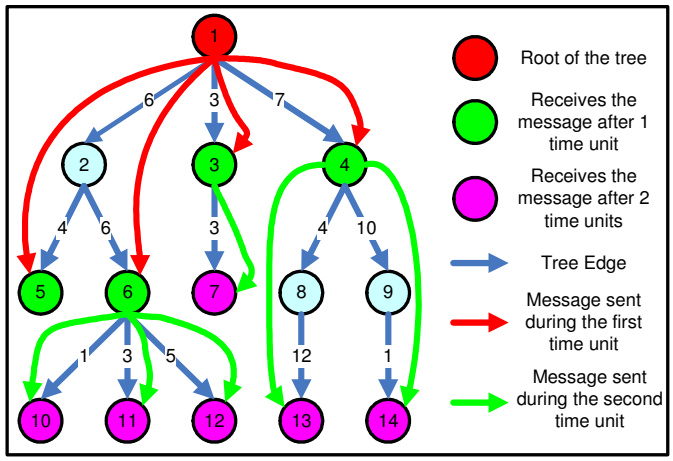


Fig. 6-3. Example of an Optimal K-Hop Multicast Strategy (K=2).

6.3. Send- and Receive-Constrained Broadcast in Tree Networks

In this section we present novel models and algorithmic solutions for several constrained broadcast problems in tree networks; we extend the single port broadcast model in trees by adding sending and receiving constraints.

6.3.1. Minimum Time Broadcast in Trees with Sending Constraints

We are given a tree network with n vertices (numbered from *1* to n). A source node *src* needs to distribute a piece of content to all the other vertices of the tree. In order to do this, it will use a broadcast strategy. At each moment t, the vertices can be partitioned into two sets A_t and B_t. The vertices in the set A_t have already received the piece of content, while those in B_t did not. Each vertex in the set A_t can send the piece of content to at most one neighboring vertex belonging to the set B_t. Transmitting the content takes one time unit. Assuming that the vertices receiving the content sent at time t form the set R_t, at time moment $t+1$, we have: $A_{t+1}=A_t \cup R_t$ and $B_{t+1}=B_t \setminus R_t$. Initially (at $t=0$), $A_0=\{src\}$ and $B_0=\{1,2,\ldots,n\}\setminus\{src\}$. The first time moment T when $A_T=\{1,2,\ldots,n\}$ and $B_T=\phi$ is equal to the duration after which every vertex of the tree receives the piece of content (the broadcast time). Obviously, T depends on the sets R_t ($t=0,1,\ldots,T-1$), chosen by the broadcast strategy.

We are interested in finding a broadcast strategy with a minimum broadcast time. When there are no other constraints, this problem is well-known and an optimal algorithm was provided many years ago [Slater, Cockayne and Hedetniemi, 1981]. We will briefly present this algorithm. The tree is rooted at the source node *src*, thus defining parent-son relationships. We compute the values $T_{min}(i)$ in a bottom-up fashion, for each vertex i, where $T_{min}(i)$=the minimum broadcast time for sending the piece of content from vertex i to all the vertices in its subtree. For a leaf i, $T_{min}(i)=0$. The optimal broadcast strategy of a vertex i having $ns(i)\geq 1$ sons consists of sending the piece of content to a different son during each of the first $ns(i)$ time moments. Assuming that the sons are $s(i,1)$, $s(i,2)$, …, $s(i,ns(i))$, in the order in which vertex i sends the content to them, the broadcast time is $max\{1+T_{min}(s(i,1)), 2+T_{min}(s(i,2)), \ldots, ns(i)+T_{min}(s(i,ns(i)))\}$. The ordering of the sons which minimizes the broadcast time has the following property: $T_{min}(s(i,1)) \geq T_{min}(s(i,2)) \geq \ldots \geq T_{min}(s(i,ns(i)))$. A straight-forward implementation of this algorithm takes $O(n \cdot log(n))$ time (because of the step where the sons need to be sorted).

In this section we consider the following extension of the problem. We are given a time duration TM and for each vertex i and each time moment t in $\{0, 1, \ldots, TM-1\}$, we are given a binary value $sendb(i,t)$. If $sendb(i,t)=1$, then the vertex i is blocked at time moment t, i.e. it cannot send anything to any neighboring vertex; if $sendb(i,t)=0$, then vertex i is not blocked and can send the piece of content to a neighboring vertex at time t (if vertex i belongs to the set A_t). We consider two cases:

(1) at any time moment $t\geq TM$, no vertex is blocked (we have $sendb(i,t)=0$ for $t\geq TM$);

(2) $sendb(i,t)=sendb(i,t-TM)$, $t\geq TM$ (the constraints are periodical).

The motivation for this extension is given by several factors. The vertices of the tree may be represented by computers which undergo some specific maintenance procedures which temporarily disrupt the functionality of the sending interface. When the vertices have asymmetric upload and download bandwidths, it is possible that at certain time moments, the entire upload bandwidth is used by another application, while enough download bandwidth is still available for receiving the content. We will present exact, efficient algorithms for this problem, using dynamic programming and greedy techniques.

6.3.1.1. A Dynamic Programming Algorithm

We will root the tree at the source vertex *src* and we will compute a table $T_{min}(i,t)$=the minimum time moment when every vertex in vertex i's subtree has received the content, considering that vertex i received the piece of content at time moment t. $T_{min}(src,0)$ will represent the minimum time duration after which every vertex of the tree receives the content (i.e. the minimum duration of the broadcast strategy). We algorithmically compute the table for $t<TM$. In case *(1)*, for $t=TM$ we can compute $T_{min}(i,TM)$ using the standard greedy algorithm we described (because there are no constraints) and $T_{min}(i,t>TM)=T_{min}(i,TM)+t-TM$. In case *(2)*, $T_{min}(i,t\geq TM)=T_{min}(i,t \bmod TM)+(t \operatorname{div} TM)\cdot TM$, where we denote by $(A \operatorname{div} B)$ the integer part of the division of A by B and by $(A \bmod B)$ the remainder of the division. Based on these values, the optimal broadcast strategy can be easily obtained.

We will traverse the tree in a bottom-up fashion (from the leaves towards the root) and

compute all the required values for a vertex i. If i is a leaf, then $T_{min}(i,t)=t$. Otherwise, let's consider $s(i,1)$, $s(i,2)$, …, $s(i,ns(i))$, the $ns(i)$ sons of vertex i. Since there are no receiving constraints, the optimal broadcast strategy requires that vertex i sends the content to its sons at the first $ns(i)$ time moments when its sending capabilities are not blocked. Assuming that vertex i receives the message at time moment t (and we want to compute $T_{min}(i,t)$), we will determine the time moments $t \le ts(i,t,1) < ts(i,t,2) < \ldots < ts(i,t,ns(i))$, such that $sendb(i,ts(i,t,j))=0$ and $sendb(i,t')=1$, for $ts(i,t,j) < t' < ts(i,t,j+1)$, $0 \le j \le ns(i)-1$ (with $ts(i,t,0)=t-1$). We can easily determine these time moments in $O(TM+ns(i))$ time: by inspecting all the time moments t'', starting from t and ending when $ns(i)$ time moments with $sendb(i,t'')=0$ were found, or when $t'' \ge TM$ (and $ns'(i) < ns(i)$ moments were found) - in case *(1)*, we add the moments TM, …, $TM + ns(i) - ns'(i) - 1$; in case *(2)*, we can obtain $P \cdot ns'(i)$ extra moments for any $P \ge 1$, by shifting the first $ns'(i)$ moments by a multiple of TM.

Once the time moments are decided, all we need to do is establish the order in which the sons will receive the content from vertex i. We will solve the following problem first: we will choose an upper limit T_{max} for the value of $T_{min}(i,t)$ and verify whether a valid ordering of the sons exists, such that $T_{min}(i,t) \le T_{max}$. It is obvious that the $T_{min}(v,*)$ values of a vertex v are non-decreasing, i.e. $T_{min}(v,t') \le T_{min}(v,t'+1)$. We will compute for each son $s(i,j)$ of the vertex i the largest time moment $tl(s(i,j),T_{max})$, such that $T_{min}(s(i,j),tl(s(i,j),T_{max})+1) \le T_{max}$, i.e. $tl(s(i,j),T_{max})$ is the largest time moment at which vertex i can send the content to the son $s(i,j)$, such that every vertex in vertex $s(i,j)$'s subtree is still able to receive the piece of content by the time moment T_{max} (or 0 if such an index does not exist). We can compute $tl(s(i,j),T_{max})$ for a son $s(i,j)$ in $O(log(TBOUND))$ time, using a binary search and the afore-mentioned property of the $T_{min}(v,*)$ values: *(1)* $TBOUND=n+TM-1$; *(2)* $TBOUND=n \cdot TM-1$. We then sort all the sons $s(i,j)$ in non-decreasing order of the $tl(s(i,j),T_{max})$ values, i.e. we will have $tl(s(i,1),T_{max}) \le tl(s(i,2),T_{max}) \le \ldots \le tl(s(i,ns(i)),T_{max})$.

The order in which the sons will receive the content from vertex i will be exactly this order of the $tl(s(i,j),T_{max})$ values. This ordering is valid if $tl(s(i,j),T_{max}) \ge ts(i,t,j)$, for all the values of j ($1 \le j \le ns(i)$). If we first initialize $T_{min}(i,t)$ to $+\infty$ and then binary search the smallest value of T_{max} such that there exists a valid ordering for the sons of the vertex i, we have already obtained an algorithm which solves our problem, but its time complexity is too high. We will successively improve this algorithm. First, we will improve the part where the values $ts(i,t,j)$ are computed for a vertex i and a receiving time moment t. For $t=0$, we will use the presented approach. As we move from the time moment t to $t+1$, we have the following situations:

- $t < ts(i,t,1)$: in this case, $ts(i,t+1,j)=ts(i,t,j)$ ($1 \le j \le ns(i)$) and we do not need to perform other computations.
- $t=ts(i,t,1)$: in this case, $ts(i,t+1,j)=ts(i,t,j+1)$ ($1 \le j \le ns(i)-1$) and we just need to search for the value $ts(i,t+1,ns(i))$ – we will inspect all the time moments starting from $ts(i,t,ns(i))+1$, until we find the first time moment t' such that $sendb(i,t')=0$ (we test at most the next TM moments).

It is easy to notice that we inspect $O(TBOUND)$ time moments overall, for all the values of t. Thus, we obtain all the values $ts(i,t,j)$ in $O(TBOUND/TM)$ amortized time for each pair (i,t). The time complexity is now $O(n \cdot TBOUND)+n \cdot TM \cdot log(TBOUND) \cdot (log(TBOUND)+log(n)))$.

The following changes constitute improvements only in some cases. We will replace the binary search for the values of $T_{min}(i,t)$ with a linear search. When computing $T_{min}(i,0)$, we will start from $T_{max}=0$ and increase it by 1, until we find a valid ordering. For $t>0$, we will start the linear search from $T_{max}=T_{min}(i,t-1)$ and increase T_{max} until we find a valid ordering. We notice that we only perform $O(TBOUND)$ tests for all the $O(TM)$ values of t (and a fixed vertex i). This way, we perform an $O(TBOUND/TM)$ amortized number of tests for each pair (i,t). The time complexity becomes $O(n \cdot TBOUND \cdot (log(TBOUND)+log(n)))$.

Once we replaced the binary search for T_{max} with a linear search, we can replace the binary search for determining the value $tl(s(i,j),T_{max})$ of each son with a linear search, as well. When we move from a candidate value T_{max} to the next candidate value $T_{max}+1$, we linearly search for the values $tl(s(i,j),T_{max}+1)$ starting from $tl(s(i,j),T_{max})$. Using the same arguments as before, we obtain an $O(1)$ amortized complexity for computing $tl(s(i,j),T_{max})$ for each tuple (i,j,T_{max}), thus reaching a complexity of $O(n \cdot TBOUND \cdot log(n))$. We will now sort the sons $s(i,j)$ of a vertex i according to their $tl(s(i,j),T_{max})$ values using a variation of *countsort*. The values $tl(s(i,j),T_{max})$ belong to the interval $[0,TBOUND]$. We could use a linked-list $LL(i,t')$ for each time moment t' and insert a son $s(i,j)$ into

LL(i, tl(s(i,j), T_{max})). Then, by traversing the linked-lists of all the time moments between *t* and *TBOUND* (when computing $T_{min}(i,t)$), we can sort the sons linearly in the number of time moments.

However, this does not really constitute an improvement, because we might need to traverse many time moments. Instead, we will compute a list of the time moments in *[0,TBOUND]* at which vertex *i* can send messages: $0 \le tcs(i,1) < tcs(i,2) < \ldots < tcs(i,ntcs(i))$, where *ntcs(i)* is the total number of such moments (we can compute and store only *ntcs'(i)=O(TM)* values of *tcs(i),* where *ntcs'(i)*= the number of time moments in *[0,TM-1]* when *i* can send the content:

(1) tcs(i,j>ntcs'(i))=TM+(j-1-ntcs'(i));

(2) tcs(i,j>ncts'(i))=tcs(i,1+((j-1) mod ntcs'(i))+((j-1) div ntcs'(i))·TM).

We will redefine the values *ts(i,t,j)* and the values $tl(s(i,j),T_{max})$ as indices into the *tcs(i)* list. Thus, *ts(i,t,j)* is an index to the time moment *tcs(i,ts(i,t,j))* in the list *tcs(i)* (thus, we can find *ts(i,t+1,ns(i))* in *O(1)* time when moving from *t* to *t+1* and *t=ts(i,t,1)*, as *ts(i,t,ns(i))+1*). Similarly, $tl(s(i,j), T_{max})$ will be the index of the largest time moment in the list *tcs(i)* such that $T_{min}(s(i,j), tcs(i, tl(s(i,j), T_{max}))+1) \le T_{max}$ (or *0* if such an index does not exist).

We will use a linked-list *LL(i, tidx)* for each index *tidx* in the list *tcs(i)* (plus the index *0*) and insert each son *s(i,j)* into $LL(i, tl(s(i,j),T_{max}))$. Then, we will traverse all the linked-lists *LL(i,tidx)*, with *tidx=ts(i,t,j)* ($1 \le j \le ns(i)$) (*ts(i,t,j)* are consecutive indices in the list *tcs(i)*). There are *ns(i)* such linked-lists, so we will sort the *ns(i)* sons in *O(ns(i))* time. If some sons were inserted in *LL(i,tidx>ts(i,t,ns(i)))*, they will be placed in any order at the end of the list of sorted sons. If some sons were inserted in *LL(i,tidx<ts(i,t,1))* or the obtained ordering of the sons is not valid, then we need to test a larger value of T_{max}. The final complexity is O($n \cdot TBOUND$). If we use the linear son sorting method, we binary search T_{max} and $tl(s(i,*),T_{max})$ in the *tcs(i)* list, and find in *O(1)* time the values *ts(i,t+1, ns(i))*, we get an $O((n+n \cdot TM) \cdot \log^2(TBOUND))$ algorithm.

6.3.1.2. A Greedy Algorithm

A greedy algorithm also exists for this problem. We will binary search for the minimum duration of broadcasting the piece of content from the source vertex *src* to all the other vertices. Let's assume that we chose a value T_{max}. We now need to perform a feasibility test. If T_{max} is feasible, we will choose a smaller value in the binary search; otherwise, we will choose a larger value. The feasibility test consists of computing the following values for each vertex: $T_{latest}(i)$=the latest time moment at which vertex *i* can receive the piece of content such that all the vertices in vertex *i*'s subtree can receive the content by time T_{max}.

We will traverse the tree bottom-up, from the leaves towards the root. For a leaf vertex *i*, we have $T_{latest}(i)=T_{max}$. For a non-leaf vertex *i*, let's consider its *ns(i)* sons *s(i,1), s(i,2), …, s(i,ns(i))*, sorted such that: $T_{latest}(s(i,1)) \ge T_{latest}(s(i,2)) \ge \ldots \ge T_{latest}(s(i,ns(i)))$. The son *s(i,1)* will be the last one to receive the content from vertex *i*, the son *s(i,2)* will be the one before the last and so on. We will consider all the time moments from $T_{latest}(s(i,1))-1$ down to *0* and, for each son *s(i,j)*, we will find the latest time moment *tsend(i,j)* at which vertex *i* can send the content to the son *s(i,j)*. If we cannot find such a time moment for every son, then the feasibility test will fail (T_{max} is not a feasible value). Otherwise, $T_{latest}(i)=tsend(i,ns(i))$.

The time complexity of the feasibility test is $O(n \cdot T_{max})$, where T_{max} is binary searched between *0* and *TBOUND*. If we compute the list *tcs(i)* of time moments at which vertex *i* can send messages (we also used this list in the dynamic programming algorithm), then we can improve the feasibility test. For each son *s(i,j)* of a vertex *i*, we can binary search the moment *tsend(i,j)* in the list *tcs(i)*. We will define a function *index(i,t)* which returns the index *k* of the largest time moment in the list *tcs(i)*, such that *tcs(i,k)≤t* (the function uses binary search). For *s(i,1)*, $tsend(i,1)=tcs(i, index(i, T_{latest}(s(i,1))-1))$. For *j>1*, $tsend(i,j)=tcs(i,index(i, min\{T_{latest}(s(i,j)), tsend(i,j-1)\}-1))$ (if some call of *index(*,*)* does not find any appropriate time moment, the feasibility test fails). The time complexity of the test is now *O(n·log(TBOUND))*. The algorithm is presented in Pseudocode 6-1.

A further improvement consists of computing a function *tprev(i,t)*=the largest time moment *t'* such that *t'≤t* and *sendb(i,t')=0* (for *t≥TM*, we have: *(1) tprev(i,t)=t*; *(2) tprev(i,t)=max{tprev(i,TM-1) + ((t div TM)-1)·TM, tprev(i,t mod TM)+(t div TM)·TM}*). We can tabulate *tprev(*,-1≤t<TM)* in *O(n·TM)* time: *tprev(i,-1)=-∞*; *tprev(i, 0≤t'<TM)=(if (sendb(i,t')=0)*

then t' else tprev(i,t'-1)). The complexity of the feasibility test becomes *O(n)*, because *tsend(i,j)=(if (j=1) then tprev(i, T_{latest}(s(i,1))-1) else tprev(i, min{T_{latest}(s(i,j)),tsend(i,j-1)}-1)*. All the algorithms require *O(n+n·TM)* preprocessing time or *O(n·TM)* storage.

```
GreedyFeasibilityTest(i, T_max):
if (ns(i)=0) then {
  T_latest(i)=T_max
  return "passed"
} else {
  for j=1 to ns(i) do {
    ret=GreedyFeasibilityTest(s(i,j), T_max)
    if (ret="failed") then return "failed"
  }
  sort the sons s.t. T_latest(s(i,1))≥...≥ T_latest(s(i,ns(i)))
  nextson=1
  for t=T_latest(s(i,1))-1 down to 0 do {
    if ((sendb(i,t)=0) and (t<T_latest(s(i,nextson)))) then {
      tsend(i,nextson)=t
      nextson=nextson+1
      if (nextson>ns(i)) then break
    }
  }
  if (nextson≤ns(i)) then return "failed"
  else {
    T_latest(i)=tsend(i,ns(i))
    return "passed"
  }
}
```

Pseudocode 6-1. The Greedy Feasibility Test for Minimum Time Broadcast in Trees with Sending Constraints.

6.3.2. Minimum Time Broadcast in Trees with Sending and Receiving Constraints

In this section we extend the problems discussed in the previous section, by adding receiving constraints, i.e. we have a function *recvb(i,t)*, which is *0* if vertex *i* can receive a message at time moment *t* and *1* if it cannot (i.e. the receiving interface is blocked). We consider the same two cases (*(1)* and *(2)*). We first present a dynamic programming algorithm similar to the one in the previous section.

We will compute the values $T_{min}(i,t)$=the minimum time moment at which all the vertices in vertex *i*'s subtree can receive the content, if vertex *i* receives the content at time *t*. We traverse the tree bottom-up and, if vertex *i* is a leaf, then we have $T_{min}(i,t)$=*(if (recvb(i,t)=1) then +∞ else t)*. For a non-leaf vertex *i*, we will determine the list of time moments *tcs(i,t)*, such that *t≤tcs(i,t,1)<tcs(i,t,2)<…<tcs(i,t,ntcs(i,t))* and *sendb(i, tcs(i,t,j))=0* (*1≤j≤ntcs(i,t)≤TBOUND*). As before, when moving from a time moment *t* to the next time moment *t+1*, we can update this list in *O(1)* time (if (*t<tcs(i,t,1)*) then *tcs(i,t+1)=tcs(i,t)*; otherwise, *tcs(i,t+1,j)=tcs(i,t,j+1)*, i.e. we remove the first time moment from *tcs(i,t)* and keep all the other time moments in *tcs(i,t+1)*). We construct a bipartite graph, containing the sons *s(i,1), …, s(i,ns(i))* of vertex *i* on one side and the time moments in *tcs(i,t)* on the other side. There exists an edge between a son *s(i,j)* and a time moment *tcs(i,t,k)* if *recvb(s(i,j), tcs(i,t,k)+1)=0*; the edge will have a weight equal to T_{min}*(s(i,j), tcs(i,t,k)+1)*. We need to find a maximum matching in which the maximum weight of an edge is minimum. This weight will be the value of T_{min}*(i,t)*.

We can find such a matching by binary searching the maximum weight *W* of an edge in the

matching (and performing a feasibility test for each candidate value). The feasibility test consists of removing all the edges with weights larger than W and computing a maximum matching in the bipartite graph using only the remaining edges. If the cardinality of this matching is $ns(i)$, then the feasibility test is passed and we can test a smaller value of W; otherwise, we need to test a larger value of W. The time complexity of the feasibility test is $O(log(TBOUND)\cdot ns(i)\cdot TBOUND\cdot sqrt(ns(i)+TBOUND))$ (if we use the $O(E\cdot sqrt(V))$ Hopcroft-Karp [Hopcroft and Karp, 1973] matching algorithm, where E is the number of edges and V is the number of vertices of the graph). The overall time complexity is obtained by multiplying the complexity of the feasibility test by $O(n\cdot TM)$.

We can use the binary search greedy approach here, too. We define

$$tprecv(i,t) = (\textbf{if } (t\geq TM) \textbf{ then } \{ (1)\ t\ ;\ (2)\ max\{tprecv(i,TM-1)+((t\ div\ TM)-1)\cdot TM,\ tprecv(i,t\ mod\ TM)+(t\ div\ TM)\cdot TM\}\ \}$$
$$\textbf{else if } (t<0) \textbf{ then } -\infty \qquad \textbf{(6-6)}$$
$$\textbf{else if } (recvb(i,t)=0) \textbf{ then } t$$
$$\textbf{else } tprecv(i,t-1))$$

(we tabulate the values $tprecv(i,t)$, $0\leq t<TM$).

For a leaf i, $T_{latest}(i)=tprecv(i,\ T_{max})$; for a non-leaf vertex i, we binary search the maximum possible value $T_{latest}(i)$ with a candidate value T_{cand}; we build the same bipartite graph as in the case of $T_{min}(i,T_{cand})$ (with i's sons and the subset of time moments of $\{T_{cand}, \ldots, T_{max}\}$ at which vertex i can send the message), from which we remove the edge weights and the edges $(s(i,j),t)$, with $t\geq T_{latest}(s(i,j))$. The feasibility test checks if the maximum matching in this bipartite graph has cardinality $ns(i)$ (i.e. T_{cand} is feasible if the matching's cardinality is $ns(i)$; if T_{cand} is feasible, then we can try larger values next, otherwise we will try smaller values). If we cannot find a maximum matching of cardinality $ns(i)$ for any value of T_{cand}, then T_{max} is not feasible (i.e. we will need to try larger values for T_{max} next).

We can also compute $T_{latest}(i)$ in a different way (without binary searching it). We will construct the bipartite graph BG mentioned before, for the case when $T_{cand}=T'=max\{0,\ min\{T_{latest}(s(i,j))|1\leq j\leq ns(i)\}\}$. We now compute a maximum matching in this graph. Let C be the cardinality of this matching. If $C<ns(i)$ then we will consider every time moment t from $T'-1$ down to 0 such that $sendb(i,t)=0$. We extend the graph BG by adding the time moment t to it, together with the corresponding edges connecting it to vertex i's sons (these edges $(s(i,j),t)$ obey the same constraints mentioned earlier, i.e. we must have $t<T_{latest}(s(i,j))$ and $recvb(s(i,j),t+1)=0$).

Then, we will try to extend the previous matching by considering the new vertex t which was added to BG. We will try to find an augmenting alternating path in BG starting from t (or a flow augmenting path in BG starting from t, if we use a maximum flow algorithm for computing the maximum matching). Let C be the cardinality of the newly obtained matching (C may remain unmodified if no augmenting path starting from t is found). If $C=ns(i)$ then we stop (we do not consider other values for t) and we set $T_{latest}(i)=t$. If C never becomes equal to $ns(i)$ then the current value T_{max} (which is binary searched) is not feasible (and we will need to consider a larger value). Overall, this approach performs at most as many computations as for the case $T_{cand}=0$ (from the previous approach); thus, it is more efficient by a $log(T_{max})$ factor asymptotically.

6.3.3. Maximum Weight Content Distribution Strategy in Trees subject to Time Limits

We consider here another variation of the restricted tree content distribution problem. Like before, a source vertex src needs to send a piece of content to all the other vertices of the tree. Every vertex i has a weight $w(i)\geq 0$. We are given a time limit T and we want to distribute the piece of content during the time interval $[0,T]$ to a subset S of vertices having a maximum total weight (a vertex i belongs to the subset S if it receives the content at a time $t\leq T$). The content is not sent further at time moments $t\geq T$ and the vertices which did not receive the content until time T will remain uninformed. This problem has applications to critical information dissemination, in which

there is very limited time for distributing very important information (regarding, for instance, a natural disaster or an enemy attack) and we want to maximize the weight (importance) of those who receive the information before a critical deadline.

We will compute the values $W_{max}(i,t)$=the maximum weight of the informed vertices in vertex i's subtree, if vertex i receives the content at time t ($0 \le t \le T$). The value $W_{max}(src, 0)$ will represent the solution to our problem. Using the $W_{max}(*,*)$ values, the optimal content distribution strategy can be easily obtained. We will compute these values bottom-up. If i is a leaf vertex, then $W_{max}(i,t)=w(i)$. For each pair (i,t) (with i being a non-leaf vertex), we will build a bipartite graph containing the sons $s(i,1), \ldots, s(i,ns(i))$ of vertex i on one side and the time moments $t, t+1, \ldots, T-1$ on the other side. We have an edge between every son $s(i,j)$ and every time moment t' such that $sendb(i,t')=0$ and $recvb(s(i,j),t'+1)=0$; the weight of this edge is $W_{max}(s(i,j),t'+1)$. We are interested in finding a maximum weight matching (where the weight of a matching is equal to the sum of the weights of the edges composing the matching). For a bipartite graph with V vertices and E edges, we can compute such a matching in $O(V^2 \cdot E)$ time. In our case, the time complexity will be $O((ns(i)+T-t)^2 \cdot ns(i) \cdot (T-t))$ for each pair (i,t). The overall time complexity is $O(n \cdot T^4 + n^2 \cdot T^3 + n^3 \cdot T^2)$.

6.4. Minimum Cost Spanning Tree with Special Offers

In this section we consider a minimum cost spanning tree problem, augmented with special offers. We have an undirected graph with n vertices (numbered from 1 to n) and m edges. Each edge e connects two different vertices $a(e)$ and $b(e)$, has an owner $o(e)$ and two prices: $np(e)$ and $sp(e)$. $np(e)$ is the normal price of the edge e and $sp(e)$ is the special price of the edge e; $sp(e) \le np(e)$.

There are q owners overall, numbered from 1 to q. Each owner has a special offer: it allows us to pay the special prices for any edges we want owned by that owner, with the condition that we do not take the special offer of any of the other owners. We want to establish a minimum cost spanning tree of the graph, by using at most one special offer of one of the edge owners.

At first, we will compute the minimum spanning tree of the graph considering normal prices for all the edges, in $O((m+n) \cdot log(n))$ or $O(m+n \cdot log(n))$ time if we use Prim's algorithm, or in $O(sort(m)+m \cdot \alpha(m,n))$, if we use Kruskal's algorithm. $sort(m)$ is the time complexity of sorting the m edges in increasing order of their normal cost. $sort(m)$ may be $O(m \cdot log(m))$, or $O(m+CMAX)$, if the costs are bounded by a small maximum value $CMAX$ (in this case we can use a sorting procedure similar to count-sort). $O(m \cdot \alpha(m,n))$ is the time complexity of using the disjoint sets data structure.

Let $MSTN$ be the set of $n-1$ edges composing the „normal" minimum spanning tree and let $CMSTN$ be the cost of the minimum spanning tree considering the normal prices. We will now consider every owner i (from 1 to q) and compute the minimum spanning tree in the case when we take advantage of owner i's special offer, i.e. when we consider the special prices for all the edges owned by i. We can recompute each such spanning tree in the same time complexity as when we computed the first minimum spanning tree, but the overall time complexity would be $O(q \cdot (m+n) \cdot log(n))$ or $O(q \cdot (m+n \cdot log(n)))$, if we use Prim's algorithm, or $O(q \cdot (sort(m)+m \cdot \alpha(m,n)))$, if we use Kruskal's algorithm.

If we initially sort all the edges once according to their normal costs and once according to their special costs in $O(sort(m))$ time, then for each owner i we can perform the edge sorting procedure (of the Kruskal's algorithm) as follows. Let $SE(i)$ be the subset of edges owned by i. We remove these edges from the ordering of the edges according to their normal prices, obtaining a sorted list L_1 of $m-|SE(i)|$ edges. Then, we remove from the ordering of the edges according to their special prices all the edges not belonging to $SE(i)$, obtaining a sorted list L_2 of all the edges from $SE(i)$. Then, by merging the lists L_1 and L_2 in $O(m)$ time, we can obtain the sorted list of all the edges, according to their corresponding price (normal or special). Thus, the time complexity in this case would be only $O(sort(m)+q \cdot (m+m \cdot \alpha(m,n)))$. Note than any removal of an edge from an edge ordering is cancelled when considering the next owner.

Nevertheless, the time complexity is too large if we use the approach presented in the previous paragraphs. Instead, we will proceed as follows. When computing the minimum spanning tree for the special offer of the owner i, we will consider only the subset of edges $ESE(i)$ composed

of all the edges owned by i (for which we consider their special prices) and all the edges from *MSTN* which are not owned by i, for which we consider their normal price. Then, we will compute the minimum spanning tree considering only $O(n+|SE(i)|)$ edges. Note that no other edge except for those we just mentioned can be part of this minimum spanning tree.

Thus, the time complexity for one minimum spanning tree computation is $O((|SE(i)|+n)\cdot log(n))$ or $O(|SE(i)|+n\cdot log(n))$, if we use Prim's algorithm, or $O(sort(|SE(i)|+n)+(|SE(i)|+n)\cdot\alpha(O(|SE(i)|+n),n))$, if we use Kruskal's algorithm. The sum of the values $(|SE(i)|+n)$ $(1\le i\le q)$ is at most $(m+(n-1)\cdot q)$. Thus, the overall time complexity will be $O((m+n\cdot q)\cdot log(n))$ or $O(m+n\cdot q+n\cdot q\cdot log(n))$, if we use Prim's algorithm, or $O(min\{sort(m+n\cdot q), sort(m)+n\cdot q\cdot log(m)\}+(m+n\cdot q)\cdot\alpha(O(m),n))$, if we use Kruskal's algorithm.

Note that when using Kruskal's algorithm, we can sort all the $O(|SE(i)|+n)$ edges in $O(|SE(i)|+n)$, if we initially sort all the edges once according to their special price, and all the edges in *MSTN* according to their normal price. After performing the initial ordering according to the special prices, we construct a list $LE(o)$ of edges for each owner o, as follows. We traverse the ordering of the edges and we add each edge at the end of the list of its owner. Thus, in $O(m)$ time we obtain all the n sorted lists (after performing a sorting procedure which takes $O(sort(m))$ time). Then, when we consider an owner i, we obtain a list L by removing from the ordering of the edges in *MSTN* those edges which are owned by i. By merging the sorted lists L and $LE(i)$, we obtain the sorted list of all the considered edges. With this method, the time complexity when using Kruskal's algorithm becomes $O(sort(m)+m+n\cdot q+(m+n\cdot q)\cdot\alpha(O(m),n))$.

The time complexities of using both Prim's and Kruskal's algorithm with the second approach are much better than that of the trivial algorithm.

6.5. Minimum Cost Steiner Tree

The minimum Steiner tree problem is well-known and we will discuss in this section only some optimizations for several special cases. We have an undirected, connected graph with n vertices and m edges. Every edge (u,v) has a cost $c(u,v)\ge 0$. d of the n vertices of the graph are special: $x(1), ..., x(d)$. We want to interconnect these nodes (in a tree-like manner) in such a way that the total cost of the used edges is minimum.

It is easy to notice that a minimum cost Steiner tree contains at most $d-2$ nodes different from the special nodes and whose degrees in the tree are larger than 2. We will start by handling some particular cases. If $d=1$ the tree consists of just one node: $x(1)$. If $d=2$ then the tree consists of the shortest path between $x(1)$ and $x(2)$. If $d=3$ then we will compute the length of the shortest path from every node $x(i)$ $(1\le i\le 3)$ to every node in the graph. Let $dmin(i,j)$ be the length of the shortest path between the nodes i and j. We will consider every node q of the graph and compute $Cost(q)=dmin(x(1),q)+dmin(x(2),q)+dmin(x(3),q)$. We will choose that node q for which $Cost(q)$ is minimum and we will connect every node $x(i)$ to q through the shortest path between $x(i)$ and q $(1\le i\le 3)$. Note that q may also be one of the 3 special nodes.

For $d\ge 4$ we can compute in $O(n^3)$ time the lengths of the shortest paths $dmin(i,j)$ between every pair of vertices (i,j) (e.g. by using the Floyd-Warshall algorithm). We will then consider every subset of nodes containing exactly S nodes $(0\le S\le min\{d-2, n-d\})$ among the $n-d$ nodes which are not special. With the nodes in the subset and the special nodes $x(i)$ $(1\le i\le d)$ we will consider the complete graph $G(S)$ which contains only these $d+S$ nodes, and the cost of an edge between two vertices a and b in $G(S)$ will be $dmin(a,b)$. We will compute $Cost(S)$=the cost of a minimum spanning tree in $G(S)$. The minimum Steiner tree is the spanning tree with minimum cost obtained this way.

In the particular case $d=4$ and $n\ge 6$ we can simplify things a bit. We can consider every pair of nodes (a,b) (nodes a and b may be identical, and any of them may also be a special node). The Steiner tree will consist of the shortest path between a and b and will then connect every node $x(i)$ to the closest node from the set $\{a,b\}$. Thus, the cost of such a tree will be $Cost(a,b)=dmin(a,b)$ plus the sum of the values $min\{dmin(x(i),a), dmin(x(i),b)\}$ $(1\le i\le 4)$. Of course, the minimum Steiner tree is obtained for the pair (a,b) with the minimum value of $Cost(a,b)$. As we can see, for $d=2,3$, the

time complexity of the solution is $O((n+m)\cdot log(n))$ (or $O(m+n\cdot log(n))$, depending on the implementation chosen for the shortest path algorithm), and for $d\geq 4$, the time complexity is $O(n^3+d^2\cdot n^{min\{d-2,n-d\}})$.

6.6. Minimum Time Broadcast Strategy in a Generalized Version of the LogP Model

In this section we want to provide an explicit algorithm for computing a minimum time broadcast strategy in a slightly generalized version of the LogP model [Karp, Sahay, Santos and Schauser, 2003]. Guidelines for developing such an algorithm have been provided in [Karp, Sahay, Santos and Schauser, 2003].

We have a network composed of P identical processors (numbered from 1 to P). The processor 1 has a message which needs to be sent to all the other processors. The duration of transmitting a message between any two processors is L. Before sending a message, a processor (which previously received the message) consumes ts time units with the preparation of the sending of the message. After the message reaches a processor, tr time units are consumed at the receiver before the message is decoded and properly received. Thus, the total duration from the moment when a proceesor A decides to send the message to a processor B and until the moment when B effectively receives the messageis equal to $ts+tr+L$. After sending the message, a processor must wait at least g time units before starting sending the next message. We want to compute a minimum time broadcast strategy, i.e. a strategy in which the maximum time moment at which a processor effectively receives the message is minimum. Note how the parameters ts and tr generalize the parameter o of the LogP model.

We will maintain a heap H with time moments at which the processors which have already received the message can start sending it further. Each value from H will also have the processor index associated to it. We will also maintain a counter np, storing the number of processors which have received the message, and a value $Tmax$.

Initially, we have $np=1$, $Tmax=0$ and we insert the value 0 (with the associated processor index 1) in H. While ($np<P$) we will extract from H the minimum value $Tmin$ (and let its associated processor index be q). The processor q will send a message to the processor $np+1$, which will be effectively received at the time moment $Tmin+ts+tr+L$. We will set $Tmax=Tmin+ts+tr+L$. Then, we will insert into H the values: $Tmax$ (with its associated processor index $np+1$) and $Tmin+g$ (with its associated processor index q). Then, we will increment np by 1.

The last value of Tmax is the minimum duration of the optimal broadcast strategy. The time complexity of the presented algorithm is $O(P\cdot log(P))$.

If the time moments are integer and we know an upper bround TM for the broadcast duration, we can maintain a list $L(t)$ with processor indices for each time moment t ($0\leq t\leq TM$). Initially, $L(0)$ will contain the processor index 1, while the other lists will be empty. Then, while ($np<P$), we will traverse these lists, in increasing order of the time moment $Tmin$ (starting with $Tmin=0$). We consider every processor q from $L(Tmin)$ and we perform the already described actions: we remove q from $L(Tmin)$, we insert q into $L(Tmin+g)$ and we insert $np+1$ into $L(Tmin+ts+tr+L)$; after this, we increment np by 1 (and we stop the algorithm if np becomes equal to P). The time complexity in this case is $O(P+TM)$. Note also that, when we reached the time moment $Tmin$, only the lists $L(t)$ with $Tmin\leq t\leq Tmin+ts+tr+L$ may be non-empty. This suggests that we may not store $O(TM)$ lists, but rather only $M=O(ts+tr+L)$ lists (e.g. $M=ts+tr+L+1$). Then, the list $L(t)$ may in fact be stored at the memory position of the list $L(t \bmod M)$ (if we use, for instance, an array of lists).

Chapter 7 – Optimal Replica Placement in Tree-Like Content Delivery Networks

In this chapter we will consider distributed systems whose application-level functionality is that of storing and facilitating the retrieval of data items. We will introduce several techniques for designing and optimizing such systems, always maintaining our focus on the communication parameters (e.g. minimizing the communication latency).

We will start by presenting novel algorithms for optimally placing replicas of highly popular data items in tree-like content delivery networks (cacti, trees and paths), where the optimization metrics are minimizing the sum of (weighted) latencies and minimizing the maximum (weighted) latency. For tree graphs we developed a new algorithmic framework which is used for solving some restricted versions of the connected k-center and k-median problems.

Then, we will discuss the balanced content replication problem in trees, which is equivalent to the k-equitable coloring problems in trees. For this problem, we present a novel efficient algorithmic solution. Then, we define a tree reliability metric based on the unrestricted vertex multicut problem in trees (for which we present the first linear time solution). The reliability metric is evaluated on the output of the k-equitable tree coloring algorithm. In the last part of this chapter we introduce a new dynamic programming framework for several optimization problems in graphs with bounded pathwidth. This framework is used for solving replica placement problems in such graphs.

The results presented in this chapter were published in [Andreica and Ţăpuş, 2009g], [Tîrşa, Andreica and Costan, 2009], [Andreica, Andreica and Vişan, 2009], [Andreica, 2008a], [Andreica and Ţăpuş, 2008a], [Andreica and Ţăpuş, 2008h], [Andreica, 2008d], [Andreica et al., 2008] and [Andreica, Tîrşa, Costan and Ţăpuş, 2009].

7.1. Replica Placement in Tree-Like Content Delivery Networks

The problem of efficiently placing data replicas in content delivery networks is very important due to the wide spread usage of such networks. The replicas should be placed such that they minimize the network traffic generated in order to reach them and acquire the stored content, according to an objective metric. Weighted min-max and min-sum metrics are most widely used and, in computer science, the corresponding optimization problems consist of locating the k-center and the k-median of the graph which represents the topology of the network.

In this section we will consider only networks with a tree-like topology: cacti, trees and paths. Although the tree-like structure may seem to be only a particular case, we argue that many existing networks have a hierarchical structure, e.g. with users devices at the edge and router backbones at its core. Moreover, many graph topologies can be reduced to tree topologies, by choosing a spanning tree or by decomposing the set of edges into edge disjoint spanning trees.

The *(k+p)*-center problem is the following: Given a graph with n vertices where each vertex has a weight and each edge has a length, we want to place at most $k \geq 0$ ($k+p \geq 1$) servers in the graph, such that the maximum weighted distance from a vertex to its closest server is minimized. If the servers may be placed only at the graph vertices, we are considering the *discrete* case. If they can be placed anywhere along the edges, it is the *continuous* version. Furthermore, $p \geq 0$ *fixed* servers are already placed in the network (at the graph vertices or on its edges) and they can be used by the vertices. Thus, we need to place at most k *extra* servers.

Note that although the fixed servers may be located along the graph's edges, we can insert new vertices on the edges at the locations of the fixed servers, obtaining a graph with $n+p$ vertices where the fixed servers are located only at its vertices (and, thus, we will only consider this case). The *(k+p)*-median problem is defined similarly, except that we want to minimize the sum of

weighted distances from all the vertices to their closest server. We also consider an extra requirement for the discrete case (of both the center and median problems): the set of (at most) k locations chosen for the servers must form a connected subgraph of the given graph. This requirement makes sense when the data stored on the servers needs to be (periodically) synchronized or when a server may redirect requests to other servers.

A *path network* with n nodes has the property that its vertices can be numbered in the order $v(1), \ldots, v(n)$, such that the only existing edges are between $v(i)$ and $v(i+1)$ ($1 \le i \le n-1$). A *tree network* is an undirected, connected, acyclic graph. A tree may be rooted, in which case a special vertex r will be called its root. Even if the tree is unrooted, we may choose to root it at some vertex.

In a rooted tree, we define $parent(r,i)$ as the parent of vertex i (when the tree's root is r) and $ns(r,i)$ as the number of sons of vertex i (when the tree root is r). For a leaf i, $ns(r,i)=0$ and for the root r, $parent(r,r)$ is undefined. The sons of a vertex i are denoted by $s(r,i,j)$ ($1 \le j \le ns(r,i)$). A vertex j is a *descendant* of vertex i if $(parent(r,j)=i)$ or $parent(r,j)$ is also a descendant of vertex i. We denote by $T(r,i)$ the subtree rooted at vertex i, i.e. the part of the tree composed of vertex i and all of its descendants (and the edges connecting them), when r is the tree root. When the root r is always the same, we may drop the index r from these notations (e.g. we may use $parent(i)$, $s(i,j)$ and $T(i)$, instead of $parent(r,i)$, $s(r,i,j)$ and $T(r,i)$).

A *cactus* is a graph in which any two cycles are edge-disjoint. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices.

7.2. Replica Placement in Cactus Networks

Every edge (u,v) has a weight $we(u,v)$ and every vertex u has a weight $wv(u)$. The *length* of a path $v(1), \ldots, v(k)$ is the sum of the weights of the vertices $v(i)$ ($1 \le i \le k$) and of the edges $(v(j),v(j+1))$ ($1 \le j \le k-1$). In this section we will consider the problems of computing the longest path, diameter, and 1-center of a cactus graph. The 1-center problem is equivalent to the replica placement problem, as mentioned in the previous section.

7.2.1. Longest Path

We will present here a linear time algorithm for the problem of computing the longest path in a cactus, i.e. the path having the largest *length*. First, we will compute the block-cut vertex tree T_{BC} [Das and Pas, 2008] of the cactus graph. Such a tree can be computed in $O(n+m)$ time for a graph with n vertices and m edges. Since $m=O(n)$ in a cactus, this takes linear time. Every vertex of T_{BC} is either a biconnected component (type BC) or a cut vertex (type C) of the original graph. We choose the root r such that it is a biconnected component. The neighbors of each vertex corresponding to a biconnected component B are the cut vertices which belong to B. The sons of each vertex corresponding to a cut vertex CV are the biconnected components which contain CV (except for the component B which is CV's parent).

For every node x of T_{BC}, we define $C(x)$ as follows: if x corresponds to a cut vertex cv, then $C(x)=cv$; otherwise, if x is not the root of the tree, then $C(x)=C(parent(x))$ (i.e. the cut vertex belonging to the biconnected component corresponding to x which is its parent in T_{BC}). If x is the root of the tree, then $C(x)$ can be set to any vertex in the biconnected component corresponding to the tree root.

For every node x of T_{BC} we will compute two values: $A(x)$=the length of the longest path starting at $C(x)$, and $B(x)$=the length of the longest path passing through $C(x)$ if x is of type C, or through any vertex of the biconnected component corresponding to x, if x is of type BC. These paths may contain only vertices contained in node x's subtree of T_{BC}. The length of the longest path is $B(r)$.

It is easy to compute these values for a node x of type C. We compute

$$A_1(x)=max\{ wv(C(x)), max\{A(y) | y \text{ is a son of } x\}\} \quad \textbf{(7-1)}$$

and set y_1 to the son y of x with the largest value $A(y)$. We also compute

$$A_2(x)=max\{A(y) | y \text{ is a son of } x,\ y \ne y_1\} \quad \textbf{(7-2)}$$

(if x has at most one son, then $A_2(x)=wv(C(x))$). We have $A(x)=A_1(x)$ and $B(x)=A_1(x)+A_2(x)$-

wv(C(x)). For a node *x* of type *BC*, we will denote by *v(x,1), …, v(x,nv(x))* the vertices of the biconnected component corresponding to *x*. In a cactus, every biconnected component is a cycle. If the component consists of only one edge *(v(x,1),v(x,2))*, we will double this edge, in order to obtain a cycle composed of two vertices and two edges (of equal weights).

We will order the vertices in the order in which they appear on the cycle, starting from *v(x,1)=C(x)* and continuing in one of the two possible directions. Thus, we have *v(x,1)=C(x)*, *v(x,i)* and *v(x,i+1)* are adjacent (*1≤i≤nv(x)-1*), and *v(x,nv(x))* and *v(x,1)* are adjacent. We will assign a value *l(x,j)* to each vertex *v(x,j)*. If *v(x,j)* (*1≤j≤nv(x)*) is a cut vertex corresponding to a node *y* which is a son of *x* in T_{BC}, we set *l(x,j)=A(y)-wv(v(x,j))*; otherwise, *l(x,j)=0*. We will first compute $A_1(x)$, the length of the longest path starting at *v(x,1)* (*=C(x)*), not containing the edge *(v(x,nv(x)),v(x,1))*.

We traverse the vertices in increasing order of their index *j* and compute: $lsum_1(x,0)=0$, $lsum_1(x,1)=wv(v(x,1))$ and for *j>1*, $lsum_1(x,j)=lsum_1(x,j-1)+we(v(x,j-1),v(x,j))+wv(v(x,j))$. $A_1(x)=max\{lsum_1(x,j)+l(x,j) \mid 1\le j\le nv(x)\}$.

$$lsum_1(x,j) = wv(v(x,j)) + \sum_{i=1}^{j-1} (wv(v(x,i)) + we(v(x,i), v(x,i+1))) \quad \textbf{(7-3)}$$

We will now compute $A_2(x)$, the length of the longest path starting at *v(x,1)*, containing the edge *(v(x,nv(x)), v(x,1))*. We compute the values $lsum_2(x,j)$:

$$lsum_2(x,j) = wv(v(x,j)) + \sum_{i=j}^{nv(x)-1} (wv(v(x,i+1)) + we(v(x,i+1), v(x,i))) \quad \textbf{(7-4)}$$

We have $lsum_2(x,nv(x))=wv(v(x,nv(x)))$ and for *2≤j≤nv(x)-1*, $lsum_2(x,j)=lsum_2(x,j+1)+we(v(x,j+1), v(x,j))+wv(v(x,j))$.

We set $A_2(x)$ to $max\{lsum_2(x,j)+ l(x,j)+wv(v(x,1))+we(v(x,1),v(x,nv(x))) \mid 2\le j\le nv(x)\}$. *A(x)* will be equal to $max\{A_1(x), A_2(x)\}$.

In order to compute *B(x)*, we consider the same two cases as before, and compute two values, $B_1(x)$=the length of the longest path passing through some vertex *v(x,j)* (*1≤j≤nv(x)*), not containing the edge *(v(x,1),v(x,nv(x)))*, and $B_2(x)$=the length of the longest path passing through some vertex *v(x,j)*, containing the edge *(v(x,1),v(x,nv(x)))*. $B(x)= max\{B_1(x), B_2(x)\}$. In order to compute $B_1(x)$, we will assign to each vertex *v(x,j)* the same value *l(x,j)* as before and then compute the same values $lsum_1(x,j)$.

Afterwards, we will also compute the values $lmax_1(x,j)$. We have $lmax_1(x,0)=0$ and $lmax_1(x,j)=max\{lmax_1(x,j-1), l(x,j)+wv(v(x,j))-lsum_1(x,j)\}$ (*1≤j≤ nv(x)*).

Then, we compute $lmax_2(x,j)=max\{wv(v(x,j))+l(x,j), l(x,j) + lsum_1(x,j) + lmax_1(x,j-1)\}$ (*1≤j≤ nv(x)*). $lmax_2(x,j)$ represents the length of the longest path containing a segment *v(x,i), …, v(x,j)* (*1≤i≤j*) of the cycle, and not containing any other part of the cycle. Thus, the path starts at some vertex *v(x,i)* (*i≤j*) or somewhere in its subtree (if *v(x,i)* is a cut vertex), walks along the cycle from *v(x,i)* to *v(x,j)* (in increasing order of the vertex indices) and either ends at *v(x,j)* or at a vertex in *v(x,j)*'s subtree (if *v(x,j)* is a cut vertex). $B_1(x)=max\{lmax_2(x,j) \mid 1\le j\le nv(x)\}$.

In order to compute $B_2(x)$, we need to compute the values $lsum_1(x,j)$ and $lsum_2(x,j)$, defined previously. Afterwards, we compute $lmax_3(x,j)$ ($lmax_3(x,1)=lsum_1(x,1)+l(x,1)$ and $lmax_3(x,j>1)=max\{lmax_3(x,j-1), lsum_1(x,j)+l(x,j)\}$) and $lmax_4(x,j)$ ($lmax_4(x,nv(x))= lsum_2(x,nv(x))+l(x,nv(x))$ and $lmax_4(x,j<nv(x)) = max\{lmax_4(x,j+1), lsum_2(x,j)+l(x,j)\}$).

$B_2(x)$ is equal to $we(v(x,1),v(x,nv(x))) + max\{lmax_3(x,j)+lmax_4(x,j+1) \mid 1\le j\le nv(x)-1\}$.

$lmax_3(x,j)$ is the length of the longest path starting at *v(x,1)* and ending at a vertex *v(x,i)*, with *i≤j* (or in *v(x,i)*'s subtree). $lmax_4(x,j)$ is the length of the longest path starting at *v(x,nv(x))* and ending at a vertex *v(x,i)*, with *i≥j* (or in this vertex's subtree). Thus, the path corresponding to $B_2(x)$ is composed of the edge *(v(x,1), v(x,nv(x)))*, a segment of the cycle starting at *v(x,1)* and ending at some vertex $v(x,i_1)$ ($1\le i_1$), a segment of the cycle starting at *v(x,nv(x))* and ending at some vertex $v(x,i_2)$ ($i_1<i_2\le nv(x)$), plus the longest paths starting at $v(x,i_1)$ and $v(x,i_2)$ and ending in their subtrees (these paths may be void). All the values can be computed in *O(n+q)* time, where *n* is the number of vertices of the graph and *q=O(n)* is the number of vertices of T_{BC}.

Note that unlike the longest path problem, finding the longest (maximum weight) cycle in a cactus graph (in linear time) is quite easy. We first perform a DFS traversal of the cactus (starting

from an arbitrary vertex r) and, thus, we obtain a DFS tree rooted at r (let $parent(u)$ be the parent of the vertex u in this DFS tree). Then, for each vertex u, we compute $plen(u)$: if $u=r$ then $plen(r)=wv(r)$; otherwise, $plen(u)=plen(parent(u))+we(parent(u), u)+wv(u)$.

We also compute, for each vertex u, its level in the DFS tree: $level(r)=1$ and $level(u\neq r)=level(parent(u))+1$. Afterwards, we consider all the edges (u,v) which are not part of the DFS tree. Each such edge closes a cycle in the DFS tree and all these cycles are disjoint. The weight of this cycle is $wlen(u,v)=we(u,v)+(if\ (level(u)<level(v))\ then\ (plen(v)-plen(u)+wv(u))\ else\ (plen(u)-plen(v)+wv(v)))$.

The maximum weight of a cycle is $max\{wlen(u,v) \mid (u,v)$ is an edge which does not belong to the DFS tree$\}$.

7.2.2. Discrete 1-Center and Diameter

In order to compute the center vertices and the diameter of a cactus, we will use the same block-cut vertex tree T_{BC}. For each node x of T_{BC}, we will compute $A(x)$=the longest shortest path starting at $C(x)$, which may contain only vertices located in node x's subtree. For each vertex i of the cactus, we will compute $l_1(i)$, $l_2(i)$ and $l_3(i)$, the lengths of the longest, 2^{nd} longest and 3^{rd} longest shortest paths in the graph which start at the vertex i, with the condition that these 3 paths *are computed* at different nodes of T_{BC}.

Initially, the values $l_1(i)$, $l_2(i)$ and $l_3(i)$ are set to 0. We also maintain the nodes $x_p(i)$ of T_{BC} where the corresponding $l_p(i)$ value was computed (initially, $x_p(i)=0$, $p=1,\ldots,3$). During the algorithm described below, we will frequently identify a candidate value val (computed at a node x of T_{BC}) for $l_1(i)$, $l_2(i)$ and $l_3(i)$. Every time we do this, if we have $x_p(i)=x$ (for some $p=1,\ldots,3$), we replace $l_p(i)$ by $max\{l_p(i), val\}$ and then re-sort the values $l_1(i), \ldots, l_3(i)$. If $x\neq x_p(i)$ (for all $p=1,\ldots,3$), we will compute val_1, val_2 and val_3, the 1^{st}, 2^{nd} and 3^{rd} maximum values in the (multi)set $\{val, l_1(i), l_2(i), l_3(i)\}$. The new values of $l_1(i)$, $l_2(i)$ and $l_3(i)$ will be val_1, val_2 and val_3 (in this order); we then also set the values $x_p(i)$ accordingly ($p=1,\ldots,3$).

For a type C node x of T_{BC}, we have $A(x)=max\{wv(C(x)), max\{A(y) \mid y \text{ is a son of } x\}\}$. For a type BC node x of T_{BC}, we consider an ordering of the vertices in the corresponding cycle (biconnected component): $v(x,1)=C(x)$, $v(x,2), \ldots, v(x,nv(x))$. We will compute $WC(x)$, the sum of all the edge and vertex weights in the cycle. Afterwards, we will double this list of vertices, by attaching a copy of the list at the end of the list. Thus, we will have $nv'(x)=2\cdot nv(x)$ elements in the list, with $v(x,j)=v(x,j-nv(x))$ ($nv(x)<j\leq nv'(x)$).

We will compute the same $lsum_1(x,j)$ and $l(x,j)$ values as in the longest path case (for all the $nv'(x)$ vertices). We will traverse all the vertices in order, from 1 to $nv'(x)$, and maintain a sorted double-ended queue (deque) DQ. All the pairs (p,val) in DQ will be maintained in increasing order. Whenever we want to add a value (q,val) at the end of DQ, we will repeatedly remove the element (p,val') at the end of DQ, as long as $val'\geq val$; only after this will we insert (q,val) at the end of DQ.

For each vertex $v(x,1\leq j\leq nv'(x))$, we will first add $(j,l(x,j)+wv(v(x,j))-lsum_1(x,j))$ at the end of DQ. Then, we will repeatedly remove the element (p,val) at the front of DQ, as long as $(p<j)$ and $(lsum_1(x,j)-lsum_1(x,p)-wv(v(x,j))>(WC(x)-wv(v(x,j))-wv(v(x,p)))/2)$.

After this, a candidate value for $l_1(v(x,j))$, $l_2(v(x,j))$ and $l_3(v(x,j))$ will be computed (at node x) as follows: (1) let (p,val) be the first element in DQ; (2) the candidate value will be $val+lsum_1(x,j)$. At the end of this stage, we will consider a different ordering v': $v'(x,1)=v(x,1)$ and $v'(x,2\leq j\leq nv'(x))=v(x,nv'(x)-j+2)$ (this is the opposite ordering on the cycle, starting from $v(x,1)$). We will run the algorithm described in this paragraph again, considering the new ordering (we start from (re)computing the $lsum_1(x,j)$ and $l(x,j)$ values, and then we traverse the vertices in the new order). After processing the node x (of type BC), we set $A(x)=max\{l_p(C(x)) \mid 1\leq p\leq 3\}$.

After running the algorithm for the entire tree T_{BC}, we need to traverse the tree again, top-down this time (i.e. we consider a node x before all of its sons). If x is a non-root node of type BC, we need to consider the following candidate values for $l_1(v(x,i))$, $l_2(v(x,i))$ and $l_3(v(x,i))$ (considering the same ordering of the vertices, starting from $v(x,1)=C(x)$): for a vertex $v(x,j>1)$, a candidate value (computed at node x) would be $dist(v(x,1), v(x,j))+max\{l_p(v(x,1))) \mid 1\leq p\leq 3,\ x_p(v(x,1))\neq x\}$ (the

last term is the length of the longest shortest path that starts at $C(x)$ and does not pass through another vertex of x).

We have $dist(v(x,1), v(x,j))=min\{lsum_1(x,j), WC(x)-lsum_1(x,j)+wv(v(x,1))+ wv(v(x,j))\}$ (i.e. the length of the shortest path between $v(x,1)$ and $v(x,j)$ on the cycle).

The radius of the cactus graph is $r=min\{l_1(i)|1\leq i\leq n\}$ and the diameter is $max\{l_1(i)+l_2(i)|1\leq i\leq n\}$. The center vertices of the graph are those vertices i with $l_1(i)=r$. The complexity is $O(n)$.

7.3. Tree (K+P)-Centers

We consider a tree with n vertices, in which every vertex u has a weight $wv(u)$ and every edge (u,v) has a length $l(u,v)$. The p fixed servers are placed at some of the tree vertices. The weighted distance from a vertex u to a server v in the tree is $wd(u,v)=wv(u)\cdot dist(u,v)$, where $dist(u,v)$ is the sum of the lengths of the edges on the unique path between u and v. We will binary search the minimum weighted distance WD, such that the weighted distance from every vertex u to the closest server is at most WD. For a candidate weighted distance WD_{cand}, we will perform the following feasibility test.

For each vertex i, we compute $dmax(i)=WD_{cand}/wv(i)$. We root the tree at an arbitrary vertex r and traverse the tree bottom-up (from the leaves towards the root). We will maintain a counter nf, initialized to 0. For each vertex i we will compute the values: $dmin(i)$=the (non-weighted) distance to the closest server in $T(r,i)$ and $smin(i)$ the maximum distance away from i at which a new server needs to be placed. We initialize $dmin(i)=+\infty$ and $smin(i)=dmax(i)$.

If we have a fixed server placed at vertex i, we set $dmin(i)=0$ and $smin(i)=+\infty$; if not, we traverse all the sons $s(r,i,j)$ of vertex i and, for each son, we perform the following updates: $dmin(i)=min\{dmin(i), l(i,s(r,i,j))+dmin(s(r,i,j))\}$ and $smin(i)=min\{smin(i), smin(s(r,i,j))-l(i,s(r,i,j))\}$. If $(dmin(i)\leq smin(i))$ then we set $smin(i)=+\infty$. If $(i\neq r)$ and $(smin(i)<l(parent(r,i),i))$ then: (1) in the discrete case, we place a server at vertex i and set $dmin(i)=0$; (2) in the continuous case, we place a server on the edge $(i, parent(r,i))$, at distance $smin(i)$ from i and, after this, we set $dmin(i)=-smin(i)$.

In both situations ((1) and (2)), we increment nf by 1 afterwards, and then set $smin(i)=+\infty$. If $(i=r)$ and $(smin(i)<dmin(i))$ then we place a new server at vertex r and increase nf by 1. If, at the end, we have $nf\leq k$, then WD_{cand} is feasible; otherwise, it is not. The time complexity of the algorithm is $O(n\cdot log(WMAX))$ ($WMAX$ is $max\{wv(i)\}$ multiplied by the length of the longest path).

For the connected k-center problem in trees, we could only find a solution for the unweighted case (i.e. all the vertex weights are equal to 1). We will start with the case where all the edges have unit length. In this case, we can repeatedly remove the layers of leaves from the tree, until we remain with only one or two vertices (the tree center(s)).

We consider the layers as $l(1), \ldots, l(nl)$, sorted from the center(s) towards the leaves of the tree ($l(1)$ contains the tree center(s)); each layer $l(i)$ has $nlv(i)$ vertices. We will find the largest index j, such that $nlv(1)+\ldots+nlv(j)\leq k$ and ($j=nl$ or $nlv(1)+\ldots+nlv(j+1)>k$) by linear or binary search. We will place the k centers at all the vertices in $l(1), \ldots, l(j)$.

When the edges have different lengths, we compute for each vertex i the value $dmax(i)$=the largest distance from vertex i to one of the tree leaves. Then, we sort the vertices such that $dmax(v(1))\geq\ldots\geq dmax(v(i))$. For both the discrete and continuous cases, we will place the k servers at $v(1), \ldots, v(k)$. It is easy to prove that they form a connected subset. The algorithm has linear time complexity in the unit edge length case and $O(n\cdot log(n))$ for the case with different edge lengths.

7.4. Centers on Wireless Path Networks

In this section we consider a wireless path network modeled as a set of n points located on the real line. Each point i is a node of the network, is located at coordinate x_i and has a weight w_i (e.g. its expected service demand or its expected amount of requested data). We are interested in placing (at most) k identical servers, each of them easily covering all the nodes at a distance equal to $L/2$ around them. In geometric terms, each server is the midpoint of an interval of fixed length $L\geq 0$, which covers all the points contained in the interval.

The weighted distance from a point x_j to an interval $[a,b]$ is: 0, if $a \le x_j \le b$; $w_j \cdot (a-x_j)$, if $x_j < a$; $w_j \cdot (x_j-b)$, if $x_j > b$. If the weighted distance is non-zero, the corresponding vertex needs to increase its power consumption in order to reach the closest server. We want to place the k servers (intervals) in such a way that the maximum weighted distance from a point to the closest interval is minimized. Furthermore, p fixed servers are already placed.

We will binary search the weighted distance *WD* with a specified accuracy. The feasibility test consists of computing for each point i an interval $[lx_i, rx_i]$ [Andreica et al., 2008], where the left endpoint of one of the k intervals can be placed. We will first verify if any of these intervals contains one of the left endpoints of the p fixed servers. We will ignore all such intervals in the decision algorithm, because the points associated with these intervals are "satisfied" by the p fixed servers (i.e. are already within weighted distance *WD* from one of these servers).

In order to exclude the points which are satisfied by the p fixed servers, we sort the intervals assigned to each point and the left endpoints of the intervals of the fixed servers and maintain a set S of open point intervals (an interval is opened when its left endpoint is encountered and closed when we encounter its right endpoint). When we reach the left endpoint of a fixed server, we mark all the open point intervals at that moment (and remove them from S).

In the end, we sort the unmarked intervals and we compute the minimum number of points np such that every interval is pierced by at least one point. If $np > k$, then a larger candidate weighted distance is tested; otherwise, we test a smaller one.

We will now present some improvements for $k=1$ and $p=0$. In *Solution 1*, the feasibility test becomes linear (no sorting is required) and consists of comparing the smallest right endpoint *sre* of an interval $[lx_i, rx_i]$ against the largest left endpoint *lle* of an interval $[lx_j, rx_j]$. If $sre \ge lle$, then the candidate weighted distance is feasible; otherwise, it is not.

In *Solution 2* we define a function $dmax(q)$=the maximum weighted distance from one of the n points to the interval $[q,q+L]$. This function is *unimodal*, i.e. it descends up to $q=q_0$, where $dmax(q_0)$ is the minimum value of $dmax(q)$ and then it ascends again. As it is well-known, we can find the minimum value (and x-coordinate) of a unimodal function by using binary search on its "derivative".

To be more precise, we binary search the optimal value q_0 in the interval $[a,b]$, where a is the x-coordinate of the leftmost point and b is $max\{a,c-L\}$, with c being the x-coordinate of the rightmost point. The feasibility test for a value q consists of computing $dif(q)=dmax(q+\varepsilon)-dmax(q)$, where $\varepsilon > 0$ is a very small constant. If $dif(q) \ge 0$, then $q \ge q_0$; otherwise, $q < q_0$. The feasibility test runs in linear time (it computes in $O(1)$ time the weighted distance from each point to the interval and selects the maximum distance).

Although the solutions presented above are more efficient than the standard algorithm for interval k-centers [Andreica et al., 2008] (for $k=1$ and $p=0$), we can improve the complexity to $O(n)$, if the points are sorted. We will first introduce a linear algorithm for computing the upper-envelope of right-oriented half-lines.

7.4.1. Upper Envelope of Right-Oriented Half-Lines

Let's consider a set of n half lines with their origins at $(x_{i,0}, y_{i,0})$ and with equations of the form $y_i(x)=y_{i,0}+w_i \cdot (x-x_{i,0})$ $(1 \le i \le n)$, defined only for x in $[x_{i,0}, \infty]$ (w_i is the slope of the half line). The upper envelope of these half-lines consists of a set of line segments (except for the last part, which is a half line). Each line segment (x_a,y_a)-(x_b,y_b) is part of a half-line i such that the values $y_i(x)$ $(x_a \le x \le x_b)$ are larger than the values of all the other half lines.

We will present a linear time algorithm for computing the upper envelope of these half-lines, if they are given in sorted order of their origins (ascending after $x_{i,0}$ and descending after $y_{i,0}$, for equal values of $x_{i,0}$), i.e. for $i<j$ we have $(x_{i,0} < x_{j,0})$ or $((x_{i,0}=x_{j,0})$ and $(y_{i,0} > y_{j,0}))$. We will assume that no two half-lines i and j have the same origin; if they do, then we will keep the one with the larger slope. Moreover, we need to have the following condition: the origins of every half line i $(x_{i,0}, y_{i,0})$ must be located below every half line $j<i$. With this condition, we present the algorithm described below.

```
UpperEnvelopePositiveHalfPlane():
stack={(1,x1,0)}
for i=2 to n do { // i=2, 3, …, n
  (hl, xfirst)=stack.top()
  if (wi≤whl) then continue // jump to the next iteration
  popped=true
  while ((stack.size()>0) and (popped=true)) do {
    (hl, xfirst)=stack.top()
    xcross=the crossing point of half-lines i and hl
    if (xcross≤xfirst) then {
      stack.pop()
      popped=true
    }
    else popped=false
  }
  if (stack.size()>0) then stack.push((i, xcross))
  else stack.push((i, xi,0))
}
```

Pseudocode 7-1. Algorithm for Computing the Upper Envelope of Right-Oriented Half-Lines.

We will maintain a stack, containing pairs *(hl, xfirst)*, where *hl* is the index of a half-line and *xfirst* is the smallest *x* value where the y-value of the half line *hl* is the largest among the values of all the other half lines. The pairs will be sorted in increasing order of both *hl* and *xfirst*. We will insert the half-lines in the stack in increasing order of their index. If the current half-line *i* has a slope which is smaller than the slope of the half-line *hl* at the top of the stack, then we will discard the half-line *i*, because it will not be part of the upper envelope. Otherwise, we will compute the x-value *xcross* where *i surpasses hl*. If *xcross* is smaller than the value *xfirst* of *hl*, then we remove *hl* from the top of the stack and repeat the procedure; otherwise, we insert the pair *(i, xcross)* at the top of the stack. The pseudocode is given in Pseudocode 7-1.

At the end of the algorithm, the pairs on the stack define the upper envelope of the *n* half-lines. A pair *(hl(b), xfirst(b))*, located on top of a pair *(hl(a), xfirst(a))* defines a line segment *(xfirst(a), $y_{hl(a)}$(xfirst(a))) – (xfirst(b), $y_{hl(a)}$(xfirst(b)))* of the upper envelope. The topmost pair *(hl(top), xfirst(top))* defines the last part of the upper envelope, given by the half-line *hl(top)*, starting from *x=xfirst(top)*. The time complexity of the algorithm is *O(n)*, because each half line is pushed on and/or popped from the stack at most once. Some particular situations where this algorithm can be used are when all the lines have their origins on the *OX* or the *OY* axis.

7.4.2. Interval 1-Center on a Path Network

Before proceeding to the algorithm, we should notice that if $L \geq x_n - x_1$, then all the points can be covered by one interval of length *L* and, thus, the weighted distance is *0*. Otherwise, we assign to each point *i* a right-oriented half-line with slope $wl(i)=w_i$, starting at $(x_i,0)$. We will compute the upper-envelope of these half-lines, restricted to the interval $[x_1,x_n]$.

In order to achieve this, we can use the linear time algorithm presented before, because the half-lines assigned to the *n* points satisfy the conditions required by that algorithm. Thus, the upper-envelope of these half-lines consists of *lp* points *a(1), a(2), ..., a(lp)* and *lp-1* indices *f(1), f(2), ..., f(lp-1)*, where: $a(1)=x_1$; $a(lp)=x_n$; any two intervals *(a(i),a(i+1))* are disjoint; the union of the intervals *[a(i),a(i+1)]* is $[x_1,x_n]$; the half-line associated to point *f(i)* has the largest *y-value* among all the other half-lines on the interval *[a(i),a(i+1)]*. With this upper envelope, we can compute in *O(1)* time the largest distance *dleft* from a point located to the left of an interval *[q,q+L]*, if we know the interval *[a(j),a(j+1)]* which contains *q* ($dleft=wl(f(j))\cdot(q-x_{f(j)})$).

Afterwards, we assign to each point *i* a left-oriented half-line with slope $wr(i)=w_i$, starting at

$(x_i,0)$. By running the same linear algorithm for computing the upper envelope and considering the half-lines from right to left, we obtain the upper-envelope of this second set of half-lines. More exactly, we obtain *rp* points, *b(1), …, b(rp)*, such that $b(1)=x_1$, $b(rp)=x_n$, every two intervals *[b(i),b(i+1)]* are disjoint and their union is $[x_1,x_n]$. We also obtain the *rp-1* indices *g(1), …, g(rp-1)*, meaning that the half-line assigned to point *g(i)* has the largest y-value among all the other half-lines on the interval *[b(i),b(i+1)]*. *lp* and *rp* are of the order *O(n)*.

We will now traverse in *O(n)* time the intervals *[a(i),a(i+1)]* with the left endpoint *q* of a length *L* interval, while the right endpoint *q+L* traverses the *[b(j),b(j+1)]* intervals. We start with *q=a(1)* and find the interval *[b(v),b(v+1)]* containing *q+L*.

For each position of *q* and considering that *q* is located in *[a(u),a(u+1)]* ($q<a(u+1)$) and *q+L* is located in *[b(v),b(v+1)]* ($q+L<b(v+1)$), we define $d(q)=min\{a(u+1)-q, b(v+1)-(q+L)\}$.

We need to compute the minimum value of the maximum weighted distance if the left endpoint of the interval belongs to *[q,q+d(q)]*. Let's consider $yleft_u(p)$=the value of the right-oriented half-line assigned to point *f(u)*, at the coordinate *p* and $yright_v(p)$=the value of the left-oriented half-line assigned to point *g(v)*, at the coordinate *p+L*. Thus, we want to compute $min\{max\{yleft_u(p),yright_v(p)\}| q\leq p\leq q+d(q)\}$.

The candidate values of *p* which could minimize the maximum weighted distance are *p=q*, *p=q+d(q)* and *p=peq*, where, if it exists, *peq* is the coordinate such that $yleft_u(peq)=yright_v(peq)$. Since $yleft_u$ and $yright_v$ are linear functions, we can compute *peq* in *O(1)* time by interpreting $yleft_u$ and $yright_v$ as straight lines instead as half-lines (*peq* is the x-coordinate of the point of intersection of the two lines) and then verifying if *peq* belongs to *[q,q+d(q)]*.

After this, we increment *q* by *d(q)*. If *q* becomes equal to *a(u+1)*, we increment *u* by *1*; if *q+L* becomes equal to *b(v+1)*, we increment *v* by *1*. We stop when *v>rp*.

The following table shows the running times of the algorithm described above and *Solution 1* (described previously). In *Solution 1*, the answer was searched for with a precision of *4* decimal digits. The range of the coordinates was $[0,10^8]$.

Table 7-1. Running Times: O(n·log(WMAX))-vs-O(n).

N	L	O(n·log(WMAX)) (Solution 1)	O(n)
1000000	17	3,1 sec	0,78 sec
1000000	900000	3,2 sec	0,78 sec
900000	0	3,2 sec	0,73 sec
999999	100000	3,4 sec	0,81 sec
100000	234567	0,4 sec	0,07 sec

7.5. An Algorithmic Framework for Some Optimization Problems in Tree Networks

In this section we introduce an algorithmic framework which will be used in the following section for developing (new) solutions for the connected k-center and the (restricted) connected k-median in a tree. The framework is applicable in the following situation. Let's assume that we want to compute a subset of vertices of a tree, subject to certain restrictions, such that the value of a global property is optimized. Let's also assume that we know how to compute the optimal subset, with the restriction that the subset contains a given vertex *i* (algorithm *A(i)*). This computation is performed bottom-up in *Q(n)* time, by rooting the tree at the vertex *i*.

In order to compute the optimal subset, an easy solution would be to root the tree at every vertex *i*, run the algorithm for each vertex, and choose the best solution obtained this way. This would take *O(n·Q(n))* time. For some problems, we can do better and maintain the *Q(n)* time complexity, by using a top-down approach. Let's assume that *V(r,i)* is a tuple of values computed for each vertex *i* of the tree, during the execution of the algorithm *A(r)* (with vertex *r* as the root). *V(r,i)* is computed based on the values *V(r,s(r,i,j))* of the sons of vertex *i*. For the root *r*, we can derive the optimal solution based on the values *V(r,r)*, but we cannot do this for the other vertices $i\neq r$, because the values *V(r,i)* are only based on the vertices in *T(r,i)*. Thus, we are interested in computing *V(i,i)* for each vertex *i*.

The proposed algorithmic framework consists of two stages. In the first stage, we choose an arbitrary vertex r and run the algorithm $A(r)$. This way, we compute the values $V(r,i)$ for every vertex i. The 2^{nd} stage consists of the following recursive algorithm, called with the vertex r as its (first) argument.

We only need to define the functions *UpdateRemove* and *UpdateAdd*. *UpdateRemove* computes the value $V(root,i)$ based on the same values which determine $V(i,i)$ ($V(r,s(r,i,j))$ and $V(i,parent(r,i))$), from which we must disconsider the son $s(r,i,j)=root$. *UpdateAdd* computes the value $V(i,i)$ based on the values $V(r,s(r,i,j))$ of vertex i's sons (which determine $V(r,i)$), at which we add an "extra son", the vertex $parent(r,i)$, which contributes with $V(i,parent(r,i))$.

The most straight-forward implementation is to reconsider all the current sons of a vertex (which may include its former parent and may exclude one of its former sons) and compute the required values the same way they are computed in the algorithm A. Let's assume that algorithm A takes $O(ns(r,i)^c)$ time to compute the values for a vertex i (thus, $Q(n)=O(n^c)$). If we use the straight-forward implementation, the extra time complexity for each edge $(parent(r,i),i)$ would be $O(ns(r,parent(r,i))^c)$. Thus, the overall time complexity would be $O(n^{c+1})=O(n \cdot Q(n))$.

In case the degree of every vertex is bounded by a constant D, then the time complexity remains $O(Q(n))$ and the straight-forward implementation is acceptable. Even if the degree is not bounded, implementing things this way may still produce large improvements over the obvious solution, in terms of running time. However, if we consider the star with one central vertex and $n-1$ leaves, we can see that we obtain no improvement over the obvious solution of considering every vertex as a possible root of the tree.

```
TopDownTraversal(i, r):
if (i≠r) then {
  Compute(parent(r,i), i, r); Compute(i, i, r)
}
for j=1 to ns(r,i) do TopDownTraversal(s(r,i,j), r)
Compute(i, root, r):
if (i≠root) then V(root,i)=UpdateRemove(i, root)
else V(i,i)=UpdateAdd(i,  parent(r,i), r)
```

Pseudocode 7-2. Functions of the Algorithmic Framework for Optimization Problems in Trees.

7.5.1. Connected K-Center and K-Median

Let's assume that we have a tree, rooted at a vertex r. Each tree edge $(parent(r,i),i)$ has a weight $w(r, parent(r,i), i)$, such that $w(r,parent(r,i),i))>w(r, i, p)$ (for any i and $p \neq parent(r,i)$); that is, the weight of the edges $(parent(r,i), i)$ are larger than the weights of the edges in $T(r,i)$. We want to find a connected subset S composed of k tree vertices, such that $r \in S$ and the sum (maximum) of the weights of the edges not connecting two vertices u and v from S is minimized.

A connected subset composed of k vertices implies that there are $k-1$ edges whose weights are not added to the sum (maximum). The best solution would be obtained if we could select the $k-1$ edges with the largest weights. A problem that could occur is that the subset of the endpoints of the $k-1$ largest edges is not connected. However, as it was proven in [Yen and Chen, 2006], it is impossible for this subset not to be connected. Let's assume that the subset is not connected. That means that there is an edge $(parent(r,u),u)$ which is not part of the set of $k-1$ edges, but some edges in $T(r,u)$ were selected. This is impossible, because $w(r,parent(r,u), u)$ is larger than the weights of the edges in $T(r,u)$. Thus, if we selected an edge in $T(r,u)$, we must have also selected the edge $(parent(r,u), u)$.

In order to select the $k-1$ largest edges, we insert all the $n-1$ tree edges into a max-heap HT, from which we extract the desired edges. Thus, in $O(n \cdot log(n))$ time, we can solve this problem. Let's assume that the problem is extended as follows. If we choose a different vertex r' as the root of the tree, the weights of the edges do not stay the same. However, they change according to the following restrictions. When considering as the new tree root a vertex r' which is a son of r, only

the weight of the edge *(r',r)* changes (it does not necessarily increase, but its new weight is larger than all the weights of the edges located in *r*'s subtree). We will use the algorithmic framework introduced in the previous section.

Let's assume that we have a min-heap *HS* which maintains all the edges in the solution for the vertex *r* as the root (the other edges are in *HT*). In the *UpdateRemove* function, if *(r,r')* is in *HS*, we remove it from *HS* and insert it in *HT*; otherwise, we remove the edge with the minimum weight from *HS* and insert it in *HT*. In the *UpdateAdd* function, we add the edge *(r',r)* to *HS*, considering its new weight (removing it from *HT*).

For the min-sum case, we will also maintain the sum *esum* of the edges which are outside *HS*. Initially, *esum*=the sum of all the edges in *HT*. Whenever we remove (insert) an edge from (into) *HT*, we decrease (increase) *esum* by the (current) weight of that edge. At any time, the maximum weight of an edge in *HT* (for the min-max case) or the value of *esum* (for the min-sum case) are the cost corresponding to the current root. Thus, in $O(n \cdot log(n))$ time, we can compute the optimal solutions for every vertex *i* as the tree root (the alternative would have been to root the tree independently at every vertex and recompute the solution from scratch every time – this would have taken $O(n^2 \cdot log(n))$ time).

We will now show how the (unweighted) connected k-center and a restricted version of the weighted connected k-median problem can be solved using the generic solution presented above, in $O(n \cdot log(n))$ time.

7.5.1.1. Unweighted Connected K-Center in Trees

We assume that every edge *(u,v)* of the tree has a length (e.g. delay, latency) *l(u,v)*. We want to find a connected subset *S* composed of *k* vertices, such that the maximum distance from a vertex outside of *S* to the nearest vertex in *S* is minimized. The distance from a vertex *u* to a vertex *v* is *d(u,v)*, the sum of the lengths of the edges on the (unique) path between *u* and *v*. The vertices in *S* need to form a connected subtree.

We will root the tree at an arbitrary vertex *r* and we will want to compute a connected subset *S* of *k* vertices which contains *r*. For each vertex *i*, we compute *lmax(r,i)*=the length of the longest path starting at *i* and ending at a vertex in *T(r,i)*, and *lmax2(r,i)*=the length of the 2^{nd} longest path starting at *i* and ending in *T(r,i)*.

We assign a weight *we(r,parent(r,u),u)* to each edge *(parent(r,u),u)*: *we(r,parent(r,u),u)=l(parent(r,u),u)+lmax(r,u)*. It is obvious that the weight of the edge *(parent(r,u),u)* is larger than all the weights of the edge in *T(r,u)*. Furthermore, if we choose a connected subset of *k* vertices containing *r*, the maximum distance from a vertex outside of *S* to its closest vertex in *S* will be equal to the largest weight of an edge which does not connect two vertices in *S*. Thus, it is optimal to choose the *k-1* edges having the largest weights.

When changing the root of the tree (a son *r'* of the previous root *r* is lifted as the new root), only the weight of the edge *(r',r)* changes. We will have *lmax(r',r)= lmax(r,r)* if the path corresponding to this value did not pass through the vertex *r'*; otherwise, *lmax(r',r)=lmax2(r,r)*. For *r'*, *lmax(r',r')* and *lmax2(r', r')* are the smallest two values in the (multi)set *{lmax(r,r'), lmax2(r,r'), l(r',r)+lmax(r',r)}*. The new weight of the edge *(r,r')* will be *l(r,r')+lmax(r',r)*. We can now easily use the algorithm presented previously. If *k=1*, then we don't need to use the heaps, as only the values *lmax(i,i)* are of interest, obtaining an *O(n)* solution for the unweighted tree center problem.

7.5.1.2. Connected K-Median in Trees

Each vertex *u* is enhanced with a weight *wv(u)* (e.g. expected amount of data transferred from the servers). The k-median problem asks for a connected subset *S* composed of *k* vertices, such that the total sum of weighted distances from the vertices outside *S* to the corresponding closest vertex in *S* is minimized. The weighted distance from a vertex *u* to a vertex *v* is *dw(u,v)=wv(u)·d(u,v)*, where *d(u,v)* is the sum of the lengths of the edges on the (unique) path between *u* and *v*.

For each vertex *i*, we will compute *wvt(r,i)*=the sum of the weights of the vertices in its

subtree; $wvt(r,i)=wv(i)+wvt(r,s(r,i,1))+ \dots +wvt(r,s(r,i,ns(r,i)))$ (if i is a leaf, $wvt(r,i)=wv(i)$).

We will assign to each edge $(parent(r,i),i)$ a weight $we(r,parent(r,i),i)= wvt(r,i)\cdot l(parent(r,i),i)$. In the general case, this weight function does not imply the property required by the algorithm described previously (that $we(r,parent(r,i),i)$ is larger than the weight of any edge in $T(r,i)$). We will restrict our attention to situations in which this property holds, e.g. when all the edge lengths are equal.

The minimum cost criterion (for a subset containing the vertex r) implies that the sum of the weights of the edges not connecting two vertices in S must be minimized. Thus, we again want to choose the $k-1$ edges with largest weights, which will form a connected subset of vertices. When lifting a vertex r' as the new root (above the previous root r), we have: $wvt(r',r)=wvt(r,r)-wvt(r,r')$ and $wvt(r',r')=wvt(r,r)$. The new weight of the edge (r',r) is $wvt(r',r)\cdot l(r',r)$. All the other edge weights and $wvt(*)$ values stay the same. Thus, we can use the presented algorithm.

7.6. Optimal Replica Placement in Tree Networks

A tree is an undirected, connected, acyclic graph. A tree may be rooted, in which case a special vertex r will be called its root. Even if the tree is unrooted, we may choose to root it at some vertex. In a rooted tree, we define $parent(i)$ as the parent of vertex i and $ns(i)$ as the number of sons of vertex i. For a leaf vertex l, $ns(l)=0$ and for the root r, $parent(r)$ is *undefined*. The sons of a vertex i are denoted by $s(i,j)$ $(1\le j\le ns(i))$. A vertex j is a *descendant* of vertex i if $(parent(j)=i)$ or $parent(j)$ is also a descendant of vertex i. We denote by $T(i)$ the subtree rooted at vertex i, i.e. the part of the tree composed of vertex i, all of its descendants and all the edges connecting them. $P(u,v)$ denotes the (unique) path between vertices u and v.

We consider a tree with n vertices, where every edge (u,v) has $Q+3\ge 3$ weights $we(u,v,d)$ $(1\le d\le Q+1)$ and every vertex v has Q weights $wv(v,d)$ $(1\le d\le Q)$, we want to identify $k\ge 1$ disjoint (not necessarily connected) subsets of vertices, satisfying the constraints: $szmin(j,d)\le sz(j,d)\le szmax(j,d)$ $(1\le j\le k;\ 1\le d\le Q-1)$, where $sz(j,d)$ is the aggregate agg_d of the weights $wv(v,d)$ and $we(u,v,d)$ of the vertices v and the edges (u,v) in the j^{th} subset (an edge (u,v) belongs to the j^{th} subset if both u and v belong to this subset).

$szmin(*)$ and $szmax(*)$ are given as lower and upper bounds on the subset sizes. The cost employed by the subsets is equal to the aggregate agg_{opt} of the weights $wv(v,Q)$, $we(u,v,Q)$, $we(u',v',Q+1)$, $we(u'',v'',Q+2)$ and $we(u''',v''',Q+3)$ of the vertices v and of the edges (u,v) and (u',v') such that:

- the vertex v does not belong to any subset
- u and v belong to the same subset
- $u'$and v' belong to different subsets
- one of u'' or v'' belongs to a subset and the other belongs to no subset
- neither u''' nor v''' belong to any subset

The vertices in each subset will store one replica of one of k replicated files in the network. We will root the tree at an arbitrary vertex r. Then, for each vertex i, we compute $C_{opt}(i, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$=the optimal cost (relative to the optimum function opt) of obtaining k subsets from the vertices of $T(i)$ such that the aggregate of the d weights in the a^{th} subset is $sz(a,b)$, with vertex i belonging to the set j (if $j=0$, then i does not belong to any subset). We will denote by e_d the neutral element of the agg_d operation, by e_{opt} the neutral element of the agg_{opt} operation, and by ∞ the worst values for the agg_{opt} aggregate. Each vertex v must have at least one weight $wv(v,d)\ne e_d$ $(1\le d\le Q-1)$ (otherwise, we can increase Q by 1 and add a new weight to each vertex, such that the property is sastisfied).

For a leaf vertex l we have $C_{opt}(l,\ 0,\ sz(*,1\le d\le Q-1)=e_d)=wv(l,Q)$, $C_{opt}(l,\ j\ge 1,\ sz(j,1\le d\le Q-1)=wv(l,d),\ sz(a\ne j,1\le d\le Q-1)=e_d)=e_{opt}$ and $C_{opt}(l,\ j,\ sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))=\infty$ for all the other tuples $(j,\ sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$ except the ones mentioned earlier.

For a non-leaf vertex i, we compute the auxiliary table $C_{aux}(i,\ q,\ j,\ sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$, having the same meaning as $C_{opt}(*)$, except that only the first q sons are considered.

We have $C_{aux}(i, 0, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$ as if i were a leaf (i.e. we have non-∞ values only for $j=0$ and $sz(*,1\le d\le Q-1)=e_d$, and for $j\ge 1$, $sz(j,1\le d\le Q-1)=wv(i,d)$ and $sz(a\ne j,1\le d\le Q-1)=e_d$). For each tuple $(q\ge 1, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$ we first initialize $C_{opt}(i, q, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))=\infty$ and then we consider all the tuples $(q-1, j', sz'(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$ (with $e_d\le sz'(a,d)\le sz(a,d)\ agg_d^{-1}\ we(i,s(i,q),d)$).

For each considered tuple we set $C_{aux}(i, q>0, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))=opt\{C_{aux}(i, q, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1)), C_{aux}(i, q-1, j, sz(a,d)\ agg_d^{-1}\ sz'(a,d)\ agg_d^{-1}\ we(i,s(i,q),d)\ (1\le a\le k;\ 1\le d\le Q-1))\ agg_{opt}\ C_{opt}(s(i,q), j', sz'(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))\ agg_{opt}\ extra\_cost(j, j', i, s(i,q))\}$.

We have $extra\_cost(j, j', u, v)=(if\ ((j=0)\ and\ (j'=0))\ then\ we(u,v,Q+3)\ else\ if\ (((j=0)\ and\ (j'\ne 0))\ or\ ((j\ne 0)\ and\ (j'=0)))\ then\ we(u, v, Q+2)\ else\ if\ (j\ne j')\ then\ we(u,v,Q+1)\ else\ we(u,v,Q))$. $C_{opt}(i, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))=C_{aux}(i, ns(i), j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$. For the agg_d operation, the $\le$ relation is defined such that e_d is the best value and ∞ is the worst value.

If all the weights $wv(*,d)$ are ∞ (each vertex must belong to a subset), we can drop the values $sz(k,d)$ from the state definition (and $sz'(k,d)$ from the states of a vertex i's son), because it can be derived from the values $sz(1\le j\le k, d)$.

For a vertex i, $sz(k,d)=wagg(T(i),d)\ agg_d^{-1}\ (sz(1,d)\ agg_d\ \dots\ agg_d\ sz(k-1,d))$ $(sz'(k,d)=wagg(T(s(i,q)),d)\ agg_d^{-1}\ (sz'(1,d)\ agg_d\ \dots\ agg_d\ sz'(k-1,d))$. For an auxiliary state (i,q) (vertex i and its first q sons), $sz(k,d)=wv(i,d)\ agg_d\ wagg(T(s(i,1)),d)\ agg_d\ \dots\ agg_d\ wagg(T(s(i,q)),d)\ agg_d^{-1}\ (sz(1,d)\ agg_d\ \dots\ agg_d\ sz(k-1,d))$. We denote by $wagg(T(a), d)$=the aggregate agg_d of the weights $wv(v,d)$ of the vertices v in $T(a)$. For a leaf vertex a we have $wagg(T(a),d)=wv(a,d)$; for a non-leaf vertex a we have $wagg(T(a),d)=wv(a,d)\ agg_d\ (wagg(T(s(a,1)),d)\ agg_d\ \dots\ agg_d\ wagg(T(s(a,ns(a))),d))$. The presented technique is, in fact, an application of the multidimensional tree knapsack method.

If the subsets are required to be connected, then, when computing $C_{aux}(i, q, j\ge 1, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$, we will only consider values of the form $C_{opt}(s(i,q), j'=j, sz'(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))$ or $C_{opt}(s(i,q), j'\ne j, sz'(a,d)\ (1\le a\le k;\ 1\le d\le Q-1)$ with $sz'(j,d)=e_d\ (1\le d\le Q-1$).

The problem can be extended to the case when $wv(v,d)$ for every vertex v and $we(u,v,d)$ for every edge (u,v) are not weights, but sets of weights from which one value can be chosen. In this case, for a leaf l, we set $C_{opt}(l, 0, sz(*,1\le d\le Q-1)=e_d)=opt\{wv(l,Q)\}$. Then, we consider every weight w in $wv(l)$ and set $C_{opt}(l, j\ge 1, sz(j,1\le d\le Q-1)=w, sz(a\ne j,1\le d\le Q-1)=e_d)=e_{opt}$.

For a non-leaf vertex i, we initialize $C_{aux}(i, 0, *, \dots, *)$ as if i were a leaf. Then, when computing $C_{aux}(i, q\ge 1, *, \dots, *)$, we will set $C_{aux}(i, q>0, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))=opt\{C_{aux}(i, q, j, sz(a,d)\ (1\le a\le k;\ 1\le d\le Q-1)), C_{aux}(i, q-1, j, sz(a,d)\ agg_d^{-1}\ sz'(a,d)\ agg_d^{-1}\ w\ (1\le a\le k;\ 1\le d\le Q-1))\ agg_{opt}\ C_{opt}(s(i,q), j', sz'(a,d)\ (1\le a\le k;\ 1\le d\le Q-1))\ agg_{opt}\ extra\_cost(j, j', i, s(i,q))\ |\ w\in we(i,s(i,q),d), e_d\le sz'(a,d)\le sz(a,d)\ agg_d^{-1}\ w\}$. In the *extra_cost* function we replace $we(u, v, *)$ by $opt\{we(u, v, *)\}$. This situation corresponds to the multiple choice multidimensional tree knapsack problem.

7.7. The Balanced Content Replication Problem on Trees

In the balanced content replication problem, we are given k pieces of content of equal importance, which have to be placed in the n vertices of a tree network. Within each vertex, only a single piece of content can be placed. For each piece of content i ($1\le i\le k$), we define nv_i as the number of vertices in which the piece was placed. Because each piece is of equal importance, the number of pieces placed in two different vertices should be approximately equal. More formally, $|nv_i-nv_j|\le 1$, for any two pieces of content i and j.

Each vertex of the tree is a server used both for storing a replica of some piece of content and for serving client requests. A client may require any piece of content and the server will get that piece from the nearest server possessing it. In order to minimize network traffic, it is desirable that replicas of the required piece of content be found very close to the server. In particular, the traffic is kept low if either the server possesses that piece of content or one of its neighbors does. In order to maximize the chance that some of its neighbors possess a required piece of content that the server does not possess, any two neighboring servers should not store the same piece of content.

The problem translates into an *equitable coloring* of the tree network, using exactly k colors.

A supplementary assumption that we consider is that the number of pieces is not smaller than the maximum number of neighbors a server has, i.e. that the number of colors is greater than or equal to the maximum degree of any vertex of the tree.

We will present next a greedy algorithm for solving the k-equitable coloring problem in trees. Afterwards, we will present a linear time solution for the unrestricted vertex multicut (UVMC) problem in trees. Then, we define a new reliability metric for trees which is based on the UVMC problem. This reliability metric is evaluated on the output of the k-equitable coloring algorithm on multiple types of trees.

7.7.1. A Greedy Algorithm for the K-Equitable Coloring Problem in Trees

We will start with some definitions. If an edge *(i,j)* belongs to the tree, then *i* is a neighbor of *j* and *j* is a neighbor of *i*. The degree deg_i of a vertex *i* is equal to the number of its neighbors. The maximum degree of the tree is:

$$D = \max_{\text{i is a vertex in the tree}} \{\deg_i\}. \quad \textbf{(7-5)}$$

If the tree has one or two vertices, then finding an equitable coloring is trivial. Another trivial situation is if the number of colors *k* is greater than or equal to *n*, because in this case, each vertex can be colored with a different color. Therefore, we will only consider the case $n \geq 3$ and $k < n$ (and $k \geq D$).

We will transform the tree into a rooted tree, by choosing a vertex *r* as the root. This vertex can be any vertex whose degree is less than *D*. For $n \geq 3$ vertices, this is always possible. For instance, the root can be any vertex of degree *1* (such a vertex always exists), because the maximum degree *D* is greater than *1*. Considering the rooted tree, each vertex has a parent (except the root) and all of its neighbors except for its parent become its sons. Vertex *i* has ns_i sons. Each vertex of degree *D* has *D-1* sons, which is the maximum number of sons any vertex may have. In an equitable coloring of a subtree or of a forest, we will call *c* a *surplus* color if there is one extra vertex colored with *c*, compared to the color having the minimum number of vertices colored with it. If the total number of vertices in the subtree is divisible by *k*, there will be no surplus colors.

The algorithm will compute an equitable coloring for each subtree of the tree, in a bottom-up fashion (from the leaves towards the root). When finding an equitable coloring for the subtree rooted at vertex *i*, several values will be computed (we denote by *A mod B* the remainder of the integer division of *A* and *B*):

- $nvtotal_i$ = the number of vertices in vertex *i*'s subtree (including *i*).
- $ncplus_i = (nvtotal_i \bmod k)$ – the number of surplus colors in an equitable coloring of vertex *i*'s subtree.
- $color_i$ = the color of vertex *i* in an equitable coloring of its subtree.
- $color\_perm_i$ = a permutation which describes how the colors in vertex *i*'s subtree should be relabeled.

During the first bottom-up traversal of the tree, the actual colors of the vertices are not fully computed. For each vertex *i*, we will know its color in an equitable coloring of the subtree rooted at *i*. This color will not necessarily be the final color of the vertex, because some of its ancestors might choose to relabel the colors in vertex *i*'s subtree.

Relabeling colors is the main mechanism employed by the described algorithm. The relabeling is described as a permutation *p*, where the values *y=p(x)* have the meaning that if a vertex was assigned color *x*, then it will be reassigned the color *y*. The *color_perm* permutations form a hierarchy of relabeling permutations.

The actual color of vertex *i* will be obtained by first composing all the *color_perm* permutations on the path from the root to vertex *i* into a permutation *p* and then assigning to vertex *i* the color $p(color_i)$. For instance, if the path from the root *r* to the vertex *i* is composed of the vertices $r=v_1, v_2, ..., v_q=i$, then $p=color\_perm_{v1} \cdot color\_perm_{v2} \cdot \ldots \cdot color\_perm_{vq}$. we will present next how to compute all the values mentioned above, especially the *color_perm* permutations, which will be used in a second top-down traversal of the tree.

If vertex i is a leaf, then vertex i's color will be set to *1* and the *color_perm* permutation will be set to the identity permutation *(1,2,..,k)*. If i is not a leaf, then an equitable coloring will be found for the subtree rooted at each son of i, independently. In order to combine all the colorings of the subtrees of vertex i's sons, some colors will have to be relabeled, i.e. for some of the sons, the *color_perm* permutation will need to be changed. An entry $color\_perm_j(c)$ means that every vertex that was colored with color c in j's subtree will need to be recolored with the color $color\_perm_j(c)$. Obviously, relabeling the colors in a subtree will not change the equitable coloring of that subtree (the actual colors of the vertices will be changed, but the difference between the number of vertices colored with any two distinct colors will still remain at most *1*).

The computation of an equitable coloring for the subtree rooted at a vertex i will maintain the following **invariant**: the $ncplus_i$ surplus colors will be the colors $1,2,...,ncplus_i$. Vertex i's color will be $ncplus_i$, if $ncplus_i>0$, or *1*, otherwise.

We will now explain how to combine the equitable colorings of the subtrees rooted at vertex i's sons into an equitable coloring of vertex i's subtree. We will consider vertex i's sons in some arbitrary order $s_1, s_2, ..., s_{nsi}$. After considering all the ns_i sons, their colors will belong to the set $\{1,...,k-1\}$, so that it will be possible to assign color k to vertex i. Furthermore, the color k will not be a surplus color, so assigning color k to vertex i will lead to an equitable coloring of vertex i's subtree. After assigning color k to vertex i, we will change $color\_perm_i$ accordingly, in order to maintain the invariant.

After considering the first $j-1$ sons, the number of surplus colors will be:

$$\text{cplus}_{\text{j-1}} = \left(\sum_{\text{p=1}}^{\text{j-1}} \text{nvtotal}(\text{s}_\text{p}) \right) \bmod \text{k} \,. \qquad \textbf{(7-6)}$$

Moreover, the colors $\{1,2,..,cplus_{j-1}\}$ will be the surplus colors. When reaching the j^{th} son, each of the first j sons is in one of the following two states: *active* or *inactive*. If $ncplus_{sj}=0$, then s_j is inactive, otherwise s_j is active. If ($ncplus_{sj}>0$ and $cplus_{j-1}=0$) or ($cplus_{j-1} + ncplus_{sj} > k$), then all the sons $s_1,...,s_{j-1}$ are made inactive and s_j will be the only active son. A counter c_{active} is maintained, storing the number of currently active sons.

If the j^{th} son is active, the colors in its subtree will be permuted in a cyclic manner, such that color c ($1\le c\le k$) becomes color $((cplus_{j-1}+c-1) \bmod k)+1$ (this change is applied to the $color\_perm_j$ permutation). Then, s_j's subtree is added to the forest composed of the subtrees of the first $j-1$ sons. With this addition, the invariant that the first $cplus_j$ colors are the surplus colors still holds.

After that, s_j's color will be relabeled (whether it is active or not), according to some rules we will mention in the following paragraph. If s_j is *active*, this relabeling needs to be "visible" to all the previous sons, but must not be "visible" to the sons which were not considered yet, i.e. it must also relabel the color classes of the previously considered sons, but not those of the sons which were not yet considered.

This can be achieved by applying the relabeling directly to the $color\_perm_{sp}$ permutations of every son s_p ($p\le j$), but this would lead to a $O(n^2 \cdot k)$ algorithm. Instead, we will maintain a stack of relabeling permutations. Then, after considering all the sons, we will need to compose all the permutations on the stack, from the top down to level *lev* into a permutation p_{lev} and then replace $color\_perm_{slev}$ by $p_{lev} \cdot color\_perm_{slev}$, for $1\le lev\le ns_i$. All the p_{lev} permutations can be computed in $O(k \cdot ns_i)$ time overall, so this maintains the time complexity of the algorithm to $O(n \cdot k)$.

As stated in the previous paragraph, s_j's color will be relabeled. If s_j is an *inactive* son, then we will swap its color with color $k-1$. This swap will be represented as a relabeling permutation and can be applied to the $color\_perm_{sj}$ permutation only. The swap does not need to be visible to any of the other sons. Therefore, we will apply the swap to the $color\_perm_{sj}$ permutation and push on the stack the identity permutation.

If s_j is an *active* son, we would like to swap s_j's color with the color indicated by the counter c_{active}. This can also be achieved using a simple relabeling permutation, which swaps the two color classes. However, this could cause some problems, for instance, if s_j's color is k, because then the color c which will be relabeled to k might have been assigned to some other son. If this happens, it will be impossible to assign color k to vertex i in the end and the algorithm will be incorrect.

```
GreedyEquitableColoringPhase1(i, k):
if (ns_i=0) then {
  color_perm_i=(1,2,..,k) // the identity permutation
  color_i=nvtotal_i=1
  ncplus_i=1 mod k
  return
}
// find an equitable coloring for each of vertex i's sons
for each j=1 to ns_i do
  GreedyEquitableColoringPhase1(s_j, k)
nvtotal_i=cplus_0=c_active=total_active=0
stack=empty
for j=1 to ns_i do {
  ntotal_i=nvtotal_i+nvtotal_sj
  if (ncplus_j=0) then {
    Swap2(s_j, k-1, k)
    stack.push((1,2,..,k))
  } else {
    CyclicPermutation(s_j, cplus_j-1, k)
    c_active=c_active+1
    total_active=total_active+1
    if (cplus+ncplus_sj>k) then c_active=1
    soncolor=color_perm_sj(color_sj).
    if (soncolor>total_active) then stack.push(Swap3(s_j, c_active, total_active, k))
    else {
      perm=(1,2,..,k)
      perm(soncolor)=c_active
      perm(c_active)=soncolor
      stack.push(perm)
    }
    if (cplus_j-1+ncplus_j=k) then c_active=0
  }
  cplus_j=nvtotal_i mod k
}
// empty the stack
lev=ns_i
p_lev+1=(1,2,..,k)
while (not stack.isEmpty()) do {
  p_lev=ComposePermutations(p_lev+1, stack.top(), k)
  color_perm_slev=ComposePermutations(p_lev, color_perm_slev,k)
  stack.pop()
  lev=lev-1
}
// choose a color for the vertex i
color_i=k
color_perm_i=(1,2,..,k)
nvtotal_i=nvtotal_i+1
ncplus_i=(cplus+1) mod k
Swap2(i, cplus+1, k) // relabel vertex i's color with cplus+1
```

Pseudocode 7-3. First Stage of the K-Equitable Coloring Algorithm.

The solution, however, is simple. We will swap not just two colors, but three. We will swap s_j's color with some color c which was not assigned to any previous son, and after that swap the

color c with the color c_{active}. These swaps can also be described by a relabeling permutation. Finding a color c not assigned to any previous son is easy: we will maintain a counter $total_{active}$, denoting the total number of sons which have ever been active (including s_j). Then, the color $total_{active}$ is just the color we need. The relabeling permutation will be pushed on the stack, as it needs to be visible to all the sons s_p ($p<j$). This relabeling using three colors will be used only if s_j's current color is greater than the value of $total_{active}$. Otherwise, the invariant that the surplus colors are the first ones will not hold.

```
GreedyEquitableColoringPhase2(i, k, stack):
real_color_perm = ComposePermutations(stack.top(), color_perm_i, k)
stack.push(real_color_perm)
real_color_i = real_color_perm(color_i)
for j=1 to ns_i do {
  GreedyEquitableColoringPhase2(s_j, k)
}
stack.pop()
```

Pseudocode 7-4. Second Stage of the K-Equitable Coloring Algorithm.

```
CyclicPermutation(j, offset, k):
for c=1 to k do {
  color_perm_j(c)=((color_perm_j(c)+offset-1) mod k) + 1
}
Swap2(j, newcol, k):
oldcol=color_perm_j(color_j)
find c' such that color_perm_j(c')=newcol
color_perm_j(c')=oldcol
color_perm_j(color_j)=newcol

ComposePermutations(p1, p2,k):
for c=1 to k do
  p_result(c)=p1(p2(c))
return p_result

Swap3(j, newcol, auxcol, k):
perm=(1,2,..,k)
oldcol=color_perm_j(color_j)
perm(oldcol)=newcol
perm(newcol)=auxcol
perm(auxcol)=oldcol
return perm
```

Pseudocode 7-5. Auxiliary Functions for the K-Equitable Coloring Algorithm.

After adding all of vertex i's sons, the obtained forest is equitably colored, the colors of vertex i's sons belong to the set $\{1,..,k-1\}$ and the first $cplus_{nsi}$ ($0 \le cplus_{nsi} < k$) colors are the surplus colors. By assigning the color k to vertex i, the coloring is kept equitable and valid. All that remains to be done is to relabel vertex i's color with $cplus_{nsi}+1$, in order to maintain the invariant that the $ncplus_i$ surplus colors in an equitable coloring of vertex i's subtree are the colors $1,2,...,ncplus_i$. This is accomplished by swapping the colors k and $cplus_{nsi}+1$ in the $color\_perm_i$ permutation.

In order to find the actual color of each vertex, we will have to traverse the tree again, starting from the root (in a top-down fashion this time). We will maintain a stack of color permutations. The first permutation pushed on the stack will be the identity permutation. When going from some vertex i to one of its sons j, we will compose the color permutation on the top of the stack with $color\_perm_j$ and push this permutation on the stack. The permutation at the top of the stack will then be used for finding the real color of vertex j. When returning from a son j to its parent i, the permutation from the top of the stack is popped.

It is easy to notice that both parts of the algorithm (the bottom-up computations and the top-down assignment of real colors) take $O(n \cdot k)$ time. The two stages of the algorithm are described in Pseudocode 7-3 and 7-4. The auxiliary functions used by the algorithm are given in Pseudocode 7-5.

7.7.2. The Unrestricted Vertex Multicut Problem

We are given a connected graph G with V vertices and E edges, as well as H critical pairs $(s_1,t_1), ..., (s_H,t_H)$. The *Unrestricted Vertex Multicut* (*UVMC*) problem asks for the minimum number

of vertices which need to be removed in order to disconnect every critical pair of vertices, i.e. at least one vertex must be removed from the path between the two vertices of each critical pair (possibly even one or both vertices of the pair). When G is a tree, a simple polynomial time algorithm is presented in Pseudocode 7-6.

Step 1. ***root*** *the tree at some vertex r and compute the parent-son relationships for all the vertices.*
Step 2. ***for*** *each critical pair* (s_i,t_i) ***do****: compute its LCA and the level of the LCA (the distance from the LCA to the root)*
Step 3. ***sort*** *all the critical pairs in non-increasing order of the level of their LCA*
Step 4. ***for*** *each critical pair* (s_i, t_i)*, in the sorted order,* ***do***
Step 4.1. ***if*** s_i *and* t_i *are not already disconnected* ***then***
Step 4.1.1. ***remove*** *the LCA of* s_i *and* t_i *from the tree*

Pseudocode 7-6. The Generic Algorithm for the Unrestricted Vertex Multicut Problem in Trees.

The number of vertices removed at Step *4.1.1* is the minimum number of vertices which need to be removed in order to disconnect all the H critical pairs. The algorithm presented above can easily be implemented in time $O(V \cdot H)$. Step 1 only takes $O(V)$ time. Computing the *LCA* of each pair in Step *2* can be done in $O(V)$ time, so Step *2* takes $O(V \cdot H)$ time overall. Step *3* can be performed in $O(H \cdot log(H))$ time. The connectivity test at Step *4.1* can be performed in $O(V)$ time. Multiplying this by H, we obtain an $O(V \cdot H)$ time complexity for Step *4*.

7.7.3. A Linear Time Algorithm for the UVMC Problem on Trees

The algorithm presented in the previous section has an obvious $O(V \cdot H)$ implementation. However, using more intelligent techniques, it can be implemented in $O(V+H)$ time.

Step *2* of the algorithm can be computed in time $O(V+H)$ for all the critical pairs. In order to achieve this, we use the algorithm presented in [Bender and Farach-Colton, 2000]. The rooted tree is preprocessed in $O(V)$ time. An array E containing the Euler tour of the tree traversal is produced, as well as an array L with the levels of the vertices, in the order in which they are encountered in the Euler tour. From the first array, a representative array R is computed: for each vertex u, $R[u]$ represents the position of the first occurence of u in E. Now, in order to compute the *LCA* of two vertices u and v, we need to find the vertex having the minimum level and which is located between $R[u]$ and $R[v]$ in E. This is performed in $O(1)$ time, by using a technique called *Range Minimum Query* (RMQ). The array L is first preprocessed in $O(V)$ time, by splitting it into blocks of suitable sizes. Then, using this preprocessing, the minimum value between two given positions can be found in $O(1)$ time. Therefore, the time complexity is $O(H)$ for all the critical pairs.

At the end of this step, we have two new arrays, *pLCA* and *level*, where $pLCA[i]$ is the lowest common ancestor of the i^{th} pair of vertices and $level[i]$ is the level of the i^{th} pair's *LCA*. All the procedures presented in this paragraph are described in detail in [Bender and Farach-Colton, 2000] and, as mentioned, lead to an $O(V+H)$ time complexity. Implementing the other steps in $O(V+H)$ is an original contribution of this book.

Step *3* of the algorithm can be implemented in $O(V+H)$ time, by using an array of linked-lists LL. For each critical pair i, we will add the pair's index at the beginning (or at the end) of the linked-list $LL[level[i]]$, in $O(1)$ time. Since $level[i]$ is between 0 and $V-1$, the array LL only has V entries. Now, in order to sort the pairs, we will traverse the entries of LL from $V-1$ down to 0. If $LL[i]$ is not empty, then we will traverse this linked-list and each element of the list is added to the end of an array *sorted_pairs*. The array *sorted_pairs* will contain the pairs in non-increasing order of their *LCA*'s level. It is obvious that, because of the order of the traversals, pairs with the *LCA* on larger levels (located lower in the tree) will be located before pairs with the *LCA* on smaller levels (located higher in the tree) in the *sorted_pairs* array. The overall complexity of this step is $O(V+H)$. Creating the LL array takes $O(V)$ time, inserting all the critical pairs in LL takes $O(H)$ time and traversing the linked-lists in LL takes $O(V+H)$ time.

Implementing Step *4* in *O(V+H)* time is the trickiest part of the algorithm. We will maintain an array of boolean values, *marked*. For each vertex of the tree, this array will tell us if the vertex was marked or not. Initially, no vertex is marked. We will consider the critical pairs in the order produced at Step *3*. Checking if the two vertices of the pair are disconnected will be performed in *O(1)* time, by simply inspecting the *marked* array. If at least one of the two vertices was marked, then the two vertices were disconnected because of the removal of the *LCA* of a previous pair. If they are still connected, we will "remove" and mark their *LCA*, as well as mark all the unmarked vertices located in their *LCA*'s subtree.

This time, removing a vertex from the tree does not mean deleting it from the tree, together with the incident edges. The tree is not modified, only a counter with the number of "removed" vertices is increased.

Let's first analyze the correctness of this algorithm. The part that needs to be considered is the connectivity test for the two vertices s_i and t_i of a critical pair *i*. For this, we use Theorem *1*.

Theorem 1: *When considering a critical pair (s_i,t_i) in Step 4 of the algorithm and (marked[s_i]=True or marked[t_i]=True), then the two vertices of the pair have already been disconnected.*

Proof. Without loss of generality, we will consider that vertex s_i is marked (t_i could be marked, too). If s_i is marked, this is because some ancestor *k* of s_i (a vertex on the path from s_i to the root of the tree) was removed and this lead to all the vertices in *k*'s subtree being marked. This ancestor *k* was the *LCA* of the vertices of a pair considered in Step *4* before the pair *i*. Because the levels of the *LCA*s of the pairs are sorted in non-increasing order, vertex *k*'s level must be greater than or equal to *level[i]*. Note that all the ancestors of s_i whose level is greater than or equal to *level[i]* are located on the path between s_i and *pLCA[i]* (including the endpoints of the path, too). Therefore, we conclude that at least one vertex on the path between s_i and *pLCA[i]* was removed previously.

Now it is easy to prove that the vertices s_i and t_i are disconnected. In the tree, the path from s_i to t_i is unique. We will consider this path as being composed of two parts: the path from s_i to *pLCA[i]* and the path from *pLCA[i]* to t_i. Since we know that at least one vertex on the path from s_i to *pLCA[i]* was removed previously, this means that at least one vertex on the path from s_i to t_i was removed, which concludes our proof.

```
TraverseAndMark(v):
  marked[v]=True
  for each w ∈ sons_v do
    if (not marked[w]) then
      TraverseAndMark(w)
```

Pseudocode 7-7. The *TraverseAndMark* Function.

Theorem *1* proves the correctness of the algorithm, but the time complexity of Step *4* is not obvious, yet. There could be *O(V)* vertices removed and each of them may have *O(V)* unmarked vertices in its subtree, which would make the time complexity $O(V^2+H)$. This is where we use of the Theorem *2*:

Theorem 2. *If some vertex v of the tree is marked, then all the vertices in v's subtree are marked, too.*

Proof. If *v* is the *LCA* of some pair which is removed in order to disconnect the vertices of that pair, then we will mark *v* and all the unmarked vertices in its subtree. Therefore, all the vertices in *v*'s subtree will be marked. If *v* is not one of the removed vertices, then *v* was marked because of the removal of some ancestor *k* of *v*. When vertex *k* was removed, all the vertices in vertex *k*'s subtree were marked. Since all the vertices in vertex *v*'s subtree also belong to vertex *k*'s subtree, they were marked, too, and this concludes the proof.

Using Theorem *2*, we can use the recursive algorithm **TraverseAndMark**, presented in Pseudocode 7-7, for marking the unmarked vertices in the subtree of a vertex *v*. In this algorithm we

denote by $sons_v$ the set of sons of the vertex v.

TraverseAndMark marks only the unmarked vertices in the subtree of the vertex v given as an argument. Since no vertex is marked twice, *TraverseAndMark* is called at most V times during Step *4* of the algorithm. During each call, all the sons of the vertex v given as an argument are considered. Overall, all the calls do not take more time than calling *TraverseAndMark* for the root of the tree, which takes $O(V)$ time. Therefore, Step *4* of the algorithm has time complexity $O(V+H)$. The pseudocode of the whole algorithm is given in Pseudocode 7-8.

```
UVMC(tree with V vertices, H pairs (s1,t1), ..., (sH,tH)):
Step 1. Choose a root r and compute the parent-son relationships for all the vertices of the tree.
Step 2. Compute the arrays pLCA and level: pLCA[i] is the lowest common ancestor of the ith pair and level[i] is the level of their LCA in the tree.
Step 3.
for l=0 to V-1 do LL[l]=empty
for i=1 to H do LL[level[i]].add(i)
sorted_pairs=empty
for lev=V-1 down to 0 do {
  if (LL[lev] is not empty) then {
    for i in LL[lev] do sorted_pairs.add(i)
  }
}
Step 4.
for vn=1 to V do marked[vn]=False
num_removed=0
for i=1 to H do {
  p=sorted_pair[i]
  if ((not marked[sp]) and (not marked[tp])) then {
    num_removed = num_removed+1
    TraverseAndMark(pLCA[p])
  }
}
return num_removed
```

Pseudocode 7-8. The Linear Algorithm for the Unrestricted Vertex Multicut Problem in Trees.

7.7.4. A New Reliability Metric for Trees

We define the reliability of a tree as the number of vertices whose removal disconnects a carefully chosen set of H critical pairs, divided by the total number of vertices belonging to at least one pair. The way critical pairs are defined depends on the purpose of the tree network. They could be pairs of vertices between which the highest amounts of traffic are recorded or pairs of vertices which, if disconnected, would highly compromise the performance of the network. We will choose the situation in which tree networks are used for balanced content replication and define the critical pairs according to the specific communication patterns of this situation. We present evaluation results of the reliability metric in this case.

We considered two types of test scenarios. For the first type, we chose different values for the following parameters: the number of vertices of the tree, the maximum degree, the number of leaf vertices and the number of colors (pieces of content). Then, using the algorithm presented earlier, we equitably colored the tree (we chose a balanced distribution of content replicas). We then chose an extra parameter C, representing the number of vertices used for serving client requests; the actual vertices were then chosen randomly. The critical pairs were the pairs (i,j) where i is a vertex serving client requests and j closer to i than other vertices q colored with the same color as vertex j. The results are shown in Table 7-2.

It is clear that the reliability decreases with the number of colors (for fixed V and D) and with the maximum degree (for fixed V and k), with a few minor exceptions. We also studied reliability variations for fixed V, variable leaf percentage and variable number of colors. Fig. 7-1 shows that reliability decreases as the leaf percentage increases. Furthermore, we tried to understand how the reliability would change with the number of vertices and different leaf percentages. Fig. 7-2 shows that leaf percentage matters much more than the number of vertices.

In the second type of test scenarios, we considered critical pairs of the form (i,j), where i and j are two vertices with the same color. The motivation behind this was that different replicas need to be synchronized occasionally, so communication between vertices hosting the same replica is needed. The obtained results are similar to the ones for the first type of test scenarios. The experiments showed that tree structure, rather than other parameters, is the most important in terms of reliability, which was to be expected.

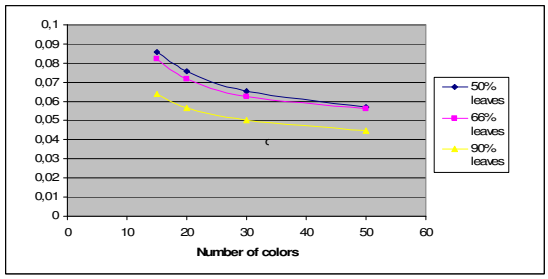


Fig. 7-1. Variation of Reliability Values for V=10000 Vertices.

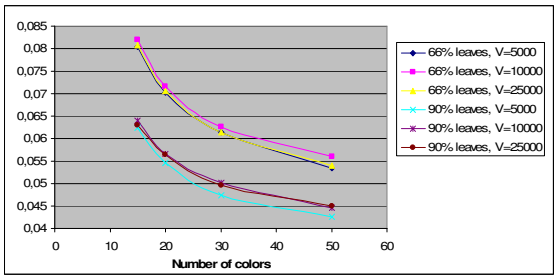


Fig. 7-2. Reliability Values for Different Numbers of Vertices and Leaf Percentages.

Table 7-2. Results for the First Type of Test Scenarios.

V	D	k	C	H	removed vertices	vertices in pairs	reliability	number of leaves
10000	2	2	1000	2000	987	2705	0,3649	2
10000	2	3	1000	2000	987	2705	0,3649	2
10000	2	5	1000	4000	987	4101	0,2407	2
10000	2	10	1000	9792	987	6550	0,1507	2
10000	3	3	1000	2774	954	3215	0,2967	50%
10000	3	4	1000	3787	954	3835	0,2488	50%
10000	3	5	1000	4987	954	4488	0,2126	50%
10000	3	10	1000	10587	954	6599	0,1446	50%
10000	5	5	1000	5545	935	4525	0,2066	50%
10000	5	6	1000	6884	935	5054	0,1850	50%
10000	5	10	1000	11885	935	6412	0,1458	50%
10000	15	15	500	9557	485	5642	0,0860	50%
10000	15	20	500	12981	485	6410	0,0757	50%
10000	15	30	500	19446	485	7412	0,0654	50%
10000	15	50	500	31735	485	8511	0,0570	50%

7.8. A Dynamic Programming Framework for Optimization Problems on Graphs with Bounded Pathwidth and Treewidth

The pathwidth of an undirected graph is a number which reflects the resemblance of the graph's structure to a path – the lower the pathwidth, the closer the graph "looks" like a path. A path decomposition of a graph G is a path D, with nodes D_1, D_2, ..., D_P (in the order they lie on the

path), having the following properties:

- each node D_i corresponds to a subset of $nv(i) \geq 0$ vertices of G (we will denote the subset by D_i, too)
- any two adjacent vertices of the graph G, u and v, belong together to at least one subset D_i
- each vertex u of G belongs to at least one subset D_i and if u belongs to two subsets D_i and D_k, then it also belongs to all the subsets in between D_i and D_k (the subsets which contain a vertex u form a sub-path of D)

The width of the path decomposition is defined as $pw_D=max\{nv(1), \ldots, nv(P)\}-1$. The minimum value of pw_D of a path decomposition of the graph is called the graph's pathwidth. Finding a path decomposition with minimum width is an NP-hard problem, but in many practical situations, a decomposition whose width is bounded by a constant can be easily found. Moreover, some efficient algorithms for finding path decompositions of small width have been developed [Bodlaender, 1996].

The pathwidth concept is strongly related to the notion of treewidth, which was introduced by Robertson and Seymour [Robertson and Seymour, 1986]. The treewidth captures the degree of similarity of a graph's structure to a tree. Many NP-hard problems can be solved in polynomial time on graphs whose pathwidth (or treewidth) are bounded by a constant. These algorithms are usually based on the dynamic programming technique and have a time complexity of the form $O(f(pw)\cdot n)$ $(O(f(tw)\cdot n))$, where $f(pw)$ $(f(tw))$ is a function which is exponential in the width of the path decomposition pw (width of the tree decomposition tw), and n is the number of vertices of the graph.

We will discuss next the case of graphs of bounded pathwidth and we will present a generic algorithmic framework for such graphs. Then, in section 7.9.6 we will show how the framework can be extended to graphs with bounded treewidth. In order to simplify the algorithms, we will use the concept of nice path decompositions. The nodes (subsets) of a nice path decomposition are of the following two types:

- *Introduce node*: If D_i is an introduce node, then $D_i = D_{i-1} \cup \{x\}$, where x is a vertex which does not belong to D_{i-1} (the introduced vertex). D_1 is an *introduce node* consisting of just one vertex.
- *Forget node*: If D_i is a forget node, then $D_i = D_{i-1} \setminus \{x\}$, where x is a vertex which belongs to D_{i-1}, but not to D_i (the forgotten vertex). D_P is a forget node with $nv(P)=0$.

Any path decomposition can be easily transformed into a nice path decomposition with $O(n)$ nodes in $O(n)$ time [Bodlaender and Kloks, 1996]. All the algorithms in the subsequent section will consider that a nice path decomposition is already known.

Dynamic programming algorithms traverse the nodes of the given nice path decomposition in order and for each node i they compute a table T_i. The size of the table T_i is exponential in the number of vertices of the subset D_i. Each entry of the table contains a state S and a value v, i.e. $T_i[S]=v$. S is the state of the vertices in D_i and is usually composed of one or several values for each vertex in D_i. v is the value of the optimization function, restricted to the vertices in

$$UD_i = \bigcup_{j=1}^{i} D_j \qquad \textbf{(7-7)}$$

and considering that the vertices in D_i are in the state S. $T_i[S]$ is computed based on the values $T_{i-1}[S']$, for some states S' which are *compatible* with the state S. The definition of state compatibility depends on the actual problem solved (just like the definition of the state itself). Each state obeys several structural rules, which depend on the problem. We will call the states which violate some of these rules *intermediate* states. These states will need to be *normalized* into valid states. The set of all valid states of a node D_i is called VS_i.

Within the proposed generic algorithm, we will iterate through all the states for node D_{i-1} and expand these states into valid states for the node D_i. The expansion function will depend on the actions that we can perform (which are problem dependent) and on the node D_i's type (*Introduce* or

Forget node). In the end, the solution will be found in one of the entries of the table T_P, considering only states belonging to a subset of *valid final states*. We are only interested in finding the value of an optimization function, not the states of the graph vertices leading to the optimal value. However, these states can easily be computed from the tables stored for each node of the path decomposition (by going back from node *P* to node *1*). The generic dynamic programming algorithm is given in Pseudocode 7-9.

From an implementation point of view, the states for each node will be generated in an array (or hash table) of states, which can be traversed easily. When reading or writing a value $T_i[S]$, we need to know the index of state *S* in the array of states (between *1* and the total number of states). The most efficient way to do this is to use two hash functions ($hash_1$ and $hash_2$). $hash_1$ will generate a unique hash value for each state *S* (no collisions are allowed). This value will be stored in a hash table, together with the state index. The hash table will use the $hash_2$ function and permits some collisions. Pseudocode 7-10 illustrates the use of this approach.

```
Generic Dynamic Programming Algorithm:
compute T1[S], for all states S in VS1
for i=1 to P-1 do {
  for all states S in VSi+1 do Ti+1[S]=uninitialized
  for S in VSi, such that Ti[S]≠uninitialized do {
    for action in setOfActions(i+1) do {
      // S' is an intermediate state (not necessarily valid)
      // C is the (new) value of the optimization function
      (S',C,ok)=expandState(S, i, action)
      if (ok=true) then {
        S''=normalize(S')
        if ((better(C, Ti+1[S'']) or (Ti+1[S'']=unintialized)) then Ti+1[S'']=C
        }
      }
    }
  }
OPT=uninitialized
for S in setOfValidFinalStates() do
  if (better(TP[S], OPT)) or (OPT=uninitialized) then OPT=TP[S]
return OPT
```

Pseudocode 7-9. The Generic Dynamic Programming Algorithm for Graphs with Bounded Pathwidth.

```
generateStates(i): // generates all the states for node Di
stateIndex=0
for each state S generated do {
  h1=hash1(S)
  stateIndex=stateIndex+1
  hashTable[i].put(S, stateIndex) // hashTable[i] uses hash2()
}

getStateIndex(S, i):
return hashTable[i].get(hash1(S))
```

Pseudocode 7-10. State Management Functions.

Since we are discussing efficiency, we should note that the sets of states of two nodes D_i and D_j will differ only if $nv(i) \neq nv(j)$. This suggests that we could generate the states only for each distinct value of the number of vertices (there are only pw_D+1 such values) and not for each node.

We will show next how we can solve several graph optimization problems (including

optimal replica placement problems) using the generic algorithmic framework that we just introduced.

7.8.1. Coloring a Graph with a Fixed Number of Colors

We are given a graph G together with a nice path decomposition of the graph. We have to assign to each vertex of the graph a color from the set $\{1,2,\ldots,C\}$, such that any two vertices connected by an edge are assigned different colors.

This is one of the simplest problems, in which the function which needs to be computed is a binary function. We need to decide if a coloring exists or not. If it exists, the vertex colors can be derived from the tables stored at each node of the path decomposition. Furthermore, we can use the solution to this problem in a binary (or linear) search algorithm, in order to find the minimum number of colors required to color the graph.

The state of the vertices of a node D_i of the path decomposition has the form $S=(c_1, c_2, \ldots, c_{nv(i)})$, where $c_i \in \{1,\ldots,C\}$ is the color of the i^{th} vertex in the subset D_i. We will, occasionally, denote by $S[i]$ the i^{th} component of the state S. We will consider the vertices of a node D_i ordered as $v_{i,1}, v_{i,2}, \ldots, v_{i,nv(i)}$. If D_i is an *Introduce* node, then we will consider that the introduced vertex is $v_{i,nv(i)}$. We will maintain these vertex ordering assumptions in all the other problems considered in this section. An entry $T_i[S]$ has one of the values *true* or *uninitialized*, meaning that there exists (does not exist) a coloring of the vertices in UD_i, such that the vertices in D_i are colored according to the state S.

```
setOfActions(i):
if (D_i is an Introduce node) then
  return the set composed of (Col, 1), (Col, 2), …, (Col, C)
else return the set containing the only element „Forget”

updateCost(S, i, C): // auxiliary function, used by expandState
let S=(c_1, c_2, …, c_nv(i))
for j=1 to nv(i)-1 do
  if ((v_i,j and v_i,nv(i) are adjacent) and (c_j=c_nv(i))) then return ( (), 0, false)
return (S, 1, true)

expandState(S, i, action):
if (D_i is an Introduce node) then {
  (Col, cx)=action // cx is the color assigned to the new vertex
  S’=(c_1, c_2, …, c_nv(i)-1, c_nv(i)=cx), where S=(c_1, c_2, …, c_nv(i)-1)
 return updateCost(S’, i, T_i-1[S])
} else {
  v_i-1,j = the “forgotten” node
  S’=(c_1, c_2, …, c_j-1, c_j+1, …, c_nv(i-1)), where S=(c_1, c_2,…, c_nv(i-1))
  return (S’,1,true)
}

normalize(S):
return S

better(cost_1, cost_2):
if (cost_1=1) then return true
else return false

setOfValidFinalStates():
return VS_P // all the states of D_P are valid final states
```

Pseudocode 7-11. Graph Coloring Functions.

The node D_1 contains only a single vertex, so we will assign $T_1[S]=true$, for all the states S

in VS_1. The set of actions which can be performed for expanding a state S of the node i into a state S' of the node $i+1$ depends on the type of the node D_{i+1}. If D_{i+1} is an *Introduce* node, the set of actions consists of coloring the introduced node in every possible color; if it is a *Forget* node, only a *"forget"* action exists. We will now define all the functions required to turn the generic algorithm from the previous section into a solution to the problem.

It is obvious that the *expandState* function is the most important one in the algorithm and this will be the case with each problem we will consider. In this function, the selected action is performed and the validity of the resulting intermediate state is checked. The complexity of the algorithm is $O((pw+1)\cdot C^{pw+1}\cdot P)$, considering that the path decomposition has width pw. Since P is $O(n)$ and $(pw+1)\cdot C^{pw+1}$ is bounded by a constant, the time complexity of the algorithm is linear.

7.8.2. Coloring a Graph with a Fixed Number of Colors – Improved State Definition

The improvement of the previous solution consists in reducing the number of states. It is obvious that, given a valid coloring of the graph's vertices, we can relabel the colors differently and still get a valid coloring. For instance, if $C=3$ and we have two vertices a and b colored with colors 3 and 2, respectively, we can relabel the colors such that vertex a is colored with 1 and vertex b is colored with 2. This suggests that the colors of a state S should form a partition and obey the following rules:

- $c_1=1$
- $c_i \in \{1,...,m_{i-1}+1\}$, where $m_{i-1} = \max_{1\le j\le i-1}\{c_j\}$

With these rules, the state S' returned by the *expandState* function may not be a valid state. Therefore, we will have to define the *normalize* function differently:

```
normalize(S):
S'=S, where S=(c1, ..., cK)
counter=0
newlabel={0,0,...,0} // K zeroes
for i=1 to K do {
  if (newlabel[S[i]]=0) then {
    counter=counter+1
    newlabel[S[i]]=counter
  S'[i]=newlabel[S[i]]
  }
}
return S'
```

Pseudocode 7-12. The *normalize* Function.

The *normalize* function relabels the colors of a state S such that they obey the structural rule. The number of states is greatly reduced. For instance, for $C=7$ and a node D_i with $nv(i)=9$, the number of states is equal to the number of partitions of a set with 9 elements into at most 7 parts, which is 21.110. Before, the number of states was $9^7 = 4.782.969$.

7.8.3. Coloring a Graph with a Fixed Number of Colors in Order to Minimize Penalties due to Coloring Conflicts

This problem is similar to the previous one, except that a valid coloring is not necessarily required. Each graph edge (u,v) has an associated penalty value $pen(u,v)$. If the vertices u and v are assigned the same color, then the penalty $pen(u,v)$ will be paid. The optimization function consists of minimizing the sum of paid penalties. For this problem, we will keep the same state definition as in the previous case, the same sets of actions and the same valid final states. We will have to slightly modify the *expandState* function, by redefining the auxiliary function *updateCost*, and the

better function. $T_i[S]$ now represents the minimum penalty paid such that all the vertices in UD_i are colored and the vertices in D_i are colored according to the state S. T_1 will be initialized with 0 for every possible state.

No other changes are necessary in order to solve this problem, which has applications to frequency assignment in wireless networks. If we want to solve a slightly different version of the problem, in which we try to minimize the maximum penalty paid instead of the sum of penalties, we only have to change the additive operator in the *updateCost* function with the *max* operator ($C'=max\{C', pen(v_{i,j}, v_{i,nv(i)})\}$). A different solution to this modified problem consists of binary searching the cost to be paid. When the cost C is fixed, we can ignore all the edges with a penalty lower than (or equal to) C and we would now have to solve a normal coloring problem (as a feasibility test).

```
updateCost(S, i, C):
C'=C
for j=1 to nv(i)-1 do {
  if ((adjacent(v_{i,j},v_{i,nv(i)})) and (S[j]=S[nv(i)])) then C'=C'+pen(v_{i,j}, v_{i,nv(i)})
}
return (S,C',true)

better(cost_1, cost_2):
if (cost_1<cost_2) then return true
else return false
```

Pseudocode 7-13. Functions for Graph Coloring with Penalties.

7.8.4. (L,U)-Replica Placement

We are given an undirected graph with n vertices together with a nice path decomposition with small pathwidth pw. Every vertex i has a weight $wk(i)$. We want to select a subset of distinct vertices of the graph and place a replica of some popular content in them, such that the sum of their wk values is at least L at most U. The cost of selecting a vertex i is $csel(i)$. If two vertices u and v which are adjacent to one another are selected, then we will also need to pay a penalty cost $pen(u,v)$. We are interested in paying the minimum total cost for placing the replicas. The state definition we will use is the following: for a node D_i, a state S has the form $(s_1, \ldots, s_{nv(i)}, x)$, where:

- $s_j=1$ if $v_{i,j}$ was selected for placing a replica
- $s_j=0$ if $v_{i,j}$ was not selected for placing a replica
- x is the total sum of the wk values of vertices selected (so far)

The set of actions of an *Introduce* node consists of two actions *{ Select, Do Not Select }* and that of a *Forget* node will be the same as before (*{Forget}*). The main functions required by the framework are shown in Pseudocode 7-14. The *normalize* function will not be presented (because all the intermediate states will be valid) and the valid final states will be only those with $L \leq x \leq U$.

We will use the same *better* function as in the minimum penalty coloring problem. The time complexity of the algorithm is $O(U \cdot 2^{pw} \cdot n)$. We can introduce several variations to this problem, like defining penalty or profit values for each pair of adjacent vertices (u,v), where u is a selected vertex and v is not. These changes would require a different *updateCost* function.

7.8.5. Covering a Partial Grid Graph with Rectangular Sub-Grids

A (m,n) grid graph has $m \times n$ vertices arranged on m rows and n columns. Each vertex is adjacent to at most four other vertices (on the rows above and below and the columns to the left and to the right). Such graphs appear, for instance, in processor interconnection networks. A partial (m,n) grid graph is a (m,n) grid graph in which some of the vertices and some of the edges may be missing.

```
updateCost(S, i, C):
C'=C
for j=1 to nv(i)-1 do {
  if ((adjacent(v_{i,j}, v_{i,nv(i)}) and (S[j]=S[nv(i)]=1)) then C'=C'+pen(v_{i,j}, v_{i,nv(i)})
}
return (S,C',true)

expandState(S, i, action):
let S=(s_1, s_2, …, s_{nv(i)-1}, x)
if (D_i is an Introduce node) then {
  if (action=Select) then {
    if (x=k) then return ((), +∞, false)
    S'=(s_1, s_2, …, s_{nv(i)-1}, 1, x+wk(v_{i,nv(i)}))
    return updateCost(S', i, T_{i-1}[S]+csel(v_{i,nv(i)}))
  } else { // action = Do Not Select
    S'=(s_1, s_2, …, s_{nv(i)-1}, 0, x)
    return (S', T_{i-1}[S], true)
  }
} else {
  v_{i-1,j} = the "forgotten" node
  S'=(s_1, s_2, …, s_{j-1}, s_{j+1}, …, s_{nv(i-1)}, x)
  return (S',T_{i-1}[S],true)
}
```

Pseudocode 7-14. Replica Placement Functions.

These graphs have their pathwidth bounded by $min\{m,n\}$. Let's assume that one of the dimensions is bounded by a constant (without loss of generality, we will assume this dimension is n). A path decomposition with pathwidth n (i.e. in which each set contains at most $n+1$ vertices) can be easily obtained by ordering the vertices from the first to the last row and, for each row, from the first to the last column and introducing the vertices in this order (and forgetting the *„oldest"* vertex after every vertex set containing $n+1$ vertices).

Moreover, the vertices within a node D_i are ordered using the same criterion. Partial grid graphs are planar graphs and because of this, the number of states in the dynamic programming algorithm can be reduced by making use of the Catalan property (mentioned in [Dorn, Fomin and Thilikos, 2007]). Let's consider the minimum path (cycle) cover problem. If a state S of a node D_i contains both endpoints of two different paths, then let's assume that $v_{i,a}$ and $v_{i,b}$ $(a<b)$ are the endpoints of the first path and $v_{i,c}$ and $v_{i,d}$ $(c<d)$ are the endpoints of the second path. One of the following conditions must hold: $(b<c)$, $(d<a)$, $((a<c)$ and $(d<b))$, $((c<a)$ and $(b<d))$. The states S which do not obey any of these conditions can be removed.

We are given a $m \times n$ *partial grid graph* (with n bounded by a constant C) and a set of K types of rectangular *pieces* $\{ (r_1,c_1), (r_2,c_2), \ldots, (r_K,c_K) \}$. The i^{th} type covers r_i rows and c_i columns of the graph (i.e. covers $r_i \times c_i$ vertices). c_i is bounded by n and r_i is bounded by a constant R. Each piece i also has a weight w_i. We want to place on the graph as many rectangular pieces of each type, such that the sum of the weights of each piece we placed is maximum, and under the following restrictions:

- no two rectangular pieces should overlap
- a rectangular piece cannot cover missing vertices
- a rectangular piece must be fully included in the graph
- we can use as many pieces of any type as we want

- we cannot rotate the pieces (although it's not excluded that both (r_i,c_i) and (c_i,r_i) belong to the set)

This problem is more difficult than the previous ones, because it is formulated taking into consideration the information that the graph is a partial grid graph (and not just a graph with bounded pathwidth). The dynamic programming states will have the following form: $(h_1, h_2, \ldots, h_{nv(i)})$, where h_j is the number of vertices above (and including) the vertex $v_{i,j}$ (i.e. on the same column) which are not covered and not missing from the graph. We are only interested in values $0 \leq h_j \leq R$ (if there are more than R vertices not covered and not missing above $v_{i,j}$, we will limit this value to R).

Then, when a new vertex is introduced, the set of actions consists of either not doing anything or choosing to place one of the K types of rectangular pieces with its lower right corner at vertex $v_{i,j}$ (if possible – i.e. for a piece (r_p, c_p) and the vertex $v_{i,j}$ located on row a and column b, we must have: $b \geq c_p$, none of the vertices on the row a and the columns $b, b-1, \ldots, b-c_p+1$ are missing from row a and each of them has at least r_p non-covered and non-missing vertices above them).

There is an extra problem here, given by the fact that we need to know the value of h_j (for the vertex $v_{i,j}$) at the moment the vertex is introduced. For this, we will consider the pathwidth to be $n+1$. This way, when $v_{i,j}$ is introduced, the vertex right above $v_{i,j}$ is located in D_i and we can compute h_j as 1 plus the corresponding value for the vertex above $v_{i,j}$ (or just 1 if that vertex is missing from the partial grid graph). When placing a rectangular piece over the columns $b-c_p+1, \ldots, b$ of the graph, the h_j values of the corresponsing vertices of the obtained state will be set to 0. The framework functions are similar to those presented for the other problems.

This problem is motivated by replica placement placement problems on connected subsets of vertices of a graph.

7.8.6. Extending the Framework to Graphs with Bounded Treewidth

The framework can be extended to graphs with bounded treewidth quite easily. First, every tree decomposition of a graph can be transformed into a nice tree decomposition which consists of three types of nodes:

- *Introduce node*: A node Q_i is an introduce node if it has only one son Q_j and $Q_i = Q_j \cup \{x\}$, where x is a vertex which does not belong to Q_j (the introduced vertex).
- *Forget node*: A node Q_i is a forget node if it has only one son Q_j and $Q_i = Q_j \setminus \{x\}$, where x is a vertex which belongs to Q_j, but not to Q_i (the forgotten vertex).
- *Join node*: A node Q_i is a join node if it has exactly two sons Q_j and Q_k, such that $Q_i=Q_j=Q_k$ (i.e. they represent the same subset of graph vertices)

With a nice tree decomposition, we will compute, for each node of the decomposition (in a bottom-up manner), the same set of states we would compute in the case of a path decomposition. When we compute the table of states for an *Introduce* or *Forget* node, we use the same rules as if the current node were a node D_i and its son were the previous from the path decomposition D_j. Thus, we only need to show how to handle the case when the node is a *Join* node.

Let's assume that we are at a *Join* node Q_i and its sons are Q_j and Q_k. We have a table of states and values, $T(Q_j)$ and $T(Q_k)$, for each of the two sons. For a state S, $T(Q_i, S)$ is computed as a combination of $T(Q_j,S)$ and $T(Q_k,S)$. Combining the two values depends on the actual problem being solved. If $better(T(Q_j,S), T(Q_k,S))=true$ then $T(Q_i,S)=T(Q_j,S)$ else $T(Q_i,S)=T(Q_k,S)$.

Chapter 8 – Conclusions

8.1. Overview of the Results

This book has addressed multiple key problems from the communication optimization domain and has proposed several novel solutions, ranging from centralized and decentralized system architectures to online and offline optimization algorithms and generic algorithmic frameworks. The book began by identifying and classifying the main communication parameters and requirements of the most important types of distributed systems and then continued by performing a thorough analysis of the current state-of-the-art scientific knowledge in the domains addressed by the book. This analysis created the context in which the results presented in this book have to be placed, by identifying both the advantages and the disadvantages of the existing solutions and by pointing out the unsolved problems which were addressed in the book.

In Chapter 2 we argued for the need of application-layer routing architectures by presenting 5 motivating scenarios. Then, in Chapter 3, we introduced a generic peer-to-peer architectural model, together with new methods for neighbouring peer selection. The model was implemented in two distinct peer-to-peer architectures (a fully decentralized one and a hybrid one), which were evaluated in real-life conditions. The evaluation results showed that data transfer throughput can be significantly increased by using peer-to-peer overlays specifically designed for this purpose.

The concepts of the peer-to-peer architectural model were also used in a fully decentralized fault-tolerant peer-to-peer architecture for storing data objects which allows multidimensional range search queries. The peers are organized in a much more natural way which allows us to overcome some of the drawbacks of the previous solutions in this area. Each objects has an owner and is replicated at the neighbors of this owner (up to a certain number of hops). Multidimensional range searches visit only as many peers as they are necessary. The number of such peers is either proportional to the volume of the multidimensional interval or to the number of objects located in the interval (whichever is higher).

In Chapter 3 we also presented novel techniques for enhancing system-level fairness and user-perceived quality of service in peer-to-peer content sharing systems with arbitrary topologies. The techniques addressed the communication layer (by using path reservations and by allocating incoming bandwidth to sockets proportionally to the sum of the priorities of the reservations sharing the socket) of the system. These techniques were evaluated in real situations: we showed that bandwidth can be allocated proportionally, according to the stated goals.

Also in Chapter 3 we introduced a new method for upload capacity estimation, based on a collaborative scenario using helper peers. Although many techniques for estimating the end-to-end capacity (or even available bandwidth) of a path have been developed, there were no results on upload capacity estimations before ours. Information regarding upload capacities is useful in many peer-to-peer file sharing systems for selecting super-peers or for trading bandwidth with other peers. The upload capacity estimation method further led to the development of a novel *less-than-best-effort* congestion control algorithm.

The development of large scale distributed systems is always a tedious task, which is made even harder by the difficulty of automatically deploying and testing the system in real conditions. In order to address this problem, we developed ServMark, a testing infrastructure for distributed systems. ServMark was developed by integrating DiPerF and GrenchMark, two previous complementary testing tools. ServMark was evaluated by testing 6 web servers, but, as we showed in Chapter 3, it can easily be used for testing and deploying any distributed system (e.g. peer-to-peer systems).

The family of fully decentralized peer-to-peer architectures and techniques presented in Chapter 3 is complemented by the development of a centralized framework for data transfer

scheduling in networks on which we can exhibit full control, presented in Chapter 4. The architecture of the framework is an extension followed by a re-design of an architecture having the same purpose presented in [Cîrstoiu, 2008].

Our focus was on the scheduling component and, thus, in Chapter 3 we presented multiple novel algorithms and data structures for this purpose. We first identified the types of data transfer requests that were of interest and then we analyzed networks with simple topologies (a single network link, a network path, or networks with tree topologies). We considered two types of scheduling modes: online (one request at a time) and batch (multiple requests at a time).

For the online scheduling mode we developed new data structures (e.g. Time Slot Groups) which can handle our types of data transfer requests better than any of the existing data structures in the literature, we defined new, generic algorithmic frameworks for the segment tree data structure and block partitioning techniques which are focused on range updates (including for the multidimensional case) and we also showed how to use some of the standard data structures for some particular cases.

For the batch scheduling case we identified the problem of scheduling the requests (over time) in the presence of a mutual exclusion (hyper-)graph. We defined new algorithms for several mutual exclusion scheduling problems – some of them consider entirely new situations, while others complement some of the approaches existing in the literature.

Chapter 4 also presents a generic technique for the real-time centralized scheduling of deadline-constrained data transfers in networks with arbitrary topologies, based on the use of a time-expanded graph and maximum flow algorithms. The technique is evaluated and compared against three decentralized methods and is shown to obtain significantly better results than any of them.

Chapter 5 complements the online techniques presented in Chapter 4 by focusing on the offline scheduling of point-to-point communication flows. We consider several such scheduling problems (e.g. files with divisible sizes, two communication flows which need to be transferred concurrently on multiple disjoint paths, an optimization problem for the TCP sender buffer management) and we present new algorithmic solutions (many of them having a polynomial time complexity). One of the problems is equivalent to a problem previously studied in the literature, for which an algorithm with a slightly better time complexity than ours already exists. However, we argue that the algorithm we presented is simpler to understand and implement.

Multicast is an important communication method required in many distributed systems (e.g. either for signalling, or for live streaming). In Chapter 6 we presented a peer-to-peer architecture organized as a multicast tree with small diameter and bounded node degrees. The architecture is non-hierarchical and, under low rates of peer arrivals and departures, the tree can be maintained balanced. Simulation results have shown that even under larger peer arrival rates, the diameter of the tree does not increase much more than in the ideal case.

The rest of Chapter 6 is dedicated to offline optimal multicast strategies. We extended the single port broadcast model in trees by introducing sending and receiving constraints and we developed a family of new algorithmic techniques for this case. We also developed new algorithms for the maximum reliability k-hop multicast strategy in tree networks. A previous algorithm for an equivalent problem already exists, but our solution is better by an $O(log(n))$ factor than the previously existing one.

In Chapter 7 we considered the problem of optimally placing replicas in tree-like networks in order to minimize metrics like the maximum latency to the closest replica or the sum of latencies to the closest replica (as well as several others, based on costs). We considered networks with path, tree and cactus architectures. We solved several center and median problems on these graphs, improving or matching the time complexities of previous existing algorithms. Examples of such cases are the 1-center and longest path problems in cacti, in which we also considered weights at the graph vertices (which are added to the weights of the edges on a path), or some restricted cases of the connected k-center and k-median problems in trees.

In the end of Chapter 7 we presented an algorithmic framework for solving combinatorial

optimization problems in graphs with bounded pathwidth and treewidth. The framework is then used for solving an optimal replica placement problem in these graphs.

8.2. Summary of the Contributions of this Book

The main contributions of this book can be summarized as follows:

- The identification and classification of the main communication parameters and requirements of the most important classes of distributed systems
- A critical analysis of the state-of-the-art scientific knowledge on the topics addressed by this book (peer-to-peer architectures, fairness enforcing techniques, bandwidth estimation techniques, congestion control algorithms, real-time centralized scheduling of data transfers, offline scheduling of point-to-point communication flows, offline multicast strategies, offline replica placement strategies in tree-like networks)
- The definition and development of a generic peer-to-peer architecture, which can be extended for multiple purposes (e.g. data transfer optimization or data storage and retrieval)
- The implementation of the architectural model in two distinct peer-to-peer architectures (a fully decentralized one and a hybrid one) and the evaluation of these architecture in real settings and through simulations
- The development and evaluation of new techniques for ensuring system-level fairness and user-perceived system quality at the communication layer
- The design, implementation and evaluation of a fault-tolerant peer-to-peer architecture for storing data objects which allows efficient multidimensional range queries (based on the generic peer-to-peer architecture defined earlier)
- The design and evaluation of a new method for estimating the upload capacity of a machine
- The development of a new *less-than-best-effort* congestion control algorithm
- The development and evaluation of a new distributed testing architecture for Grid and web services, as well as any other distributed system (including peer-to-peer systems)
- The definition and development of a centralized architecture for real-time data transfer scheduling
- The design, implementation and evaluation of novel, efficient algorithms and data structures with guaranteed theoretical performances for online data transfer scheduling in networks with particular topologies (single network link, one network path, network with tree topology)
- The development of new centralized techniques for real-time scheduling of data transfers in networks with arbitrary topology and their evaluation in comparison with distributed scheduling techniques
- The development of new algorithms for data transfer scheduling with mutual exclusion graphs, with guaranteed theoretical performances
- The design of new algorithms with guaranteed theoretical performances for the optimal offline scheduling of point-to-point communication flows
- The design, implementation and evaluation of a peer-to-peer architecture with the structure of a multicast tree with small diameter and bounded node degrees
- Novel extensions to the single port broadcast model in trees (i.e. the addition of sending and receiving constraints) and a new family of polynomial time algorithms addressing these extensions
- The development of new algorithms, with a better time complexity than the previously existing ones, for the maximum reliability k-hop multicast strategy in tree networks
- The design of new offline algorithms with guaranteed theoretical performances for the optimal placement of data items in networks with a tree-like topology (e.g. paths, trees, cacti, graphs with bounded pathwidth and treewidth)

List of Publications

Papers in Conference Proceedings indexed in the ISI Database

1. Mugurel Ionuţ Andreica, Iosif Charles Legrand, Nicolae Ţăpuş, **"Towards a Communication Framework based on Balanced Message Flow Distribution"**, Proceedings of the IEEE International Conference on "Computer as a Tool" (EUROCON), pp. 2354-2359, Warsaw, Poland, 9-12 September, 2007. (ISBN: 978-1-4244-0813-X)
2. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"High Multiplicity Scheduling of File Transfers with Divisible Sizes on Multiple Classes of Paths"**, Proceedings of the 12th IEEE International Symposium on Consumer Electronics (ISCE), pp. 516-519, Vilamoura, Portugal, 14-16 April, 2008. (ISBN: 978-1-4244-2422-1)
3. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Maximum Reliability K-Hop Multicast Strategy in Tree Networks"**, Proceedings of the 12th IEEE International Symposium on Consumer Electronics (ISCE), pp. 169-172, Vilamoura, Portugal, 14-16 April, 2008. (ISBN: 978-1-4244-2422-1)
4. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Reliability Analysis of Tree Networks Applied to Balanced Content Replication"**, Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics (THETA 16) (AQTR), pp. 79-84, Cluj-Napoca, Romania, 22-25 May, 2008. (ISBN: 978-1-4244-2576-1)
5. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Cristina Teodora Andreica, Romulus Andreica, Mihai Aristotel Ungureanu, "**Optimal Geometric Partitions, Covers and K-Centers"**, Proceedings of the 9th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 173-178, Bucharest, Romania, 24-26 June, 2008. (ISBN: 978-960-6766-76-3)
6. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Optimal Offline TCP Sender Buffer Management Strategy"**, Proceedings of the 1st IARIA/IEEE International Conference on Communication Theory, Reliability, and Quality of Service (CTRQ), pp. 41-46, Bucharest, Romania, 29 June - 5 July, 2008. (ISBN: 978-1-4244-4225-6)
7. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling"**, Proceedings of the 7th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pp. 285-292, Krakow, Poland, 1-5 July, 2008. (ISBN: 978-0-7695-3472-5)
8. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Intelligent Strategies for Several Zero-, One- and Two-Player Games"**, Proceedings of the 4th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 253-256, Cluj-Napoca, Romania, 28-30 August, 2008. (ISBN: 978-1-4244-2673-7)
9. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Constrained Content Distribution and Communication Scheduling for Several Restricted Classes of Graphs"**, Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 129-136, Timisoara, Romania, 26-29 September, 2008. (ISBN: 978-0-7695-3523-4)
10. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Optimal Scheduling of Two Communication Flows on Multiple Disjoint Packet-Type Aware Paths"**, Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 137-144, Timisoara, Romania, 26-29 September, 2008. (ISBN: 978-0-7695-3523-4)
11. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, **"Towards a Real-Time Scheduling Framework for Data Transfers in Tree Networks"**, Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 467-474, Timisoara, Romania, 26-29 September, 2008. (ISBN: 978-0-7695-3523-4)
12. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control"**, Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 354-357, Timisoara, Romania, 26-29 September, 2008. (ISBN: 978-0-7695-3523-4)
13. Mădălina Ecaterina Andreica, Nicolae Cătăniciu, Mugurel Ionuţ Andreica, **"Econometric and Neural Network Analysis of the Labor Productivity and Average Gross Earnings Indices in the Romanian Industry"**, Proceedings of the 10th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 106-111, Prague, Czech Republic, 23-25 March, 2009. (ISBN: 978-960-474-063-5)
14. Mădălina Ecaterina Andreica, Mugurel Ionuţ Andreica, Nicolae Cătăniciu, **"Multidimensional Data Structures and Techniques for Efficient Decision Making"**, Proceedings of the 10th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 249-254, Prague, Czech Republic, 23-25 March, 2009. (ISBN: 978-960-474-063-5)

15. Eliana-Dina Tîrşa, Mugurel Ionuţ Andreica, Alexandru Costan, **"Data Replication Techniques with Applications to the MonAlisa Distributed Monitoring System"**, Proceedings of the IEEE International Conference on "Computer as a Tool" (EUROCON), pp. 339-346, Sankt-Petersburg, Russia, 18-23 May, 2009. (ISBN: 978-1-4244-3861-7)
16. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Central Placement of Storage Servers in Tree-Like Content Delivery Networks"**, Proceedings of the IEEE International Conference on "Computer as a Tool" (EUROCON), pp. 1901-1908, Sankt-Petersburg, Russia, 18-23 May, 2009. (ISBN: 978-1-4244-3861-7)
17. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Decision Optimization Techniques for Efficient Delivery of Multimedia Streams"**, Proceedings of the IEEE International Symposium on Signals, Circuits and Systems (ISSCS), vol. 2, pp. 333-336, Iasi, Romania, 8-10 July, 2009. (ISBN: 978-1-4244-3785-6)
18. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Nicolae Ţăpuş, **"A Fault-Tolerant Peer-to-Peer Object Storage Architecture with Multidimensional Range Search Capabilities and Adaptive Topology"**, Proceedings of the 5th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 221-228, Cluj-Napoca, Romania, 27-29 August, 2009. (ISBN: 978-1-4244-5007-7)
19. Mugurel Ionuţ Andreica, Vasile Deac, Stelian Tipa, **"Towards Providing Low-Risk and Economically Feasible Network Data Transfer Services"**, Proceedings of the 9th WSEAS International Conference on Multimedia, Internet & Video Technologies (MIV), Budapest, Hungary, 3-5 September, 2009. (ISBN: 978-960-474-114-4 / ISSN: 1790-5109)
20. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Efficient Upload Bandwidth Estimation and Communication Resource Allocation Techniques"**, Proceedings of the 9th WSEAS International Conference on Multimedia, Internet & Video Technologies (MIV), Budapest, Hungary, 3-5 September, 2009. (ISBN: 978-960-474-114-4)
21. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Practical Algorithmic Techniques for Several String Processing Problems"**, Proceedings of the 8th RoEduNet International Conference, pp. 136-141, Galati, Romania, 3-4 December, 2009. (ISBN: 978-606-8085-15-9)
22. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"New Algorithmic Approaches for Computing Optimal Network Paths with Several Types of QoS Constraints"**, Proceedings of the 8th RoEduNet International Conference, pp. 7-12, Galati, Romania, 3-4 December, 2009. (ISBN: 978-606-8085-15-9)
23. Mugurel Ionuţ Andreica, Andrei Grigorean, Nicolae Ţăpuş, **"Algorithms for Identifying Sequence Patterns with Several Types of Occurrence Constraints"**, Proceedings of the 11th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, 26-29 September, 2009. In Press.
24. Mugurel Ionuţ Andreica, Irina Borozan, Lucian-Ionuţ Bălăceanu, Nicolae Ţăpuş, **"Fairness and QoS Enhancement Models and Techniques for Peer-to-Peer Content Sharing Systems"**, Proceedings of the 11th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, 26-29 September, 2009. In Press.

Papers in Journals indexed in the ISI Database

25. Mugurel Ionuţ Andreica, **"Algorithmic Decision Optimization Techniques for Multiple Types of Agents with Contrasting Interests"**, Metalurgia International, vol. 14, special issue no. 11, pp. 162-170, 2009. (ISSN: 1582-2214)
26. Mugurel Ionuţ Andreica, Mădălina Ecaterina Andreica, Costel Vişan, **"Optimal Constrained Resource Allocation Strategies under Low Risk Circumstances"**, Metalurgia International, vol. 14, special issue no. 8, pp. 143-154, 2009. (ISSN: 1582-2214)
27. Mugurel Ionuţ Andreica, Sorin Briciu, Mădălina Ecaterina Andreica, **"Algorithmic Solutions to Some Transportation Optimization Problems with Applications in the Metallurgical Industry"**, Metalurgia International, vol. 14, special issue no. 5, pp. 46-53, 2009. (ISSN: 1582-2214)

Papers in International Conference & Workshop Proceedings (non-ISI)

28. Mugurel Ionuţ Andreica, **"A Dynamic Programming Framework for Combinatorial Optimization Problems on Graphs with Bounded Pathwidth"**, Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics (THETA 16) (AQTR), pp. 120-125, Cluj-Napoca, Romania, 22-25 May, 2008 (Poster Session). (ISBN: 978-973-713-248-2)
29. Mugurel Ionuţ Andreica, **"Optimal Scheduling of File Transfers with Divisible Sizes on Multiple Disjoint Paths"**, Proceedings of the 7th IEEE Romania International Conference "Communications", pp. 155-158, Bucharest, Romania, 5-7 June, 2008. (ISBN: 978-606-521-008-0)
30. Mugurel Ionuţ Andreica, Romulus Andreica, Angela Andreica, **"Minimum Dissatisfaction Personnel Scheduling"**, Proceedings of the 32nd American Romanian Academy of Arts and Sciences' International Congress, pp. 459-463, Boston, USA, 22-25 July, 2008. (ISBN: 978-2-553-01424-6)

31. Alexandra Carpen-Amarie, Mugurel Ionuţ Andreica, Valentin Cristea, **"An Algorithm for File Transfer Scheduling in Grid Environments"**, Proceedings of the 2nd International Workshop on High Performance Grid Middleware (HiPerGrid), pp. 33-40, Bucharest, Romania, 21-22 November, 2008. (ISSN: 2065-0701)
32. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Offline Algorithmic Techniques for Several Content Delivery Problems in Some Restricted Types of Distributed Systems"**, Proceedings of the 2nd International Workshop on High Performance Grid Middleware (HiPerGrid), pp. 65-72, Bucharest, Romania, 21-22 November, 2008. (ISSN: 2065-0701)
33. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Nicolae Ţăpuş, **"A Peer-to-Peer Architecture for Multi-Path Data Transfer Optimization using Local Decisions"**, Proceedings of the 3rd Workshop on Dependable Distributed Data Management (WDDDM), pp. 2-5, Nuremberg, Germany, 31 March-3 April, 2009. (ISBN: 978-1-60558-462-1)
34. Alexandru Costan, Corina Stratan, Eliana-Dina Tîrşa, Mugurel Ionuţ Andreica, Valentin Cristea, **"Towards a Grid Platform for Scientific Workflows Management"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 37-44, Bucharest, Romania, 26-29 May, 2009. (ISSN: 2066-4451)
35. Florin Pop, Ciprian Mihai Dobre, Alexandru Costan, Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Corina Stratan, Valentin Cristea, **"Critical Analysis of Middleware Architectures for Large Scale Distributed Systems"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 29-36, Bucharest, Romania, 26-29 May, 2009. (ISSN: 2066-4451)
36. Ciprian Mihai Dobre, Florin Pop, Alexandru Costan, Mugurel Ionuţ Andreica, Valentin Cristea, **"Robust Failure Detection Architecture for Large Scale Distributed Systems"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 433-440, Bucharest, Romania, 26-29 May, 2009. (ISSN: 2066-4451)
37. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Nicolae Ţăpuş, Florin Pop, Ciprian Mihai Dobre, **"Towards a Centralized Scheduling Framework for Communication Flows in Distributed Systems"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 441-448, Bucharest, Romania, 26-29 May, 2009. (ISSN: 2066-4451)
38. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Alexandru Costan, Nicolae Ţăpuş, **"Offline Algorithms for Several Network Design, Clustering and QoS Optimization Problems"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 273-280, Bucharest, Romania, 26-29 May, 2009. (ISSN: 2066-4451)
39. Andrei-Horia Mogoş, Mugurel Ionuţ Andreica, **"Approximating Mathematical Semantic Web Services Using Approximation Formulas and Numerical Methods"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 2, pp. 533-538, Bucharest, Romania, 26-29 May, 2009. (ISSN: 2066-4451)
40. Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa, Nicolae Ţăpuş, **"Data Distribution Optimization using Offline Algorithms and a Peer-to-Peer Small Diameter Tree Architecture with Bounded Node Degrees"**, Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 2, pp. 445-452, Bucharest, Romania, 26-29 May, 2009 (3rd International Workshop on High Performance Grid Middleware - HiPerGrid). (ISSN: 2066-4451)
41. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Online and Offline Algorithmic Techniques for Communication Performance Optimization in Distributed Systems"**, Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), Rome, Italy, 25-29 May, 2009 (Poster-TCPP Ph.D. Forum). (ISBN: 978-1-4244-3750-4)
42. Mădălina Ecaterina Andreica, Mugurel Ionuţ Andreica, Angela Andreica, **"Efficient Algorithms for Several Constrained Activity Scheduling Problems in the Time and Space Domains"**, Proceedings of the 33rd American Romanian Academy of Arts and Sciences' International Congress, vol. 1, pp. 59-63, Sibiu, Romania, 2-5 June, 2009. (ISBN: 978-2-553-01433-8)
43. Mugurel Ionuţ Andreica, Mădălina Ecaterina Andreica, Daniel Ardelean, **"Efficient Algorithms for Several Constrained Resource Allocation, Management and Discovery Problems"**, Proceedings of the 33rd American Romanian Academy of Arts and Sciences' International Congress, vol. 1, pp. 64-68, Sibiu, Romania, 2-5 June, 2009. (ISBN: 978-2-553-01433-8)

Papers in International Journals (non-ISI)

44. Mugurel Ionuţ Andreica, **"Efficient Algorithmic Techniques for Several Multidimensional Geometric Data Management and Analysis Problems"**, "Informatica Economica" Journal, vol. 12, no. 4 (48), pp. 17-21, 2008. (ISSN: 1453-1305)
45. Mugurel Ionuţ Andreica, **"Algorithmic Techniques for Several Optimization Problems Regarding Distributed Systems with Tree Topologies"**, ROMAI Journal, vol. 4, no. 1, pp. 1-25, 2008 (16th International Conference on Applied and Industrial Mathematics, Oradea, Romania, 9-11 October, 2008). (ISSN: 1841-5512)

46. Mugurel Ionuţ Andreica, Romulus Andreica, Angela Andreica, **"Inferring Company Structure from Limited Available Information"**, Supplement of the Journal "Quality - Access to Success", vol. 89, pp. 412-416, 2008. (ISSN: 1582-2559)
47. Mugurel Ionuţ Andreica, Cristina Teodora Andreica, Mădălina Ecaterina Andreica, **"Locating Restricted Facilities on Binary Maps"**, Supplement of the Journal "Quality - Access to Success", vol. 89, pp. 87-91, 2008. (ISSN: 1582-2559)
48. Emanuela Cerchez, Mugurel Ionuţ Andreica, **"Romanian National Olympiads in Informatics & Training"**, "Olympiads in Informatics" Journal, vol. 2, pp. 37-47, 2008. (ISSN:1822-7732)
49. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Efficient Offline Algorithmic Techniques for Several Packet Routing Problems in Distributed Systems"**, Acta Universitatis Apulensis - Mathematics-Informatics, no. 18, pp. 111-128, 2009. (ISSN: 1582-5329)
50. Mugurel Ionuţ Andreica, Ion Pârgaru, Florin Ionescu, Cristina Teodora Andreica, **"Algorithmic Aspects of Several Data Transfer Service Optimization Problems"**, "Quality - Access to Success" Journal, vol. 10, special issue no. 101, pp. 30-32, 2009. (ISSN: 1582-2559)
51. Mugurel Ionuţ Andreica, Mihai Aristotel Ungureanu, Romulus Andreica, Angela Andreica, **"An Algorithmic Perspective on Some Network Design, Construction and Analysis Problems"**, "Quality - Access to Success" Journal, vol. 10, special issue no. 101, pp. 20-22, 2009. (ISSN: 1582-2559)
52. Mădălina Ecaterina Andreica, Mugurel Ionuţ Andreica, Marin Andreica, **"Using Financial Ratios to Identify Romanian Distressed Companies"**, Economy Journal - Series Management, vol. 12, special issue no. 1, pp. 46-55, 2009. (ISSN: 1454-0320)
53. Mugurel Ionuţ Andreica, **"Efficient Gaussian Elimination on a 2D SIMD Array of Processors without Column Broadcasts"**, Politehnica University of Bucharest (UPB) Scientific Bulletin, Series C - Electrical Engineering and Computer Science, vol. 71, issue 4, pp. 83-98, 2009. (ISSN: 1454-234X)
54. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Algorithmic Solutions for Several Offline Constrained Resource Processing and Data Transfer Multicriteria Optimization Problems"**, To appear in Analele Universitatii de Vest Timisoara, Mathematics-Informatics Series, 2010. (ISSN: 1224-970X)
55. Mugurel Ionuţ Andreica, Eduard-Marius Dragomir, Nicolae Ţăpuş, **"Centralized and Decentralized Techniques for the Real-Time Scheduling of Deadline-Constrained Out-of-Order Data Transfers"**, To appear in Analele Universitatii de Vest Timisoara, Mathematics-Informatics Series, 2010. (ISSN: 1224-970X)
56. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, **"Practical Range Aggregation, Selection and Set Maintenance Techniques"**, To appear in Politehnica University of Bucharest (UPB) Scientific Bulletin, Series C - Electrical Engineering and Computer Science, 2010. (ISSN: 1454-234X)

Technical Reports

57. Mugurel Ionuţ Andreica, Nicolae Ţăpuş, Alexandru Iosup, Dick H. J. Epema, Cătălin Dumitrescu, Ioan Raicu, Ian Foster, Matei Ripeanu, **"Towards ServMark, an Architecture for Testing Grids"**, CoreGRID Technical Report 0062, November 2006.

References

[Abdallah and Buyukkaya, 2006] M. Abdallah, E. Buyukkaya, „Efficient Routing in Non-Uniform DHTs for Range Query Support", *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, pp. 239-246, 2006*.

[Afsahi and Dimopoulos, 2002] A. Afsahi, N. J. Dimopoulos, „Efficient communication using message prediction for clusters of multiprocessors", *Concurreny and Computation: Practice and Experience, vol. 14 (10), pp. 859-883, 2002*.

[Andreica, 2008a] M. I. Andreica, „Algorithmic Techniques for Several Optimization Problems Regarding Distributed Systems with Tree Topologies", *ROMAI Journal, vol. 4, no. 1, pp. 1-25, 2008.*

[Andreica, 2008b] M. I. Andreica, „Efficient Algorithmic Techniques for Several Multidimensional Geometric Data Management and Analysis Problems", *„Informatica Economica" Journal, vol. 12, no. 4 (48), pp. 17-21, 2008.*

[Andreica, 2008c] M. I. Andreica, „Optimal Scheduling of File Transfers with Divisible Sizes on Multiple Disjoint Paths", *Proceedings of the 7th IEEE Romania International Conference "Communications", pp. 155-158, 2008.*

[Andreica, 2008d] M. I. Andreica, „A Dynamic Programming Framework for Combinatorial Optimization Problems on Graphs with Bounded Pathwidth", *Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics (THETA 16) (AQTR), pp. 120-125, 2008.*

[Andreica, 2009a] M. I. Andreica, „Algorithmic Decision Optimization Techniques for Multiple Types of Agents with Contrasting Interests", *Metalurgia International, vol. 14, special issue no. 11, pp. 162-170, 2009.*

[Andreica et al., 2006] M. I. Andreica, N. Ţăpuş, A. Iosup, D. H. J. Epema, C. Dumitrescu, I. Raicu, I. Foster, M. Ripeanu, „Towards ServMark, an Architecture for Testing Grids", *CoreGRID Technical Report 0062, 2006.*

[Andreica et al., 2008] M. I. Andreica, E.-D. Tîrşa, C. T. Andreica, R. Andreica, M. A. Ungureanu, „Optimal Geometric Partitions, Covers and K-Centers", *Proceedings of the 9th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 173-178, 2008.*

[Andreica, Andreica and Andreica, 2008] M. I. Andreica, R. Andreica, A. Andreica, „Minimum Dissatisfaction Personnel Scheduling", *Proceedings of the 32nd American Romanian Academy of Arts and Sciences' International Congress, pp. 459-463, 2008.*

[Andreica, Andreica and Andreica, 2009] M. E. Andreica, M. I. Andreica, A. Andreica, „Efficient Algorithms for Several Constrained Activity Scheduling Problems in the Time and Space Domains", *Proceedings of the 33rd American Romanian Academy of Arts and Sciences' International Congress, vol. 1, pp. 59-63, 2009.*

[Andreica, Andreica and Ardelean, 2009] M. I. Andreica, M. E. Andreica, D. Ardelean, „Efficient Algorithms for Several Constrained Resource Allocation, Management and Discovery Problems", *Proceedings of the 33rd American Romanian Academy of Arts and Sciences' International Congress, vol. 1, pp. 64-68, 2009.*

[Andreica, Andreica and Cătăniciu, 2009] M. E. Andreica, M. I. Andreica, N. Cătăniciu, „Multidimensional Data Structures and Techniques for Efficient Decision Making", *Proceedings of the 10th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 249-254, 2009.*

[Andreica, Andreica and Vişan, 2009] M. I. Andreica, M. E. Andreica, C. Vişan, „Optimal Constrained Resource Allocation Strategies under Low Risk Circumstances", *Metalurgia International, vol. 14, special issue no. 8, pp. 143-154, 2009.*

[Andreica, Borozan, Bălăceanu and Ţăpuş, 2009] M. I. Andreica, I. Borozan, L.-I. Bălăceanu, N. Ţăpuş, „Fairness and QoS Enhancement Models and Techniques for Peer-to-Peer Content

Sharing Systems", *Proceedings of the 11th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009. In Press.*

[Andreica, Briciu and Andreica, 2009] M. I. Andreica, S. Briciu, M. E. Andreica, „Algorithmic Solutions to Some Transportation Optimization Problems with Applications in the Metallurgical Industry", *Metalurgia International, vol. 14, special issue no. 5, pp. 46-53, 2009.*

[Andreica, Cătăniciu and Andreica, 2009] M. E. Andreica, N. Cătăniciu, M. I. Andreica, „Econometric and Neural Network Analysis of the Labor Productivity and Average Gross Earnings Indices in the Romanian Industry", *Proceedings of the 10th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 106-111, 2009.*

[Andreica, Deac and Tipa, 2009] M. I. Andreica, V. Deac, S. Tipa, „Towards Providing Low-Risk and Economically Feasible Network Data Transfer Services", *Recent Advances in Signals & Systems, pp. 204-209, 2009 [Proceedings of the 9th WSEAS International Conference on Multimedia, Internet & Video Technologies (MIV), 2009].*

[Andreica, Dragomir and Ţăpuş, 2009] M. I. Andreica, E.-M. Dragomir, N. Ţăpuş, "Centralized and Decentralized Techniques for the Real-Time Scheduling of Deadline-Constrained Out-of-Order Data Transfers", *To Appear in Analele Universităţii de Vest Timişoara, Mathematics-Informatics Series, 2010 [2nd International Workshop on Global Computing Models and Technologies - GlobalComp, Timişoara, 26-29 September, 2009].*

[Andreica, Grigorean and Ţăpuş, 2009] M. I. Andreica, A. Grigorean, N. Ţăpuş, „Algorithms for Identifying Sequence Patterns with Several Types of Occurrence Constraints", *Proceedings of the 11th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009. In Press.*

[Andreica, Legrand and Ţăpuş, 2007] M. I. Andreica, I. C. Legrand, N. Ţăpuş, „Towards a Communication Framework based on Balanced Message Flow Distribution", *Proceedings of the IEEE International Conference on "Computer as a Tool" (EUROCON), pp. 2354-2359, 2007.*

[Andreica, Pârgaru, Ionescu and Andreica, 2009] M. I. Andreica, I. Pârgaru, F. Ionescu, C. T. Andreica, „Algorithmic Aspects of Several Data Transfer Service Optimization Problems", *„Quality - Access to Success" Journal, vol. 10, special issue no. 101, pp. 30-32, 2009.*

[Andreica and Tîrşa, 2008] M. I. Andreica, E.-D. Tîrşa, „Towards a Real-Time Scheduling Framework for Data Transfers in Tree Networks", *Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 467-474, 2008.*

[Andreica, Tîrşa, Costan and Ţăpuş, 2009] M. I. Andreica, E.-D. Tîrşa, A. Costan, N. Ţăpuş, „Offline Algorithms for Several Network Design, Clustering and QoS Optimization Problems", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 273-280, 2009.*

[Andreica, Tîrşa and Ţăpuş, 2009a] M. I. Andreica, E.-D. Tîrşa, N. Ţăpuş, „A Fault-Tolerant Peer-to-Peer Object Storage Architecture with Multidimensional Range Search Capabilities and Adaptive Topology", *Proceedings of the 5th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 221-228, 2009.*

[Andreica, Tîrşa and Ţăpuş, 2009b] M. I. Andreica, E.-D. Tîrşa, N. Ţăpuş, „Data Distribution Optimization using Offline Algorithms and a Peer-to-Peer Small Diameter Tree Architecture with Bounded Node Degrees", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 2, pp. 445-452, 2009.*

[Andreica, Tîrşa and Ţăpuş, 2009c] M. I. Andreica, E.-D. Tîrşa, N. Ţăpuş, „A Peer-to-Peer Architecture for Multi-Path Data Transfer Optimization using Local Decisions", *Proceedings of the 3rd Workshop on Dependable Distributed Data Management (WDDDM), pp. 2-5, 2009.*

[Andreica, Tîrşa, Ţăpuş, Pop and Dobre, 2009] M. I. Andreica, E.-D. Tîrşa, N. Ţăpuş, F. Pop, C. M. Dobre, „Towards a Centralized Scheduling Framework for Communication Flows in Distributed Systems", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 441-448, 2009.*

[Andreica and Ţăpuş, 2008a] M. I. Andreica, N. Ţăpuş, „Offline Algorithmic Techniques for Several Content Delivery Problems in Some Restricted Types of Distributed Systems", *Proceedings of the 2nd International Workshop on High Performance Grid Middleware (HiPerGrid), pp. 65-72, 2008.*

[Andreica and Ţăpuş, 2008b] M. I. Andreica, N. Ţăpuş, „Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control", *Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 354-357, 2008.*

[Andreica and Ţăpuş, 2008c] M. I. Andreica, N. Ţăpuş, „Optimal Scheduling of Two Communication Flows on Multiple Disjoint Packet-Type Aware Paths", *Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 137-144, 2008.*

[Andreica and Ţăpuş, 2008d] M. I. Andreica, N. Ţăpuş, „Constrained Content Distribution and Communication Scheduling for Several Restricted Classes of Graphs", *Proceedings of the 10th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 129-136, 2008.*

[Andreica and Ţăpuş, 2008e] M. I. Andreica, N. Ţăpuş, „Intelligent Strategies for Several Zero-, One- and Two-Player Games", *Proceedings of the 4th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 253-256, 2008.*

[Andreica and Ţăpuş, 2008f] M. I. Andreica, N. Ţăpuş, „Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling", *Proceedings of the 7th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pp. 285-292, 2008.*

[Andreica and Ţăpuş, 2008g] M. I. Andreica, N. Ţăpuş, „Optimal Offline TCP Sender Buffer Management Strategy", *Proceedings of the 1st IARIA/IEEE International Conference on Communication Theory, Reliability, and Quality of Service (CTRQ), pp. 41-46, 2008.*

[Andreica and Ţăpuş, 2008h] M. I. Andreica, N. Ţăpuş, „Reliability Analysis of Tree Networks Applied to Balanced Content Replication", *Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics (THETA 16) (AQTR), pp. 79-84, 2008.*

[Andreica and Ţăpuş, 2008i] M. I. Andreica, N. Ţăpuş, „Maximum Reliability K-Hop Multicast Strategy in Tree Networks", *Proceedings of the 12th IEEE International Symposium on Consumer Electronics (ISCE), pp. 169-172, 2008.*

[Andreica and Ţăpuş, 2008j] M. I. Andreica, N. Ţăpuş, „High Multiplicity Scheduling of File Transfers with Divisible Sizes on Multiple Classes of Paths", *Proceedings of the 12th IEEE International Symposium on Consumer Electronics (ISCE), pp. 516-519, 2008.*

[Andreica and Ţăpuş, 2009a] M. I. Andreica, N. Ţăpuş, „Algorithmic Solutions for Several Offline Constrained Resource Processing and Data Transfer Multicriteria Optimization Problems", *To Appear in Analele Universităţii de Vest Timişoara, Mathematics-Informatics Series, 2010 [6th International Workshop on Agents for Complex Systems - ACSys, Timisoara, 26-29 September, 2009].*

[Andreica and Ţăpuş, 2009b] M. I. Andreica, N. Ţăpuş, „Practical Range Aggregation, Selection and Set Maintenance Techniques", *Politehnica University of Bucharest (UPB) Scientific Bulletin, Series C - Electrical Engineering and Computer Science, 2010. In Press.*

[Andreica and Ţăpuş, 2009c] M. I. Andreica, N. Ţăpuş, „Efficient Offline Algorithmic Techniques for Several Packet Routing Problems in Distributed Systems", *Acta Universitatis Apulensis - Mathematics-Informatics, no. 18, pp. 111-128, 2009.*

[Andreica and Ţăpuş, 2009d] M. I. Andreica, N. Ţăpuş, „Efficient Upload Bandwidth Estimation and Communication Resource Allocation Techniques", *Recent Advances in Signals & Systems, pp. 186-191, 2009 [Proceedings of the 9th WSEAS International Conference on Multimedia, Internet & Video Technologies (MIV), 2009].*

[Andreica and Ţăpuş, 2009e] M. I. Andreica, N. Ţăpuş, „Decision Optimization Techniques for Efficient Delivery of Multimedia Streams", *Proceedings of the IEEE International Symposium on Signals, Circuits and Systems (ISSCS), vol. 2, pp. 333-336, 2009.*

[Andreica and Ţăpuş, 2009f] M. I. Andreica, N. Ţăpuş, „Online and Offline Algorithmic Techniques for Communication Performance Optimization in Distributed Systems", *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2009.*

[Andreica and Ţăpuş, 2009g] M. I. Andreica, N. Ţăpuş, „Central Placement of Storage Servers in Tree-Like Content Delivery Networks", *Proceedings of the IEEE International Conference on "Computer as a Tool" (EUROCON), pp. 1901-1908, 2009.*

[Andreica, Ungureanu, Andreica and Andreica, 2009] M. I. Andreica, M. A. Ungureanu, R.

Andreica, A. Andreica, „An Algorithmic Perspective on Some Network Design, Construction and Analysis Problems", *„Quality - Access to Success" Journal, vol. 10, special issue no. 101, pp. 20-22, 2009.*

[Argyriou and Chakareski, 2008] A. Argyriou, J. Chakareski, „Distributed Optimization of Media Flows in Peer-to-Peer Overlay Networks", *Proceedings of the IEEE Global Telecomm. Conf., 2008.*

[Baker and Coffman, Jr., 1996] B. Baker, E. G. Coffman, Jr., „Mutual exclusion scheduling", *Theoretical Computer Science, vol. 162 (2), pp. 225–243, 1996.*

[Bal et al., 2000] H. E. Bal et al., „The Distributed ASCI Supercomputer Project", *Operating Systems Review, vol. 34 (4), pp. 76-96, 2000.*

[Ben-Moshe, Bhattacharya, Shi, Tamir, 2007] B. Ben-Moshe, B. Bhattacharya, Q. Shi, A. Tamir, „Efficient Algorithms for Center Problems in Cactus Graphs", *Theoretical Computer Science, vol. 378, pp. 237-252, 2007.*

[Bender and Farach-Colton, 2000] M. A. Bender, M. Farach-Colton, „The LCA Problem Revisited", *Lecture Notes in Computer Science, vol. 1776, pp. 88-94, 2000.*

[Bender and Farach-Colton, 2004] M. A. Bender, M. Farach-Colton, „The Level Ancestor Problem Simplified", *Theoretical Computer Science, vol. 321 (1), pp. 5-12, 2004.*

[Berman et al., 2004] P. Berman, et al., „Fast Optimal Genome Tiling with Applications to Microarray Design and Homology Search", *J. Comput. Biol., vol. 11, pp. 766-785, 2004.*

[Bharambe, Agrawal and Seshan, 2004] A. R. Bharambe, M. Agrawal, S. Seshan, „Mercury: Supporting Scalable Multi-Attribute Range Queries", *Proceedings of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 353-366, 2004.*

[Birchler, Esfahanian and Torng, 1996] B. D. Birchler, A.-H. Esfahanian, E. K. Torng, „Information Dissemination in Restricted Routing Networks", *Proceedings of the International Symposium on Combinatorics and Applications, pp. 33-44, 1996.*

[Bitran and Yanasse, 1982] G. R. Bitran, H. H. Yanasse, „Computational Complexity of the Capacitated Lot Size Problem", *Management Science, vol. 28, pp. 1174-1186, 1982.*

[Bittorrent] Bittorrent. http://www.bittorrent.com/

[Bodlaender, 1996] H. L. Bodlaender, „A Linear-Time Algorithm for Finding Tree Decompositions of Small Treewidth", *SIAM Journal of Computing, vol. 25, pp. 1305-1317, 1996.*

[Bodlaender, and Fomin, 2004] H. L. Bodlaender, F. V. Fomin, „Equitable Colorings of Bounded Tree-width Graphs", *Theoretical Computer Science, vol. 349 (1), pp. 22–30, 2004.*

[Bodlaender and Kloks, 1996] H. L. Bodlaender, T. Kloks, „Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs", *Journal of Algorithms, vol. 21, pp. 358-402, 1996.*

[Borradaile and Klein, 2006] G. Borradaile, P. Klein, „An O (nlogn) Algorithm for Maximum st-flow in a Directed Planar Graph", *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM Press, pp. 524-533, 2006.*

[Brodnik and Nilsson, 2002] A. Brodnik, A. Nilsson, "An Efficient Data Structure for Advance Bandwidth Reservations on the Internet", *Proceedings of the 3rd Conference on Computer Science and Electrical Engineering, 2002.*

[Bucur and Epema, 2003] A. I. D. Bucur, D. H. J. Epema, „Trace-based Simulations of Processor Co-Allocation Policies in Multiclusters", *Proceedings of the 12th IEEE HPDC, pp. 70-79, 2003.*

[Burchard, 2005] L.-O. Burchard, "Analysis of Data Structures for Admission Control of Advance Reservation Requests", *IEEE Transactions on Knowledge and Data Engineering, vol. 17 (3), pp. 413-424, IEEE Press, 2005.*

[den Burger, Kielmann and Bal, 2005] M. den Burger, T. Kielmann, H. E. Bal, „Balanced Multicasting: High-Throughput Communication for Grid Applications", *Proceedings of the ACM/IEEE Conf. on Supercomputing, 2005.*

[Carpen-Amarie, Andreica and Cristea, 2008] A. Carpen-Amarie, M. I. Andreica, V. Cristea, „An Algorithm for File Transfer Scheduling in Grid Environments", *Proceedings of the 2nd International Workshop on High Performance Grid Middleware (HiPerGrid), pp. 33-40, 2008.*

[Carbunar, Ioannidis and Nita-Rotaru, 2004] B. Carbunar, I. Ioannidis, C. Nita-Rotaru,

„JANUS: Towards Robust and Malicious Resilient Routing in Hybrid Wireless Networks", *Proceedings of the 3rd ACM Workshop on Wireless Security, pp. 11-20, 2004.*

[Castro et al., 2003] M. Castro, et al., "Splitstream: High-Bandwidth Multicast in Cooperative Environments", *Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003.*

[Chen and Chao, 2007] K.-Y. Chen, K.-M. Chao, „On the Range Maximum-Sum Segment Query Problem", *Discrete Applied Mathematics, vol. 155, pp. 2043-2052, 2007.*

[Chen and Lih, 1994] B.-L. Chen, K.-W. Lih, „Equitable Coloring of Trees", *Journal of Combinatorial Theory, Series B, vol. 61, pp. 83–87, 1994.*

[Chen and Primet, 2007] B. B. Chen, P. V.-B. Primet, „Scheduling Deadline-Constrained Bulk Data Transfers to Minimize Network Congestion", *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid, pp. 410-417, 2007.*

[Choe, Schuff, Dyaberi and Pai 2007] Y. R. Choe, D. L. Schuff, J. M. Dyaberi, V. S. Pai, „Improving VoD Server Efficiency with Bittorrent", *Proceedings of the 15th International Conference on Multimedia, 2007, pp. 117-126.*

[Chekuri and Khanna, 2000] C. Chekuri, S. Khanna, „A PTAS for the Multiple Knapsack Problem", *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms, pp. 213-222, 2000.*

[Cîrstoiu, 2008] C. Cîrstoiu, „Optimizations in Distributed Systems - Optimization Framework for Data Intensive Applications in Large Scale Distributed Systems", *Ph.D. Thesis, Politehnica University of Bucharest, 2008.*

[Cîrstoiu, Voicu and Ţăpuş, 2008] C. Cîrstoiu, R. Voicu, N. Ţăpuş, „Framework for High-Performance Data Transfers Optimization in Large Distributed Systems", *Proceedings of the IEEE International Symposium on Parallel and Distributed Computing, pp. 385-392, 2008.*

[Coffman, Jr., Garey and Johnson, 1987] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, „Bin packing with divisible item sizes", *Journal of Complexity, pp. 406-428, 1987.*

[Cohen, Fraginaud and Mitjana, 2002] J. Cohen, P. Fraginaud, M. Mitjana, "Polynomial-Time Algorithms for Minimum-Time Broadcast in Trees", *Theory Comput. Syst., vol. 35 (6), pp. 641-665, 2002.*

[Cohen and Kaempfer, 2001] R. Cohen, G. Kaempfer, „A Unicast-based Approach for Streaming Multicast", *Proceedings of the IEEE Computer and Communication Societies Joint Conference (INFOCOM), pp. 440–448, 2001.*

[Cormen, Leiserson, Rivest and Stein, 2001] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, „Introduction to Algorithms", *MIT Press and McGraw-Hill, 2001.*

[Costan, Stratan, Tîrşa, Andreica and Cristea, 2009] A. Costan, C. Stratan, E.-D. Tîrşa, M. I. Andreica, V. Cristea, „Towards a Grid Platform for Scientific Workflows Management", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 37-44, 2009.*

[Das and Pas, 2008] K. Das, M. Pas, „An Optimal Algorithm to Find Maximum and Minimum Height Spanning Trees on Cactus Graphs", *Advanced Modeling and Optimization, vol. 10 (1), pp. 121-134, 2008.*

[Dobre, Pop, Costan, Andreica and Cristea, 2009] C. M. Dobre, F. Pop, A. Costan, M. I. Andreica, V. Cristea, „Robust Failure Detection Architecture for Large Scale Distributed Systems", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 433-440, 2009.*

[Dobre and Stratan, 2004] C. M. Dobre, C. Stratan, „Monarc Simulation Framework", *Proceedings of the RoEduNet International Conference, 2004.*

[Dorn, Fomin and Thilikos, 2007] F. Dorn, F. V. Fomin, D. M. Thilikos, „Subexponential Parameterized Algorithms", *Lecture Notes in Computer Science, vol. 4596, pp. 15-27, 2007.*

[Dumitrescu, Raicu, Ripeanu and Foster, 2004] C. Dumitrescu, I. Raicu, M. Ripeanu, I. Foster, „DiPerF: An Automated DIstributed PERformance Testing Framework", *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 289-296, 2004.*

[Eltayeb, Dogan and Ozguner, 2004] M. S. Eltayeb, A. Dogan, F. Ozguner, „Concurrent Scheduling: Efficient Heuristics for Online Large-Scale Data Transfers in Distributed Real-Time Environments", *Lecture Notes in Computer Science, vol. 3149, pp. 468-475, 2004.*

[Erlebach and Jansen, 1997] T. Erlebach, K. Jansen, „Call Scheduling in Trees, Rings and

Meshes", *Proceedings of the 30th Hawaii International Conference on System Science: Software Technologies and Architecture, pp. 221-222, 1997.*

[Ernemann, Hamscher, Schwiegelshohn, Yahyapour and Streit, 2002] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, A. Streit, „On Advantages of Grid Computing for Parallel Job Scheduling", *Proceedings of the IEEE International Conference on Cluster Computing and the Grid (CCGRID), pp. 39-49, 2002.*

[Farshi and Gudmundsson, 2005] M. Farshi, J. Gudmundsson, „Experimental Study of Geometric t-Spanners", *Lecture Notes in Computer Science, vol. 3669, 556-567, 2005.*

[FDT] FDT. http://monalisa.cern.ch/FDT/

[Federgruen and Tzur, 1991] A. Federgruen, M. Tzur, „A Simple Forward Algorithm to Solve General Dynamic Lot Sizing Models with n Periods in O(nlogn) or O(n) Time", *Management Science, vol. 37, pp. 909-925, 1991.*

[Floyd, 2003] S. Floyd, "HighSpeed TCP for Large Congestion Windows", *RFC 3649, Experimental, December 2003.*

[Galdi, Kaklamanis, Montangero and Persiano, 2001] C. Galdi, C. Kaklamanis, M. Montangero, P. Persiano, „Optimal and Approximate Station Placement in Networks", *Lecture Notes in Computer Science, vol. 2010, pp. 271-282, 2001.*

[Galinier, Habib and Paul, 1995] P. Galinier, M. Habib, C. Paul, „Chordal Graphs and Their Clique Graphs", *Lecture Notes in Computer Science, vol. 1017, pp. 358-371, 1995.*

[Gao and Steenkiste, 2004] J. Gao, P. Steenkiste, „An Adaptive Protocol for Efficient Support of Range Queries in DHT-based Systems", *Proceedings of the International Conference on Network Protocols, pp. 239-250, 2004.*

[Goldenberg, Kagan, Ravid and Tsorkin, 2005] D. Goldenberg, M. Kagan, R. Ravid, M. S. Tsorkin, „Zero Copy Sockets Direct Protocol over Infiniband – Preliminary Implementation and Performance Analysis", *Proceedings of the 13th IEEE Symposium on High Performance Interconnects, pp. 128-137, 2005.*

[Golumbic, Lipshteyn and Stern, 2005] M. C. Golumbic, M. Lipshteyn, M. Stern, „Representations of Edge Intersection Graphs of Paths in a Tree", *Proceedings of the European Conference on Combinatorics, Graph Theory and Applications, pp. 87-92, 2005.*

[Gottlob and Lee, 2007] G. Gottlob, S. T. Lee, „A Logical Approach to Multicut Problems", *Information Processing Letters, vol. 103 (4), pp. 136-141, 2007.*

[GrenchMark] GrenchMark. http://grenchmark.st.ewi.tudelft.nl/

[Gu and Grossman, 2007] Y. Gu, R. L. Grossman, „UDT: UDP-based Data Transfer for High-speed Wide Area Networks", *Computer Networks: The International Journal of Computer and Telecommunications Networking, vol. 51 (7), pp. 1777-1799, 2007.*

[Gu, Hong, Mazzucco, Grossman] Y. Gu, X. Hong, M. Mazzucco, R. L. Grossman, "SABUL: A High Performance Data Transfer Protocol", *IEEE Communication Letters, 2003.*

[Guo, Huffner, Kenar, Niedermeier and Uhlmann, 2006] J. Guo, F. Huffner, E. Kenar, R. Niedermeier, J. Uhlmann, „Complexity and Exact Algorithms for Multicut", *Proceedings of the 32nd International Conference on Current Trends in Theory and Practice of Computer Science, Lecture Notes in Computer Science, vol. 3831, pp. 303-312, 2006.*

[Gupta, Agrawal and Abbadi, 2005] A. Gupta, D. Agrawal, A. E. Abbadi, „Distributed Resource Discovery in Large Scale Computing Systems", *Proceedings of the Symposium on Applications and Internet, pp. 320-326, 2005.*

[Gupta, Birman, Linga, Demers and van Renesse, 2003] I. Gupta, K. Birman, K. Linga, A. Demers, R. van Renesse, „Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead", *Lecture Notes in Computer Science 2735, pp. 160-169, 2003.*

[Halman, 2003] N. Halman, „A Linear Time Algorithm for the Weighted Lexicographic Rectilinear 1-center Problem in the Plane", *Information Processing Letters, vol. 86 (3), pp. 121-128, 2003.*

[Hauswirth and Schmidt, 2005] M. Hauswirth, R. Schmidt, „An Overlay Network for Resource Discovery in Grids", *Proceedings of the International Workshop on Database and Expert Systems Applications, pp. 343-348, 2005.*

[He, Leigh, Yu and DeFanti, 2002] E. He, J. Leigh, O. Yu, T. A. DeFanti, "Reliable Blast UDP: Predictable High Performance Bulk Data Transfer", *Proceedings of the International*

Conference on Cluster Computing and the Grid (CCGRID), pp. 317-324, 2002.

[He, Primet and Welzl, 2005] E. He, P. V.-B. Primet, M. Welzl, "A Survey of Transport Protocols other than Standard TCP", *Global Grid Forum Draft, Data Transport Research Group, 2005.*

[Hedetniemi, Hedetniemi, and Beyer, 1982] S. M. Hedetniemi, S. T. Hedetniemi, T. Beyer, „A Linear Algorithm for the Grundy (Coloring) Number of a Tree", *Congressus Numerantium, vol. 36, pp. 351-362, 1982.*

[Henzinger and Leonardi, 2003] M. R. Henzinger, S. Leonardi, „Scheduling Multicasts on Unit-Capacity Trees and Meshes", *J. of Comp. and Syst. Sci., vol. 66 (3), pp. 567-611, 2003.*

[Hopcroft and Karp, 1973] J. E. Hopcroft, R. M. Karp, „An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs", *SIAM Journal on Computing, vol. 2 (4), pp. 225-231, 1973.*

[Iosup and Epema, 2006] A. Iosup, D. H. J. Epema, „GrenchMark: A Framework for Analyzing, Testing, and Comparing Grids", *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID), pp. 313-320, 2006.*

[Jansen, 2003] K. Jansen, „The Mutual Exclusion Scheduling Problem for Permutation and Comparability Graphs", *Information and Computation, vol. 180 (2), pp. 71-81, 2003.*

[Jansen, Mastrolilli and Solis-Oba, 2000] K. Jansen, M. Mastrolilli, R. Solis-Oba, „Approximation Algorithms for Flexible Job Shop Problems", *Lecture Notes in Computer Science, vol. 1776, pp. 68-77, 2000.*

[Jarvis and Zhou, 2001] M. Jarvis, B. Zhou, „Bounded Vertex Coloring of Trees", *Discrete Mathematics, vol. 232, pp. 145–151, 2001.*

[Jin, Wei and Low, 2004] C. Jin, D. X. Wei, S. H. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance", *Proceedings of the IEEE INFOCOM, 2004.*

[Jurca and Frossard, 2008] D. Jurca, P. Frossard, „Distributed Media Rate Allocation in Multipath Networks", *Image Communication, vol. 23 (10), pp. 754-768, 2008.*

[Kako, Ono, Hirata and Haldorsson, 2005] A. Kako, T. Ono, T. Hirata, M. M. Haldorsson, „Approximation Algorithms for the Weighted Independent set Problem", *Lecture Notes in Computer Science, vol. 3787, pp. 341-350, 2005.*

[Karp, Sahay, Santos and Schauser, 2003] R. M. Karp, A. Sahay, E. E. Santos, K. E. Schauser, „Optimal Broadcast and Summation in the LogP Model", *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures, pp. 142-153, 1993.*

[Kazaa] Kazaa. http://www.kazaa.com/

[Kelly, 2003] T. Kelly, „Scalable TCP: Improving Performance in Highspeed Wide Area Networks", *ACM SIGCOMM Computer Communication Review (April 2003) 33, 83-91, 2003.*

[Kinoshita, Shiroshita and Nagata, 1998] S. Kinoshita, T. Shiroshita, T. Nagata, „The RealPush Network: a New Push-Type Content Delivery System using Reliable Multicasting", *IEEE Transactions on Consumer Electronics, IEEE Press, pp. 1216-1224, 1998.*

[Koh and Tcha, 1991] J. Koh, D. Tcha, „Information Dissemination in Trees with Nonuniform Edge Transmission Times", *IEEE Transactions on Computers, vol. 40 (10), pp. 1174-1177, 1991.*

[Lai and Baker, 1999] K. Lai, M. Baker, „Measuring Bandwidth", *Proceedings of the IEEE INFOCOM, pp. 235-245, 1999.*

[Lai and Liao, 2001] J. R. Lai, W. Liao, „Mobile Multicast with Routing Optimization for Recipient Mobility", *IEEE Transactions on Consumer Electronics, IEEE Press, pp. 199-206, 2001.*

[Lan, Wang and Suzuki, 1999] Y.-F. Lan, Y.-L. Wang, H. Suzuki, „A Linear-Time Algorithm for Solving the Center Problem on Weighted Cactus Graphs", *Information Processing Letters, vol. 71, pp. 205-212, 1999.*

[Lee et al., 2001] J. Lee, et al., „Applied Techniques for High Bandwidth Data Transfers across Wide Area Networks", *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics, 2001.*

[Legrand et al., 2004] I. C. Legrand, et al., „MonALISA: An Agent based, Dynamic Service System to Monitor, Control and Optimize Grid based Applications", *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics, 2004.*

[Locher, Moor, Schmid, Wattenhofer, 2006] T. Locher, P. Moor, S. Schmid, R. Wattenhofer, „Free Riding in BitTorrent is Cheap", *Proceedings of the 5th Workshop on Hot Topics in Networks, 2006.*

[Lopes and Baquero, 2007] N. Lopes, C. Baquero, „Implementing Range Queries with a Decentralized Balanced Tree over Distributed Hash Tables", *Lecture Notes in Computer Science 4658, pp. 197-206, 2007.*

[Lua, Crowcroft, Pias, Sharma and Lim, 2005] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, „A Survey and Comparison of Peer-to-Peer Overlay Network Schemes", *IEEE Comm. Surveys and Tut. 7, pp. 72-93, 2005.*

[Malkhi, Naor and Ratajczak, 2002] D. Malkhi, M. Naor, D. Ratajczak, "Viceroy: a Scalable and Dynamic Emulation of the Butterfly", *Proceedings of the ACM PODC, pp. 183–192, 2002.*

[March and Teo, 2006] V. March, Y. M. Teo, „Multi-Attribute Range Queries on Read-Only DHT", *Proceedings of the International Conference on Computer Communications and Networks, pp. 419-424, 2006.*

[Marchal, Primet, Robert, and Zeng, 2005] L. Marchal, P. V.-B. Primet, Y. Robert, J. Zeng, "Optimizing Network Resource Sharing in Grids", *Proceedings of the 48th IEEE Global Telecommunication Conference, IEEE Press, pp. 123-132, 2005.*

[Maymounkov and Mazieres, 2002] P. Maymounkov, D. Mazieres, "Kademlia: A Peer-to-Peer Information System based on the Xor Metric", *Processings of the IPTPS, 2002, pp. 53–65.*

[McCormick, Smallwood and Spieksma, 2001] S. T. McCormick, S. R. Smallwood, F. C. R. Spieksma, „A Polynomial Algorithm for Multiprocessor Scheduling with Two Job Lengths", *Mathematics of Operations Research, pp. 31-49, 2001.*

[Meulpolder, Pouwelse, Epema and Sips, 2009] M. Meulpolder, J. A. Pouwelse, D. H. J. Epema, H. J. Sips, „BarterCast: A Practical Approach to Prevent Lazy Freeriding in P2P Networks", *6th Intl. Workshop on Hot Topics in P2P Systems, 2009.*

[Miloucheva, Reyes, Mahnke and Jonas, 2006] I. Miloucheva, N. Reyes, J. Mahnke, K. Jonas, „Efficient Reliable On-Demand Multicast for Content Delivery", *Proceedings of the International Conference on Software and Computer Networks, IEEE Press, 2006.*

[Mogoş and Andreica, 2009] A.-H. Mogoş, M. I. Andreica, „Approximating Mathematical Semantic Web Services Using Approximation Formulas and Numerical Methods", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 2, pp. 533-538, 2009.*

[Mohamed and Epema, 2004] H. H. Mohamed, D. H. J. Epema, „An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters", *Proceedings of the IEEE International on Conference Cluster Computing, 2004.*

[Mohamed and Epema, 2005] H. H. Mohamed, D. H. J. Epema, „Experiences with the KOALA Co-Allocating Scheduler in Multiclusters", *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the GRID, pp. 784-791, 2005.*

[Napster] Napster. http://free.napster.com/

[ns2] Network Simulator– ns2. http://nsnam.isi.edu/nsnam/index.php/User_Information

[Ocher and Cohen, 1998] A. Ocher, R. Cohen, „A Dynamic Approach for Efficient TCP Buffer Allocation", *Proceedings of the 7th IEEE International Conference on Computer Communications and Networks, 817-824, 1998.*

[Padmanabhan, Wang, Chou and Sripanijkulchai, 2002] V. N. Padmanabhan, H. J. Wang, P.A. Chou, K. Sripanijkulchai, „Distributing Streaming Media Content Using Cooperative Networking", *Proceedings of the ACM NOSSDAV, pp. 177–186, 2002.*

[Pavlo et al., 2006] A. Pavlo, P. Couvares, R. Gietzel, A. Karp, I. D. Alderman, M. Livny, „The NMI Build & Test Laboratory: Continuous Integration Framework for Distributed Computing Software", *Proceedings of the 20th USENIX Large Installation System Administration Conference (LISA), 2006.*

[Pop et al., 2009] F. Pop, C. M. Dobre, A. Costan, M. I. Andreica, E.-D. Tîrşa, C. Stratan, V. Cristea, „Critical Analysis of Middleware Architectures for Large Scale Distributed Systems", *Proceedings of the 17th International Conference on Control Systems and Computer Science (CSCS), vol. 1, pp. 29-36, 2009.*

[Popen5] Popen5. www.lysator.liu.se/~astrand/popen5

[Pouwelse et al., 2008] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, H.J. Sips, "Tribler: A Social-based Peer-to-Peer System", *Concurrency and Computation: Practice and Experience, vol. 20, pp. 127-138, 2008.*

[Pouwelse, Garbacki, Epema and Sips, 2005] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, H. J. Sips, „The Bittorrent P2P File Sharing System: Measurement and Analysis", *Proceedings of the International Workshop on Peer-to-Peer Systems, 2005.*

[Prasad, Dovrolis, Murray and Caffy, 2003] R. Prasad, C. Dovrolis, M. Murray, K. C. Caffy, „Bandwidth Estimation: Metrics, Measurement Techniques, and Tools", *IEEE Network, Vol. 17, no. 6, pp. 27-35, 2003.*

[Pruhs, Sgall and Torng, 2004] K. Pruhs, J. Sgall, E. Torng, „Online Scheduling", CRC Press, 2004.

[Ratmasamy, Francis, Handley, Karp and Shenker, 2001] S. Ratmasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A Scalable Content Addressable Network", *Proceedings of the ACM SIGCOMM Conference, 2001.*

[Rhee and Xu, 2005] I. Rhee, L. Xu, „CUBIC: A New TCP-Friendly High-Speed TCP Variant", *Proceedings of the 3rd International Workshop on Protocols for Fast Long-Distance Networks, 2005.*

[Robertson and Seymour, 1986] N. Robertson, D. Seymour, „Graph Minor II. Algorithmic Aspects of Tree Width", *Journal of Algorithms, vol. 7, pp. 309-322, 1986.*

[Rosenthal and Pino] A. Rosenthal, J. A. Pino, „A Generalized Algorithm for Centrality Problems on Trees", *Journal of the ACM, vol. 36 (2), pp. 349-361, 1989.*

[Roskind and Tarjan, 1985] J. Roskind, R. E. Tarjan, „A Note on Finding Minimum-Cost Edge-Disjoint Spanning Trees", *Mathematics and Operations Research, vol. 10 (4), pp. 701-708, 1985.*

[Rowstron and P. Druschel, 2001] A. Rowstron, P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems", *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms, 2001.*

[Rowstron, Kermarrec, Castro and Druschel, 2001] A. Rowstron, A. M. Kermarrec, M. Castro, P. Druschel, „Scribe: The Design of a Large Scale Event Notification Infrastructure", *Proceedings of the 3rd Intl. Workshop on Networked Group Communication, 2001.*

[Sahni et al., 2007] S. Sahni, N. Rao, S. Ranka, Y. Li, E.-S. Jung, N. Kamath, „Bandwidth Scheduling and Path Computation Algorithms for Connection-Oriented Networks", *Proceedings of the 6th IEEE International Conference on Networking, pp. 47-52, 2007.*

[Shakshuki, Hussain, Matin and Matin, 2006] E. Shakshuki, S. Hussain, A. W. Matin, A. R. Matin, „Agent-based Peer-to-Peer Layered Architecture for Data Transfer in Wireless Sensor Networks", *Proceedings of the IEEE Intl. Conf. on Granular Computing, 2006.*

[Sivakumar, Bailey and Grossman, 2000] H. Sivakumar, S. Bailey, R. L. Grossman, „PSockets: the Case for Application-Level Network Striping for Data Intensive Applications using High Speed Wide Area Networks", *Proceedings of the ACM/IEEE Conference on Supercomputing, 2000.*

[Slater, Cockayne and Hedetniemi, 1981] P. J. Slater, E. J. Cockayne, S. T. Hedetniemi, „Information Dissemination in Trees", *SIAM Journal on Computing, vol. 10, pp. 692-701, 1981.*

[Soldati, Zhang and Johansson, 2009] P. Soldati, H. Zhang, M. Johansson, „Deadline-Constrained Transmission Scheduling and Data Evacuation in WirelessHART Networks", *Proceedings of the European Control Conference, 2009.*

[Stoica, Morris, Karger, Kaashoek and Balakrishnan, 2001] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", *Proceedings of the ACM SIGCOMM Conference, 2001.*

[Tewari and Kleinrock, 2007] S. Tewari, L. Kleinrock, „Analytical Model for BitTorrent-Based Live Video Streaming", *Proceedings of the IEEE Consumer Communications and Networking Conference, 2007, pp. 976-980.*

[Tîrşa, Andreica and Costan, 2009] E.-D. Tîrşa, M. I. Andreica, A. Costan, „Data Replication Techniques with Applications to the MonAlisa Distributed Monitoring System", *Proceedings of the IEEE International Conference on "Computer as a Tool" (EUROCON), pp. 339-346, 2009.*

[Tran, Hua and Do, 2003] D. A. Tran, K. A. Hua, T. Do, „ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming", *Proceedings of the 22nd IEEE Conference INFOCOM, pp. 1283-1292, 2003.*

[Tsouloupas and Dikaiakos, 2005] G. Tsouloupas, M. D. Dikaiakos, „GridBench: A

Workbench for Grid Benchmarking", *Lecture Notes in Computer Science, vol. 3470, pp. 211-225, 2005.*

[Vandevoorde, 1998] D. Vandevoorde, „The Maximal Rectangle Problem", Dr. Dobb's Journal, vol. 23, pp. 30-32, 1998.

[Venkataraman, Yoshida and Francis, 2006] V. Venkataraman, K. Yoshida, P. Francis, „Chunkyspread: Multi-Tree Unstructured Peer-to-Peer Multicast", *Proceedings of the IEEE International Conference on Network Protocols, pp. 2–11, 2006.*

[Verhaegh and Aarts, 1997] W. F. J. Verhaegh, E. H. L. Aarts, "A Polynomial-Time Algorithm for Knapsack with Divisible Item Sizes", *Information Processing Letters, vol. 62 (4), pp. 217-221, 1997.*

[Wagelmans, van Hoesel and Kolen, 1992] A. Wagelmans, S. van Hoesel, A. Kolen, „An O(nlogn) Algorithm that Runs in Linear Time in the Wagner-Whitin Case", *Operations Research, vol. 40, pp. 145-156, 1992.*

[Wagner and Whitin, 1958] H. M. Wagner, T. M. Whitin, "Dynamic Version of the Economic Lot Size Model", *Management Science, vol. 5, pp. 89-96, 1958.*

[Wang and Chen, 2002] T. Wang, J. Chen, "Bandwidth Tree - a Data Structure for Routing in Networks with Advanced Reservations", *Proceedings of the 21st International Performance, Computing, and Communications Conference, IEEE Press, pp. 37-44, 2002.*

[Weichenberg, Chan and Medard, 2004] G. E. Weichenberg, V. W. S. Chan, M. Medard, „High-Reliability Architectures for Networks under Stress", *Proceedings of the 24th Joint Conference of the IEEE Computer and Communications Societies, 2004.*

[Weil and Feitelson, 2001] A. M. Weil, D. G. Feitelson, „Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling", *IEEE Trans. Parallel Distrib. Syst., 12(6), pp. 529-543, 2001.*

[Wong, 2002] W. S. Wong, „Fairness Issues in Peer-to-Peer Networks", *Proceedings of the 41st IEEE Conference on Decision and Control, pp. 2005-2010, 2002.*

[Wood, 2001] A. P. Wood, „Reliability-metric Varieties and Their Relationships", *Proceedings of the IEEE Reliability and Maintainabilty Symposium, pp. 110-115, 2001.*

[Wu et al., 2008] W. Wu, Y. Chen, X. Zhang, X. Shi, L. Cong, B. Deng, Xing Li, „LDHT: Locality-Aware Distributed Hash Tables", *Proceedings of the International Conference on Information Networking, 2008.*

[Wu and Katzela, 2000] R. Wu, I. Katzela, „MRMA: A Multicast Resource Management Architecture for IP Platforms", *Lecture Notes in Computer Science, vol. 1960, pp. 25-36, 2000.*

[Xiao, Liu, Gu, Xuan, Liu, 2006] L. Xiao, X. Liu, W. Gu, D. Xuan, Y. Liu, „A Design of Overlay Anonymous Multicast Protocol", *Proceedings of the IEEE International Parallel and Distributed Systems Symposium (IPDPS), 2006.*

[Xiong, Wu, Xing, Wu and Zhang, 2005] Q. Xiong, C. Wu, J. Xing, L. Wu, H. Zhang, "A Linked-List Data Structure for Advance Reservation Admission Control", *Lecture Notes in Computer Science, vol. 3619, pp. 901-910, 2005.*

[Yen and Chen, 2006] W. C.-K. Yen, C.-T. Chen, „The Connected p-Center Problem with Extension", *Proceedings of the Joint Conference on Information Sciences, Kaohsiung, Taiwan, 2006.*

[Zghaibeh and Harmantzis, 2008] M. Zghaibeh, F. C. Harmantzis, „Revisiting Free Riding and the Tit-for-Tat in BitTorrent: A Measurement Study", *Peer-to-Peer Networking and Applications, vol. 1 (2), 2008.*

[Zheng, Shen, Li and Shenker, 2006] C. Zheng, G. Shen, S. Li, S. Shenker, „Distributed Segment Tree: Support Range Query and Cover Query over DHT", *Proceedings of the International Workshop on P2P Systems, 2006.*

[Zissimos, Doka, Chazapis and Koziris, 2007] A. Zissimos, K. Doka, A. Chazapis, N. Koziris, „GridTorrent: Optimizing Data Transfers in the Grid with Collaborative Sharing", *Proceedings of the 11th Panhellenic Conference on Informatics, 2007.*

Mugurel Ionuț Andreica

Asistent universitar la Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București

- **Membru al comisiei centrale a Olimpiadei Naționale de Informatică (2002 - prezent)**
- **Conducător secund al delegației României la Olimpiada Internațională de Informatică (2007 - 2009)**
- **Membru al comisiei științifice a Olimpiadei de Informatică a Europei Centrale (2009)**
- **Membru al Juriului etapei regionale sud-est europene a concursului internațional ACM ICPC (București, 2009)**
- **Antrenor al echipelor Universității Politehnica din București ce au participat la concursul internațional ACM ICPC (2005 - prezent)**
- **Membru al comisiei științifice a Balcaniadei de Informatică (Iași, 2003)**

Studii:

- **2006 - 2009: studii doctorale în domeniul CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI (Facultatea de Automatică și Calculatoare, Universitatea Politehnica din București)**
  Titlul tezei de doctorat: *Tehnici pentru optimizarea fluxurilor de comunicație în sisteme distribuite.*
  Rezultatele cercetării au fost diseminate prin 57 articole științifice dintre care 27 indexate ISI.
- **2006 - 2008: studii de masterat - *Sisteme avansate pentru aplicații Internet* (Facultatea de Automatică și Calculatoare, Universitatea Politehnica din București), absolvite cu media generală 10**
- **2001 - 2006: studii de licență (Facultatea de Automatică și Calculatoare, Universitatea Politehnica din București), absolvite cu media generală 10 (șef de promoție)**
- **1997 - 2001: studii liceale (Colegiul Național „Cantemir-Vodă", București), absolvite cu media generală 9,99 (șef de promoție)**

Premii, distincții, burse, granturi:

- **Bursă de cercetare doctorală din partea IBM: IBM Ph.D. Fellowship (2008-2009)**
- **Bursă de cercetare doctorală din partea Oracle (2006-2009)**
- **Medalie de bronz la finala mondială a concursului internațional de informatică, algoritmică și programare ACM ICPC (Shanghai, 2005)**
- **Locul I la etapa regională sud-est europeană a concursului internațional de informatică, algoritmică și programare ACM ICPC (București, 2001 și 2004)**
- **Directorul unui grant de cercetare pentru tineri doctoranzi, câștigat în urma unei competiții organizate de CNCSIS / UEFISCSU (2008-2009)**
- **Participant la granturi internaționale: MonALISA (2006-prezent), EU-NCIT (FP6, 2006-2008), P2P-NEXT (FP7, 2008-2011)**
- **Participant la granturi naționale: Servere Web Construite pe Clustere de Arhitecturi Multi-core (CNCSIS / IDEI, 2009-2011), PALIROM (2009), SISEB (2007)**
- **Medalie de argint la Olimpiada Internațională de Informatică (Tampere, 2001)**
- **Premiul I la Olimpiada Națională de Informatică (2001)**
- **Locul I la Concursul Național de Programare Cupa COMPAQ (București, 2001)**
- **Locul I la Olimpiada Internațională Pluridisciplinară Tuymaada, disciplina Informatică (Yakutsk, 2000)**

ISBN: 978-973-88451-4-5
9789738845145