Towards a Real-Time Scheduling Framework for Data Transfers in Tree Networks

Mugurel Ionut Andreica, Eliana-Dina Tirsa

Towards a Real-Time Scheduling Framework for Data Transfers in Tree Networks

Mugurel Ionuţ Andreica, Eliana-Dina Tîrşa
Computer Science and Engineering Department
Politehnica University of Bucharest
Bucharest, Romania
{mugurel.andreica, eliana.tirsa}@cs.pub.ro

Abstract— **As the amount of transferred data increases world-wide, network data transfers may exhibit poor performance due to resource usage conflicts, even when enough resources are available (at a different place or at another time). Thus, efficient resource management and intelligent scheduling of the data transfers are required. These problems are, however, quite difficult to solve, both from a theoretical and a practical perspective. In this paper we propose several novel algorithmic techniques for the real-time scheduling of data transfers when the network has a tree topology. Some of these techniques can be later implemented in a (re)configurable data transfer scheduling framework.**

I. INTRODUCTION

As distributed applications and services are being developed and deployed all around the world, the amount of transferred data and the Quality-of-Service (QoS) requirements of the transfers increase at a rapid pace. The Internet only provides best-effort guarantees, which are not sufficient anymore for many classes of communication flows (e.g. multimedia streams, large file transfers). Furthermore, network resource usage conflicts occasionally occur, although enough resources are available at a different place or at another time (e.g. two or more communication flows sharing the same bottleneck network link). Under these circumstances, efficient resource management and intelligent scheduling of the communication flows are strongly required. However, these problems are quite difficult to tackle, both from a theoretical and a practical perspective. In this paper 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 and leave the preemptive case for future work.

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 [1]. 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), 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. The rest of this paper is structured as follows. In Section II we present the real-time data transfer scheduling model. In Sections III-VI we consider several non-preemptive data transfer scheduling cases for which we present efficient algorithmic techniques. In Section VII we discuss techniques for answering aggregate queries on path and tree networks. In Section VIII we consider a few fault tolerance aspects of data transfer scheduling and in Section IX we present related work. Finally, in Section X we conclude and discuss future work.

II. 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, 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 the number m of received requests equals R or until a short time limit is exceeded, if $1 \leq m < R$). 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.

III. 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 (this situation is considered in subsection III.B). 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 transfer 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 start). Thus, the optimization algorithm only needs to take care of "conflicts" among the requests in the same batch.

A. *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 corresponds 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 and sum the results together). 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 ($O(V+E)$) [2]. 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 $j \in C_i$ 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:

$$W_{max}(i,0) = \sum_{C_k \text{ son of } C_i} \max\left\{ \begin{matrix} W_{max}(k,0) \\ W_{max}(k,p), p \in (C_k \setminus C_i) \end{matrix} \right\}, \quad (1)$$

$$W_{max}(i, j \neq 0) = w(j) + \sum_{C_k \text{ son of } C_i} \begin{cases} W_{max}(k,j) - w(j), \text{if } j \in C_k \\ \max\left\{ \begin{matrix} W_{max}(k,0) \\ W_{max}(k,p), p \in (C_k \setminus C_i) \end{matrix} \right\}, \text{if } j \notin C_k \end{cases}. \quad (2)$$

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* $p \in (C_k \setminus C_{parent(k)})\}$ and using it in eq. (1) and (2), we can reduce the time complexity to $O(V^2)$ ($W_{excl}(k)$ can be computed

in $O(|C_k|)$ time, by considering every graph vertex in C_k and testing in $O(1)$ time if it also belongs to C_k's parent C_i). If the number of vertices in a maximal clique is bounded by a value U, the time complexity is, in fact $O(V \cdot U)$ (e.g. if the chordal graph is a tree or a forest, every maximal clique contains at most two vertices and the complexity becomes $O(V)$).

If the mutual exclusion graph is also a complement-reducible graph (cograph), then we can use a more efficient algorithm, designed especially for cographs. The algorithm is inspired from the ideas presented in [3]. A cograph has the property that it is either a single vertex, the (disjoint) union of two or more cographs or the complement of a cograph. Based on these properties, we can construct a co-tree associated to any co-graph. Every vertex of the co-tree denotes a cograph and its type (single vertex, union or complement). The root of the cotree corresponds to the whole graph. A simple dynamic programming algorithm goes as follows: for each node q of the cotree we will compute $mwis(q)$ and $mwclq(q)$, denoting the maximum weight independent set and the maximum weight clique in the cograph represented by the node q of the cotree. For a node q corresponding to a single vertex j, we have $mwis(q)=mwclq(q)=w(j)$. For a union of cographs, we have $mwis(q)=mwis(s_1)+\ldots+mwis(s_k)$ and $mwclq(q)=max\{ mwclq(s_1), \ldots, mwclq(s_k)\}$, where $s_1, \ldots, s_k$ are the sons of the node q in the cotree. For the complement of a cograph, we have $mwis(q)=mwclq(q')$ and $mwclq(q)=mwis(q')$, where q' is the (only) son of the node q. For the case of *edge-disjoint* data transfers we suggest using some of the approximation algorithms for the maximum weight independent set (in the mutual exclusion graph) presented in the literature (e.g. [5]), where the weights of the vertices are the profits of the corresponding transfers. Edge intersection graphs of paths in a tree (corresponding to unicast data transfers) were studied in [4], but we could not find any useful properties.

B. *Batches of Size One (one request)*

If the batch size is *1* 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 [6] and assign such a data structure (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 (e.g. a 2D structure, corresponding to graph edges or vertices in the 1^{st} dimension and time slots in the 2^{nd} dimension), together with a heavy path decomposition of the tree network [11] (see also Section VII.B). We can associate a multi-dimensional segment tree (or block partition) to each path in the decomposition. Then, when a request's path intersects multiple paths of the decomposition, we will query/update all of them appropriately. This method presents only an $O(log(n))$ performance loss over the path network case (considered in [6]). 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 range). 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 on every link in a range of edges of a path network.

C. *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. $LF_1 \leq LF_2 \leq \ldots \leq LF_m$. We consider the next available time slot to be *tcur*. We will construct a bipartite graph with all the requests on one side and all the $nt=LF_m-tcur+1$ candidate time slots t_j on the other side ($t_1=tcur$, $t_2=t_1+1$, …, $t_i=t_{i-1}+1$, $t_{nt}=LF_m$). 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. 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 is $O(min\{m,nt\})$, leading to an $O((m+nt)^4)$ time complexity, which is too large to be used in a real-time system. 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 below:

MaxCostMatching():

```
HR_match={}; HT_unm={}; HT_match={}
max_cost = 0 ; tlast(0)=0
for i=1 to m do // i=1,2,…,m
  tlast(i)=LF_i-tcur+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
      (wts, ts)=HT_match.getMax()
      HR_match.deleteMin(); HT_match.deleteMax()
      HT_unm.insert((wts, ts))
  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))
```

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 and 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. The algorithm we described is of interest by itself, because it computes efficiently a maximum cost matching in a chain bipartite graph when the edges have a special cost function and the graph is given implicitly (i.e. it is not given on an edge-by-edge basis). Such implicit representations occur in several geometric matching problems (e.g. matching black and white points on the real line, where each point has a weight and a black point can be matched only to a white point located to its left) or in sum coloring problems (in [7], an efficient algorithm for the sum coloring of chain bipartite graphs is given, with a time complexity of $O(V+E)$, where V is the number of vertices and $E=O(V^2)$ is the number of edges of the graph; using an implicit representation of the graph, that algorithm can be improved to $O(V)$, by computing in $O(1)$ time the value $tlast(i)$ for each vertex i).

D. A (Simple) 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 obtain a profit p_3 ($p_3 \leq p_2$ and may be negative).

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 profit 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 $P_{max}(i,j)$=the maximum profit obtained if we focus only on the requests $r_i, \ldots, r_j$ and consider that the requests $r_1, r_2, \ldots, r_{i-1}$ and $r_{j+1}, \ldots, r_m$ were already matched. 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 $P_{max}(i,j<i)=0$. When computing $P_{max}(i,j)$, 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 r_i or r_j to time slot t_k. We have $P_{max}(i,j)=max\{c(i,k)+P_{max}(i+1,j), c(j,k)+P_{max}(i, j-1)\}$. By computing the values $P_{max}(i,j)$ 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 $P_{max}(i, i+q-1)$

The optimal profit is $P_{max}(1,m)$. There exists another solution, 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.

IV. 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 the batch 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 structure as in subsection III.B, 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).

V. 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 below:

HyperGraph-VertexRemoval():

$H=empty$
for each vertex i **do**
  $deg(i)=0$; $removed(i)=false$

for each edge e, **such that** $i \in e$, **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** $i \in e$, **do**
for each vertex $j \neq i$, **s.t.** $((j \in e)$ **and** (**not** $removed(j))$), **do**
$H.remove((deg(j), j))$
$deg(j)=deg(j)-1$
if $(deg(j)>0)$ **then** $H.insert((value=deg(j), vertex=j))$

The non-removed vertices form an independent set. This heuristic can be used even when the requests 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.

VI. 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 x 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 [6] to every link to enhance the speed of the procedure which verifies if a request can be granted or not. We cannot use a multi-dimensional data structure here, because the pair of operations (range addition update, range minimum query), which is required in this case, is only supported by 1D data structures.

VII. Aggregate Queries on Paths and Trees

A. Aggregate Queries on Path Networks

A path network is an undirected graph composed of n vertices $1, \ldots, n$, in which the only existing edges are $(i,i+1)$ $(1 \leq i \leq n-1)$. Every edge (u,v) has a weight $we(u,v) \geq 0$ and every vertex v has a weight $wv(v) \geq 0$. We want to efficiently answer the following types of queries: what is the aggregate value of the edge (vertex) weights between vertices u and v ? The aggregated values may have inverses relative to the aggregate function $aggf$ (e.g. when $aggf=sum, xor$) or may not (e.g. $aggf=max, min$). When an inverse element exists, we can compute the prefix "sums": $swe(i)$ and $swv(i)$. We have $swe(1)=uninitialized$ (e.g. 0, for $aggf=sum, xor$) and $swe(i>1)=aggf(swe(i-1),we(i-1,i))$; $swv(0)=unitinialized$ and $swv(i>0)=aggf(wv(i),swv(i-1))$. The "sum" of the edge (vertex) weights between vertices u and v $(u<v)$ is $aggf(swe(v), swe(u)^{-1})$ $(aggf(swv(v), swv(u-1)^{-1}))$. If the aggregate function is max (min), then we can use the *Range Maximum (Minimum) Query* technique [11] on an array a consisting of the $n-1$ edge weights or the n vertex weights. This technique computes the values $m(i,j)$, representing the maximum (minimum) value of the array in the interval $[i,i+2^j-1]$. In order to compute the maximum (minimum) value on a range $[i,j]$, we compute $k=floor(\log_2(j-i+1))$. The answer is $max(min)\{m(i,k), m(j-2^k+1,k)\}$.

We will now assign a second set of weights to every edge (vertex), $we_2(u,v)$ $(wv_2(v))$, and we consider the following queries: compute the aggregate of all the $we(a,b)$ edge $(wv(a)$ vertex) weights on the path between vertices u and v which satisfy the extra condition that $we_2(a,b) \leq x$ $(wv_2(a) \leq x)$, where x is a query parameter. When all the m queries are known in advance (the offline case) and all the edge (vertex) weights are static, we sort the queries in increasing order of the parameter x: $x(1) \leq \ldots \leq x(m)$ $(x(i)$ is the x parameter of query $i)$. We will use a segment tree [6]. Initially, we will consider that the $we(u,v)$ $(wv(v))$ weights of each edge (vertex) are $uninitialized$ (e.g. they are 0, for $aggf=sum$ or xor, $-\infty$ for $aggf=max$, $+\infty$ for $aggf=min$). We traverse the queries in sorted order and, when we reach query i, we initialize the $we(u,v)$ $(wv(v))$ weights of all the edges (u,v) (vertices v) which are still $uninitialized$ and whose $we_2(u,v)$ $(wv_2(v))$ values are $\leq x(i)$. By sorting the edges (vertices) in increasing order of their $we_2(*,*)$, $(wv_2(*))$ values and maintaining a pointer to the first still $uninitialized$ edge (vertex), we can perform this procedure in overall $O(n \cdot \log(n)+m)$ time (where $\log(n)$ is the initialization time). In order to initialize the weight of an edge (vertex), we perform a *point update* in the segment tree: we set the value of the corresponding leaf and then traverse its ancestors and modify their stored aggregate values, by looking at their left and right sons. Then, the answer for query i is the answer to the range (aggregate) query on an interval $[a(i),b(i)]$ (where $a(i)$ and $b(i)$ are computed based on the two vertex parameters of query i), which can be obtained in $O(\log(n))$ time. For the online case (where the queries cannot be sorted), we need to maintain a 2D range tree. A point inserted in this range tree has coordinates $(u,we_2(u,v))$ and weight $we(u,v)$ for an edge (u,v) (and $(u,wv_2(u))$ and weight $wv(u)$ for a vertex u). A query i translates into computing the aggregate value of the weights of all the points in the 2D range $[a(i),b(i)]_x[-\infty,x(i)]$. We can also support updates of the $we(u,v)$ $(wv(u))$ values. A query/update takes $O(\log^2(n))$ time.

B. *Aggregate Queries on Tree Networks*

A tree contains n vertices and every edge (u,v) has a weight $we(u,v)$ and every vertex v has a weight $wv(v)$. We want to answer efficiently queries of the following types: what is the aggregate weight of the edge (vertex) weights on the path between the vertices u and v ? The aggregation functions can be the same as in the previous subsection. The case where the weights have inverses relative to the aggregate function $aggf$ is easy to handle. We root the tree at an arbitrary vertex r. Then, we compute two values for each vertex i: $swe(i)$=the "sum" of the edge weights on the path from the root to vertex i and $swv(i)$=the "sum" of the vertex weights on the path from the root to i. We have $swe(r)=uninitialized$ and $swe(i\neq r)=aggf(swe(parent(i)), we(parent(i),i))$; $swv(r)=wv(r)$ and $swv(i\neq r)=aggf(wv(i), swv(parent(i)))$. In order to compute the aggregate of the edge weights on the path between two vertices u and v, we compute the lowest common ancestor of the two vertices in the tree ($LCA(u,v)$). Then, the answer is $aggf(aggf(swe(u), swe(LCA(u,v))^{-1}), aggf(swe(v), swe(LCA(u,v))^{-1}))$. For the vertex case the answer is $aggf(swv(u), swv(LCA(u,v))^{-1}, swv(v), swv(LCA(u,v))^{-1}, wv(LCA(u,v)))$. As we can see, the main procedure employed in these cases is the computation of the lowest common ancestor of two vertices. A simple solution is based on the "jump pointers" idea. For each vertex i, we compute $O(log(n))$ values $Anc(i,j)$=the ancestor of vertex i located 2^j levels higher in the tree (or the tree root, if the level of vertex i is less than 2^j). We have $Anc(i,0)=parent(i)$ and $Anc(i,j>0)=Anc(Anc(i,j-1),j-1)$. We also assign to each vertex i its DFS number, $DFSnum(i)$. If $DFSnum(i)=j$, then vertex i was the j^{th} distinct vertex visited during a DFS traversal of the tree. All the DFS numbers of the descendants of a vertex i form a set of consecutive values. Thus, the DFS numbers of the vertices in $T(i)$ (vertex i's subtree) form an interval $[DFSnum(i),DFSmax(i)]$, where $DFSmax(i)$ is the maximum DFS number in vertex i's subtree. With these values, we can easily detect if a vertex i is located in the subtree of a vertex j: we must have $DFSnum(i)\in [DFSnum(j),DFSmax(j)]$. In order to compute $LCA(u,v)$, we first verify if $DFSnum(v)\in [DFSnum(u), DFSmax(u)]$ (in this case, $LCA(u,v)=u$). If the condition does not hold, we start with $j=ceil(log(n))$ and a pointer $pu=u$. While $j\geq 0$ we perform the following actions. We test if $DFSnum(v)\in [DFSnum(Anc(pu,j)),DFSmax(Anc(pu,j)))]$. If it does, we decrement j by 1; otherwise, we set $pu=Anc(pu,j)$. At the end, we have $LCA(u,v)=Anc(pu,0)$. This way, we can compute the lowest common ancestor of two vertices and answer any query in $O(log(n))$ time (with $O(n\cdot log(n))$ preprocessing). When the weights can be dynamically

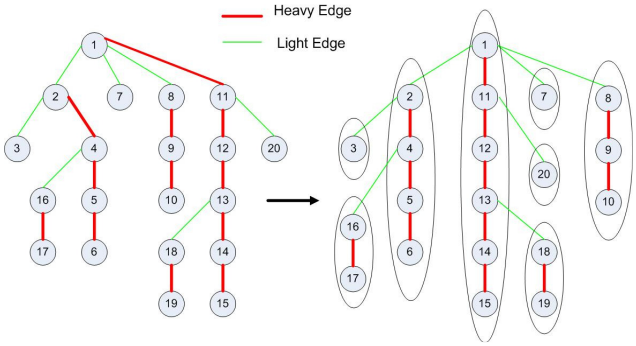

Figure 1. An example of the heavy-path decomposition technique.

updated, we need to maintain the $swe(v)$ ($swv(v)$) values up to date. In order to do this, we will maintain a segment tree, where each leaf i corresponds to the vertex j with $DFSnum(j)=i$. Initially, the values corresponding to all the leaves are $unintialized$ (e.g. 0, for $aggf=sum, xor$). Whenever we change the weight of an edge (u,v) with $u=parent(v)$ (vertex v) from $we(u,v)$ to $aggf(we(u,v),y)$ (from $wv(v)$ to $aggf(wv(v),y)$), we range update the interval $[DFSnum(v), DFSmax(v)]$ with the value y. The up to date value of $swe(i)$ ($swv(i)$) will be, at any time moment, the aggregate between the initial value and the result of the *point query* in the segment tree, corresponding to position $DFSnum(i)$ (we start at leaf i and traverse all of its ancestors, aggregating all the query aggregates stored at every node we traversed). We can also use a block partition [6] instead of the segment tree.

For the other aggregate functions (and also for the case discussed above), we can use the heavy-path decomposition technique [11] (see also Fig. 1). This technique computes a *compressed tree* CT. Each node of CT corresponds to a path of the original tree. Each edge of CT is also an edge of the original tree. Basically, the technique works as follows. We choose a vertex r and root the tree at r. For each vertex i, we compute the number of vertices in its subtree. The path PR corresponding to the root node of CT starts at r and goes to the son i of r with the largest number of vertices in its subtree. The path continues like this, going from the last added son i to i's son with the largest number of vertices, and so on. All the edges which connect a vertex of PR to a vertex outside PR belong to CT. Then, we perform the same operation, recursively, starting at every vertex which is adjacent to a vertex in PR. The edges contained inside a computed path are *heavy* edges; the others are *light* edges. The main property of the compressed tree is that a path between two vertices u and v of the original tree intersects $O(log(n))$ nodes of CT (paths in the original tree) and contains $O(log(n))$ light edges. Thus, we can solve all the problems we considered for the path networks case as follows. We compute the set of nodes SN of CT and the set of light edges SL which are intersected by the query path between u and v (by following the path between u's heavy path node and v's heavy path node in CT). We solve the problem for each element in SN and SL independently (restricting our attention to the intersection, which is

a subpath of a node in *SN*, or the edge itself, for every edge in *SL*). Then, we aggregate the results obtained for every node in *SN* and every edge in *SL*. Obviously, this implies that we will need to construct a segment tree (or a range tree, or other data structures) over the edges (vertices) of each heavy path corresponding to a node in *CT*. The time complexity is $O(log(n)\cdot P(n))$, where $P(n)$ is the time complexity in the case of a path network. If instead of the *largest number of nodes in the subtree* we use *the longest path from the vertex to a leaf* criterion, we obtain a *longest path decomposition*, which has the property that the compressed tree has height $O(n^{1/2})$, thus obtaining $O(n^{1/2}\cdot P(n))$ solutions for the problems.

VIII. Real-Time Fault Tolerant Scheduling on Multiple Disjoint Paths/Trees

In this section we consider a fault tolerant scheduling problem in the following context: There are n network nodes connected in an undirected graph topology. We partition the edges of the graph into k edge-disjoint spanning trees [1] and every such tree is controlled by a scheduler according to the scheduling model defined in Section II. An application needs to transfer data from a source node to one or more destination nodes. In each tree, the subgraph required in order to perform the data transfer is either a path (for one destination) or a subtree (for multiple destinations). In order to have an increased fault tolerance level, the transfer needs to be performed on (at least) $p \leq k$ distinct routes (paths or subtrees) in parallel. In order to do this, the application needs to use the services of (at least) p schedulers at each time moment. Let's assume the following interaction between the application and the schedulers: the application can ask each scheduler which is the largest time interval containing a given time moment t during which it can transfer the application's data within the corresponding tree (if such an interval cannot be found, the scheduler returns the first such interval after the time moment t); the application can then either reserve (part of) this interval or not. We consider that there are no cancellations of reservations and, thus, if a scheduler reports a time interval $[t_1,t_2]$ when given the time parameter t, it will report the interval $[max\{t',t_1\}, t_2]$ if given a time parameter t', $t<t'<t_2$. We will now present an online strategy for maximizing the duration during which (at least) p distinct routes are reserved, starting with a time moment t_0:

RealTimeFaultTolerantScheduling(t_0, p, k):

Step1:

```
ans={}
for s=1 to k do // s=1,2,…,k
  [t1, t2]=scheduler_s.query(t0)
  if (t1≤t0) then ans.add((int=[t0, t2], sched=s))
if (ans.size()<p) then return 0
sort the intervals in ans in decreasing order of their t2
reserved_schedulers = {}; min_heap = {}
for i=1 to p do
  s=ans[i].sched
  scheduler_ans[i].sched.reserve([t0, ans[i].int.t2])
  reserved_schedulers.add(ans[i].sched)
  min_heap.add((value=ans[i].int.t2, sched=ans[i].sched))
```

Step 2:

```
while (min_heap.size()≥p) do
  (tmin, sched)=min_heap.removeMinimum()
  reserved_schedulers.remove(sched)
  tcand=tmin ; scand=0
  for s=1 to k do
    if (not (s in reserved_schedulers)) then
      [t1,t2]=scheduler_s.query(tmin)
      if ((t1≤tmin) and (t2>tcand)) then { tcand=t2 ; scand=s }
  if (tcand>tmin) then
    scheduler_scand.reserve([tmin,tcand])
    reserved_schedulers.add(scand)
    min_heap.add((value=t2, sched=scand))
return tmin-t0 // duration=last value of tmin minus t0
```

During the first step, the algorithm queries every scheduler and reserves the time intervals of the p schedulers which contain the initial time moment and have the largest lengths. Then, during Step *2*, the algorithm maintains a heap with time moments when reservations end. At every time moment t_{min} when a reservation ends, a new reservation must be made at one of the schedulers which are not already reserved. Thus, the algorithm queries every scheduler for which no reservation containing the time moment t_{min} was made and chooses the longest interval of those answers containing the time moment t_{min}. We mention that the algorithm works equally well even if any p intervals are chosen in the beginning and any interval containing the time moment t_{min} (not necessarily the largest) is chosen when a reservation ends. However, by choosing the largest intervals, we aim at minimizing the number of reservations made (instead of many "small" reservations at several schedulers, we make one "large" reservation at one scheduler). The algorithm also works in an offline manner, if all the intervals during which reservations can be made are known in advance (the intervals corresponding to the same scheduler must be disjoint). In this

case, we sort the intervals according to their left endpoint, choose p of the intervals whose left endpoint is t_0 and then traverse the intervals (in sorted order) maintaining the same heap as before. If the left endpoint of the current interval is larger than the minimum value in the heap, then the minimum value in the heap is the largest time moment during which (at least) p schedulers can be reserved; otherwise, if the right endpoint of the current interval is larger than the minimum value in the heap, we remove the minimum value t_{min} from the heap and insert the right endpoint of the interval (meaning that the interval was reserved starting from t_{min}). The offline algorithm takes $O(n \cdot log(n))$ time (n is the total number of intervals).

IX. Related Work

The problem of efficient scheduling of data transfers is of high practical interest and has been studied by several researchers, under many restrictions. In [9], the authors study the offline scheduling of calls (non-preemptive data transfers) of unit duration and full link usage in trees, rings and meshes. In [10], efficient algorithms are presented for offline and online scheduling of unit capacity (full link usage) of multicast data transfers of large (infinite) durations in trees and meshes. In [6], the authors present an algorithmic framework for several efficient data structures which can be used for data transfer scheduling on single-link and path networks. Theoretical aspects of some of the algorithmic problems we encountered when presenting our techniques were studied in [1,2,3,4,5,8].

X. Conclusions and Future Work

In this paper we introduced a real-time data transfer scheduling model which can easily be implemented in practical settings and we presented several algorithmic techniques for real-time data transfer scheduling in tree networks. The algorithms have been implemented independently in C/C++ and Python, but we haven't integrated them into a general scheduling framework, yet. As future work, we intend to integrate some of the algorithmic techniques in a centralized data transfer scheduling framework. However, there is much work left to be done, from which we point out several interesting directions: selecting "good" batches from the pool of available requests (not only based on the time stamp of the request submission); finding good algorithms which guarantee a large global profit of the scheduled requests (most of the techniques we presented are only locally optimal); use machine learning and prediction techniques (e.g. in order to insert fictional requests which "reserve" resources for future real requests); extend the techniques to "tree-like" graphs (e.g. graphs with bounded treewidth).

References

[1] J. Roskind and R. E. Tarjan, "A Note on Finding Minimum-Cost Edge-Disjoint Spanning Trees," Mathematics and Operations Research, vol. 10 (4) , 1985, pp. 701-708.

[2] P. Galinier, M. Habib, and C. Paul, "Chordal Graphs and Their Clique Graphs," Proc. of the 21st Intl. Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, vol. 1017, 1995, pp. 358-371.

[3] R. B. Borie, R. G. Parker, and C. A. Tovey, "Solving problems on recursively constructed graphs," ACM Comput. Surveys, in press.

[4] M. C. Golumbic, M. Lipshteyn, and M. Stern, "Representations of Edge Intersection Graphs of Paths in a Tree," Proc. of the European Conf. on Combinatorics, Graph Theory and App., 2005, pp. 87-92.

[5] A. Kako, T. Ono, T. Hirata, and M. M. Haldorsson, "Approximation Algorithms for the Weighted Independent set Problem," Lecture Notes in Computer Science, vol. 3787, 2005, pp. 341-350.

[6] M. I. Andreica and N. Ţăpuş, "Efficient data structures for online QoS-constrained data transfer scheduling," Proc. of the IEEE Intl. Symp. on Parallel and Distrib. Computing (ISPDC), 2008, in press.

[7] M. Salavatipour, "On Sum Coloring of Graphs," M.Sc. Thesis, University of Toronto, 2000.

[8] D. X. Shaw, G. Cho, and H. Chang, "A Depth-First Dynamic Programming Procedure for the Extended Tree Knapsack Problem in Local Access Network Design," Telecommunication Systems, vol. 7 (1-3), 1997, pp. 29-43.

[9] T. Erlebach and K. Jansen, "Call Scheduling in Trees, Rings and Meshes," Proc. of the 30th Hawaii Intl. Conf. on System Sci., Soft. Tech. and Architecture, 1997, pp. 221-222.

[10] M. R. Henzinger and S. Leonardi, "Scheduling multicasts on unit-capacity trees and meshes," J. of Comp. and Syst. Sci., vol. 66 (3) , 2003, pp. 567-611.

[11] D. Harel and R. E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," SIAM J. Comput., vol. 13, 1984, pp. 338-355.