Decision Optimization Techniques for Efficient Delivery of Multimedia Streams

Mugurel Ionut Andreica, Nicolae Tapus

Decision Optimization Techniques for Efficient Delivery of Multimedia Streams

Mugurel Ionuţ Andreica[1], Nicolae Ţăpuş[1]
[1] Computer Science and Engineering Department, Politehnica University of Bucharest, Bucharest, Romania
{mugurel.andreica, nicolae.tapus}@cs.pub.ro

***Abstract*—With the amount of available multimedia data increasing world-wide and with the expanding need for rapid access to this data, efficient data delivery has become a significant problem for which no fully satisfactory solutions have been provided, yet. A major issue is the concurrent usage of the same bottleneck network resources by multiple communication flows. In the case of multimedia streams, which consist of large volumes of data being transferred over usually long periods of time, and which require strict quality of service guarantees, this problem is even more acute. In this paper we present several decision support and optimization techniques for one of the most common types of solutions employed nowadays – advance resource reservations.**

I. INTRODUCTION

The amount of data which is globally available and accessible has been increasing quickly during the past few years. Out of it, multimedia data (e.g. audio-video and static visual data) represents a large percentage. Since the quantity and quality of the network resources has also been increasing (e.g. higher bandwidth network links, more and faster network devices), efficient access and transfer of remote multimedia streams has become a reality for many users. This brought about unexpected increases in world-wide (multimedia) traffic, which, in order to respond to the user's expectations, must adhere to some strict quality of service guarantees (e.g. constant latency and a minimum level of guaranteed bandwidth). However, this is in contrast with the *best effort* model of the Internet (and of most local or regional area networks), which cannot provide any kind of guarantees. A solution to some of these problems, which can be applied on local, regional or Internet Service Provider (ISP) level, consists of (advance) resource reservations (the most common and most important resource is bandwidth). The (local) owner of the network links uses a resource broker (or data transfer scheduler) which receives data transfer requests. These requests can specify several constraints, like start time, finish time, minimum required bandwidth, maximum admissible latency, jitter, and so on. A request can be granted, in which case the requested resources (most commonly, network bandwidth) are reserved for the duration of the data transfer, or can be rejected. In this paper we focus on the process of decision optimization on the resource owner's side. In Section II we present new algorithmic methods for implementing profit maximizing heuristics, by choosing which requests to accept and which to reject, under certain constraints. In Section III we present several algorithmic techniques for analyzing traffic (self-) similarity over time. In Section IV we present efficient techniques for answering restricted range selection queries. In Section V we discuss related work and we conclude.

II. IMPROVING HEURISTICS BASED ON CONFLICT GRAPHS

Quite often, multiple pairs of data transfer requests are in conflict, because they require exclusive access to the same network resources, during overlapping time intervals. In such cases, a common technique [4] is to construct the conflict graph of the requests, in which every vertex corresponds to a request and an edge between two vertices i and j means that the requests i and j are in conflict. In order to maximize the profit of the accepted requests and avoid conflicts, we should compute a maximum weight independent set (MWIS) in this graph, where the weight of each vertex is the profit of the corresponding request. Computing a MWIS is an NP-hard problem for general conflict graphs, and, thus, heuristics are usually employed. Many of these heuristics are based on repeatedly extracting a vertex from the graph and either adding it to the MWIS or ignoring it from now on. In this section we present improved algorithmic techniques for the *repeated vertex extraction problem*. We will consider *4* vertex extraction rules. Let's assume that X is the set of all the vertices which have already been extracted from the graph and Y is the set of the vertices which have not been extracted, yet. At every step we will extract a node (vertex) v from Y which has the property: (1) v has a minimum number of neighbors in X; (2) v has a minimum number of neighbors in Y; (3) v has a maximum number of neighbors in X; (4) v has a maximum number of neighbors in Y. At first, $X=\{\}$ and $Y=\{1,2,...,N\}$ (N=the number of vertices of the graph). We will assign to each node u a value $val(u)$, representing the criterion used during the selection process. For the cases *1* and *3*, we will initially have $val(u)=0$ ($1 \le u \le N$); for the cases *2* and *4*, $val(u)$=the number of neighbors u has in the graph ($1 \le u \le N$). The first type of algorithms we will describe make use of a heap data structure (for the cases *1* and *2* it will be a min-heap, and for the cases *3* and *4* it will be a max-heap). At first, we will introduce in the heap all the pairs $(val(u), u)$ (the elements in the heap will be compared against each other using the key $val(u)$). At every step, we will extract from the heap the pair $(val(v), v)$ which is at the root of the heap (the one with the smallest value $val(v)$, for the cases *1* and *2*, or the one with the largest value $val(v)$, for the cases *3* and *4*). We will maintain a value $set(z)$ for every node z, denoting the set to which z currently belongs (i.e. $set(z)=X$ or $set(z)=Y$); initially, $set(z)=Y$ for all $1 \le z \le N$. We will

set *set(v)=X* and then we will traverse all the neighbors *u* of the newly extracted vertex *v*, which have the property that *set(u)=Y*. For each such neighbor, we will update the value *val(u)*: (1) at first, we remove the pair *(val(u),u)* from the heap; (2) then, for the cases *1* and *3*, we increment *val(u)* by *1*; for the cases *2* and *4*, we decrement *val(u)* by *1*; (3) we insert back into the heap the pair *(val(u), u)* (with the updated value *val(u)*). After extracting every node *v*, we process the node (we may select its corresponding request or reject it). The time complexities of these approaches are $O((N+M)\cdot log(N))$ (*M* is the number of edges of the graph).

We will now present *O(N+M)* algorithms for each of the *4* cases. We first notice that the values *val(u)* are integer numbers between *0* and *N-1*. Thus, we will replace the heap(s) with an array *VL* which contains *N* lists. *VL(x)* will contain a list with all the nodes *u* from *Y* having *val(u)=x*. After the initial computation of the values *val(u)*, we will introduce node *u* into the list *VL(val(u))* ($1 \le u \le N$). For the cases *1* and *2*, we will maintain an index *valmin* (initially *0*), and for the cases *3* and *4*, we will maintain an index *valmax* (initially equal to *N-1*). At each of the *N* steps (corresponding to a vertex extraction), we perform the following actions. For the cases *1* and *2*, as long as *VL(valmin)* contains no element, we increment *valmin* by *1*. For the cases *3* and *4*, as long as *VL(valmax)* contains no element, we decrement *valmax* by *1*. When the list *VL(valmin)* (*VL(valmax)*) contains at least one element, we choose one of the elements of the list (e.g. the first one). Let this element be *v*. We will proceed next the same way as in the case of the heap-based algorithms: we remove *v* from *VL(valmin)* (*VL(valmax)*); we set *set(v)=X*; then, we traverse all the neighbors *u* of *v*, with the property that *set(u)=Y*. For each such neighbor, we first remove it from the list *VL(val(u))*, then we update the value *val(u)* accordingly (we either increment it or decrement it by *1*, depending on the case), and then we insert *u* into *VL(val(u))* (where we consider the updated value *val(u)*). After traversing all the neighbors *u* of the newly extracted vertex *v*, we will need to update the value *valmin* (*valmax*). In case *1*, node *v* had the smallest number of neighbors in *X*. At the next step, the extracted node will have at least the same number of neighbors in *X* (since *X* has one extra element); thus, *valmin* remains unchanged. In case *2*, node *v* had the smallest number of neighbors in *Y*. In the worst case, there may be other nodes $w \in Y$ with the same number of neighbors in *Y* as *v*, and which are neighbors of *v*. After removing *v*, *val(w)* decreases by *1*. Thus, we will have to decrement *valmin* by *1*, because the next minimum value may be smaller than the current minimum value by *1*. In case *3*, we will have to increment *valmax* by *1*, because it is possible for a node $w \in Y$ to have one extra neighbor in *X* than the currently extracted vertex *v* (e.g. a node $w \in Y$ which is a neighbor of *v* and had *val(w)=val(v)* at the moment when *v* was extracted). In case *4*, the maximum number of neighbors in *Y* of a node in *Y* decreases or stays the same; thus, we do not have to change *valmax*. The time complexities of these solutions are *O(N+M)*, because every insertion and removal is performed in *O(1)* time, and the total number of increase and decrease operations of the values *valmin* or *valmax* is *O(N)*. For deletions of the vertices *u* (before updating a value *val(u)*), we can use the *lazy deletion* technique: in this case, whenever we want to delete a vertex from a list, we only mark it as being deleted (or perhaps not even that). Then, in order to check if *VL(x)* contains any node *v* which has not been previously deleted, we traverse the elements *v* in *VL(x)*; if *v* was previously marked for deletion, or $val(v) \neq x$, only now will we (physically) remove *v* from *VL(x)*. In this case, the total number of elements in all the lists can reach *O(N+M)*, but the time complexity remains the same.

Case *3* corresponds to the well-known *Maximum Cardinality Search* algorithm, which is used for identifying chordal graphs. Case *2* is used for computing the largest clique in a planar graph. A planar graph with *N* vertices has at most $3 \cdot N-6$ edges. Thus, the sum of the degrees of all of its vertices is at most $6 \cdot N-12$, meaning that there is at least one vertex with degree at most *5*. By repeatedly extracting the node *v* with the minimum degree (in *Y*), we know that this node always has at most *5* neighbors (in *Y*). Thus, we can consider all the subsets formed by *v* and its neighbors (in *Y*) and verify for each of them if it forms a clique. We will maintain the largest clique found this way. If the nodes have weights (node *i* has weight *w(i)*), we can maintain the clique with the largest sum of its nodes' weights found like this. Thus, we can compute the largest clique of a planar graph in *O(N)* time. Case *2* also has applications to finding a subgraph in which the minimum degree of a node is as large as possible: we repeatedly extract the vertex with the smallest degree (in *Y*); the result is equal with the maximum value among the degrees of the extracted nodes, at the moment when they are extracted. Case *4* is used in heuristic algorithms for computing the minimum vertex cover of a graph.

III. Online Analysis of Traffic (Self-) Similarity

An important part in efficient data transfer scheduling (i.e. acceptance and rejection of data transfer requests) is the identification of traffic patterns and (self-) similarities. Patterns make the traffic behaviour more predictable and, thus, more efficient decisions can be made. In this section we consider a time slot-based model, in which the time horizon is divided into *T* time slots. For each time slot *t* ($1 \le t \le T$), *traf(t)* denotes the amount of traffic during the time slot. We leave the problem at this more general level, in order to be able to use it in multiple places (e.g. we can have the traffic of a single data transfer, the overall traffic on a network link, and so on). For any two traffic values *a* and *b*, we have a function *eval(a, b)*, which computes information about their similarity in *O(1)* time. For instance, we can have $eval(x,y)=|x-y|$ or *eval(x,y)=if (|x-y|<threshold) then 0 else 1*, or many other functions, depending on the application. In this section we are interested in answering efficiently aggregate traffic (self-) similarity queries. In order to make the presented solution as general as possible, we will consider that we have two arrays of *T* time slots, *tr(1)* and *tr(2)*, for which we want to answer similarity queries. Self-similarity queries are answered when *tr(1)=tr(2)*. A query has three parameters: *Q(a, b, len)* ($a \le b$) and returns the aggregate of the values: *eval(tr(1)(a+i), tr(2)(b+i))* ($0 \le i \le len-1$). The aggregate function *aggf* can be, for instance, +, *max*, or any other function. Occasionally, updates can occur: *U(t, v, h)*

changes the value of $tr(h)(t)$ to v. The easiest solution is to do nothing special: we answer each query in time proportional to the *len* parameter and update a value $tr(h)(t)$ in $O(1)$ time (we simply set $tr(h)(t)=v$). However, since we expect queries to be much more frequent than updates, we need a more efficient approach. We will present three such approaches in this section. For the first approach we will divide the T time slots into T/k groups of k consecutive slots each (the last group may contain fewer than k slots). We number the groups from 1 to $ng=O(T/k)$, the total number of groups. For each group j we define $left(j)$ and $right(j)$, the first and last time slots in group j. We also define $nslots(j)=right(j)-left(j)+1$. Moreover, for each time slot t we store $group(t)=j$ if $left(j)\le t\le right(j)$. Then, we compute a table $Tagg(i,j)$=the aggregate of the values $eval(tr(1)(i+q), tr(2)(left(j)+q))$ $(0\le q\le nslots(j)-1;\ 1\le i\le T-nslots(j)+1)$. This table will be computed in the beginning in $O(T^2)$ time. A query $Q(a, b, len)$ can be answered as follows. We initialize $qagg$ to a neutral value (which depends on the aggregate function we use: e.g. 0 for +; 1 for *; $-\infty$ for max). Then, we initialize a counter idx to 0. As long as $group(b+idx)=group(b)$ and $idx<len$, we set $qagg=aggf(qagg, eval(tr(1)(a+idx), tr(2)(b+idx)))$ and then $idx=idx+1$. After this stage, if $idx<len$, then $b+idx$ is the first slot of the group $j=group(b+idx)$. While $(idx+nslots(j)-1<len)$ we set $qagg=aggf(qagg, Tagg(a+idx, j))$ and then we set $idx=idx+nslots(j)$. In the final part, if $idx<len$, we will perform the same actions as in the first part, i.e. while $(idx<len)$ we set $qagg=aggf(qagg, eval(tr(1)(a+idx), tr(2)(b+idx)))$ and then we increment idx by 1. The time complexity of a query operation is $O(k+T/k)$. An update $U(t, v, h)$ is handled as follows. We modify $tr(h)(t)$. Then, if $h=2$, we need to modify the values $Tagg(i, j)$, with $j=group(t)$. If $h=1$, we need to modify the values $Tagg(i, j)$, for which $t-nslots(j)+1\le i\le t$. We can recompute from scratch all these $O(T)$ values in $O(k)$ time per value, obtaining a time complexity of $O(T\cdot k)$. However, if the function $aggf$ is invertible (e.g. +, *xor*), we can recompute each value $Tagg(i,j)$ in $O(1)$ time as follows. We do not modify $tr(h)(t)$ at the beginning. If $h=2$, we set $Tagg(i,j)=aggf(aggf^{-1}(Tagg(i,j), eval(tr(1)(i+t-left(j)), tr(2)(t))), eval(tr(1)(i+t-left(j)), v))$. If $h=1$, we set $Tagg(i,j)=aggf(aggf^{-1}(Tagg(i,j), eval(tr(1)(t), tr(2)(left(j)+t-i))), eval(v, tr(2)(left(j)+t-i))$. After modifying all the values $Tagg(i,j)$ which need to be modified, we can set $tr(h)(t)=v$. $aggf^{-1}(a,b)$ is equivalent to $aggf(a,b^{-1})$, where b^{-1} is b's inverse value relative to the $aggf$ function (e.g. $b^{-1}=-b$, for $aggf$=+, or $1/b$, for $aggf$=*). Thus, a query is answered in $O(k+T/k)$ time, an update takes $O(T\cdot k)$ (or $O(T)$) time and the amount of occupied memory is $O(T\cdot k)$. If we choose $k=T^{1/2}$, we obtain a query time of $O(T^{1/2})$ and the amount of memory is $O(T^{3/2})$.

For the second approach we will maintain $2\cdot T-1$ segment trees. We will extend $tr(2)$ with $T-1$ time slots to the left (numbered from $-(T-2)$ to 0), having arbitrary values. Segment tree i ($ST(i)$) is built over the time slots $[i,min\{i+T-1,T\}]$ of $tr(2)$ $(-(T-2)\le i\le T)$. Every leaf $j\ge 1$ in $ST(i)$ corresponds to time slot $i+j-1$ from $tr(2)$. Every node q of $ST(i)$ whose subtree contains the leaves in the interval $[left(q),right(q)]$ maintains the aggregate of the values $eval(tr(1)(j), tr(2)(i+j-1))$ $(1\le left(q)\le j\le right(q)\le min\{T,T-i+1\})$. A query $Q(a,b,len)$ consists of a range aggregate query in $ST(b-a+1)$, for the interval of leaves $[a, a+len-1]$, which can be answered in $O(log(T))$ time. An update $U(s,v,h)$ with $h=2$ updates all the segment trees $ST(i)$ with $\max\{-(T-2),s-T+1\}\le i\le s$. We modify the value of the leaf j corresponding to slot s in $ST(i)$, $eval(tr(1)(j), tr(2)(i+j-1))$, $(j=s-i+1)$ and then we traverse the ancestors of the leaf j, starting from leaf j's parent and ending at the root of $ST(i)$. For each visited ancestor anc, we recompute the aggregate value stored at the ancestor, by aggregating the values stored at its left and right sons. For $h=1$, we update the value assigned to the leaf $j=s$ of every segment tree $ST(i)$ $(-(T-2)\le i\le T-s+1)$ and then, like before, we update the values assigned to the ancestors of these leaves in the corresponding trees. Thus, an update takes $O(T\cdot log(T))$ time. The total memory is $O(T^2)$.

We also mention a third approach, which is of interest only because it is easy to implement. It is similar to the first approach we presented. We will compute a table $Tagg2(i,j)$=the aggregate of the values $eval(tr(1)(i+q), tr(2)(j+q))$ $(0\le q\le k-1)$. A query $Q(a,b,len)$ is answered as follows. We maintain an index $idx=0$ and a partial aggregate value $qagg$. While $idx+k-1<len$, we set $qagg=aggf(qagg, Tagg2(a+idx, b+idx))$ and then $idx=idx+k$. After this part, while $idx<len$, we set $qagg=aggf(qagg, eval(tr(1)(a+idx), tr(2)(b+idx)))$ and then we increment idx by 1. The time complexity of a query is $O(k+T/k)$. An update $U(t,v,h)$ modifies $tr(h)(t)$ and then recomputes the values $Tagg2(i,j)$ with $t-k+1\le i\le t$ (if $h=1$) or $t-k+1\le j\le t$ (if $h=2$) ($O(T\cdot k)$ pairs (i,j)). We can recompute each pair in $O(k)$ time, or, if $aggf$ is invertible, in $O(1)$ time, just like we did in the first approach (by removing from $Tagg2(i,j)$ the contribution determined by $tr(h)(t)$ and replacing it by the contribution of the new value v; for instance, for $h=1$, $Tagg2(i,j)=aggf(aggf^{-1}(Tagg2(i,j), eval(tr(1)(t), tr(2)(j+t-i))), eval(v, tr(2)(j+t-i))$; for $h=2$, the equation is the same if we swap the indices i and j and the order of the arguments of the table $Tagg2$ and of the function $eval$).

IV. Range Selection Queries (and Updates)

We are given n points in $d\ge 1$ dimensions (e.g. n resources, like network links, each of them having d features, like latency or available bandwidth). Each point i has coordinates $(x(i,1), ..., x(i,d))$ and a weight $w(i)$. We want to be able to answer selection queries of the following type: find the k^{th} smallest weight among the set of weights of the points contained in a given d-dimensional range $[xa(1),xb(1)]$ x ... x $[xa(d),xb(d)]$. A point i is contained in the given range if $xa(j)\le x(i,j)\le xb(j)$ for every dimension $1\le j\le d$. We propose the following solution. We will construct for each point i a point $p(i)$ in $(d+1)$ dimensions, with coordinates $(x(i,1), ..., x(i,d), w(i))$. Then, we insert all the $p(i)$ points in a $(d+1)$-dimensional range tree RT. Such a range tree is capable of answering the following type of range count query in $O(log^{d+1}(n))$ time (or $O(log^d(n))$ time by using fractional cascading, if the set of points is static): compute the number of points in the tree which are located in a given $(d+1)$-dimensional range $[xa(1),xb(1)]$ x ... x $[xa(d+1),xb(d+1)]$. RT can be constructed in $O(n\cdot log^d(n))$ time and takes $O(n\cdot log^d(n))$ space. We can now answer a selection query (find the k^{th} smallest weight in a range) by binary searching the k^{th} smallest weight wk. We sort the weights of the

points as a preprocessing step and then binary search wk using the sorted array of point weights, where every position of the array can be accessed in $O(1)$ time; if the set of points is dynamic, we can store all the weights in a balanced tree, which can be interpreted as a sorted array, except that the value of the p^{th} position ($1 \le p \le n$) is accessed in $O(log(n))$ time. In the binary search procedure we need to perform a feasibility test which, given a parameter wt, decides if $wt \ge wk$ or $wt < wk$. The feasibility test consists of querying RT with the $(d+1)$-dimensional range $[xa(1),xb(1)]\ x\ ...\ x\ [xa(d),xb(d)]\ x\ (-\infty,wt]$. If the number p of points contained in this range is $\ge k$, then $wt \ge wk$; otherwise, $wt < wk$. With this feasibility test, the binary search computes the smallest weight wk such that there are at least k points i with $w(i) \le wk$ in the given d-dimensional range, i.e. exactly the k^{th} smallest weight in that range. The time complexity of a selection query is $O(log(n)) \cdot O(log^{d+1}(n)) = O(log^{d+2}(n))$ (or $O(log^{d+1}(n))$, if the point set is static and we use fractional cascading). If the set of points is dynamic, i.e. we can insert a new point, delete an old point or change the weight of an existing point, we need to make the range tree dynamic as well. In this case, the time complexity of an insertion, deletion or change of weight is $O(log^{d+1}(n))$ for the $(d+1)$-dimensional range tree RT. We should notice that insertion/deletion and change of weight are partially equivalent. Changing the weight of a point is equivalent to deleting the point and reinserting it with the new weight. Selection queries arise whenever we do not want to reserve the „best" resource, but the k^{th} best one.

In another application we have n network links (with 0 reservations initially) on a path, in order, from 1 to n. We can perform two types of operations. An *update* operation with parameters (o, v, i, j) adds o reservations with the same priority v ($V_{MIN} \le v \le V_{MAX}$) to each link in the range $[i,j]$. A *query* with parameters (k, i, j) asks for the k^{th} smallest priority of a reservation among all the reservations contained in a given range of links $[i,j]$. We will maintain a segment tree over the n links. Each node q of the segment tree stores its leftmost and rightmost leaf index ($left(q)$ and $right(q)$), and two balanced trees, $T_1(q)$ and $T_2(q)$ (initially empty). During an update, we compute a canonical decomposition of $O(log(n))$ tree nodes of the update interval $[i,j]$. For each tree node q in the decomposition we search a pair $(v,*)$ in $T_1(q)$. If we cannot find such a pair, then we add the pair (v,o) to $T_1(q)$. If we find a pair (v,x), we remove it from $T_1(q)$ and add the pair $(v,x+o)$ back to $T_1(q)$. For each node a which is an ancestor of a node q in the decomposition (there are $O(log(n))$ ancestor nodes), we search for a pair $(v,*)$ in $T_2(a)$. If we find such a pair (v,x), we remove it from $T_2(a)$ and add $(v,x+o \cdot ilen(a,i,j))$ back to $T_2(a)$; otherwise, we just add $(v,o \cdot ilen(a,i,j))$ to $T_2(a)$. $ilen(a,i,j)$ is the number of links in the intersection of the intervals $[left(a),right(a)]$ and $[i,j]$. Every search, delete and insert operation on $T_1(q)$ ($T_2(a)$) takes $O(log(m))$ time; m=the number of update operations which modified $T_1(q)$ ($T_2(a)$). $T_1(*)$ and $T_2(*)$ are augmented binary search trees, i.e. each node of $T_1(*)$ ($T_2(*)$) maintains the sum of the second argument and the minimum and maximum v value of all the pairs (v,x) in its subtree. Thus, we will be able to compute the sum of all the values x of the pairs (v,x) with $a \le v \le b$ in logarithmic ($O(log(m))$) time. For a query we binary search the k^{th} smallest priority vk inside the interval of links $[i,j]$, between V_{MIN} and V_{MAX}. For every candidate priority v, we compute the number p of reservations with priorities $v' \le v$. If $p \ge k$, we have $v \ge vk$; otherwise, $v < vk$. We initialize p to 0 and then we compute the canonical decomposition of the query interval $[i,j]$. For each tree node q in the decomposition, we compute in logarithmic time the value $S(q,v)$=the sum of the parameters x of the pairs (v',x) in $T_1(q)$ with $-\infty \le v' \le v$. Then, we increase p by $S(q,v) \cdot (right(q)-left(q)+1)$, i.e. by the product between $S(q,v)$ and the number of links contained in the range corresponding to the node q. We also compute $R(q,v)$=the sum of the parameters x of the pairs (v',x) in $T_2(q)$, with $-\infty \le v' \le v$ and increase p by $R(q,v)$. For every ancestor tree node a of a tree node q in the canonical decomposition, we compute $S(a,v)$, having the same meaning for node a, as $S(q,v)$ for node q; then, we increase p by $S(a,v) \cdot ilen(a,i,j)$. An update (query) takes $O(log^2(n))$ ($O(log(V_{MAX}-V_{MIN}+1) \cdot log^2(n))$) time.

V. Related Work and Conclusions

In [1], the authors analyze the offline scheduling of unit duration calls in trees, rings and meshes. In [2], the authors present efficient algorithms for offline and online scheduling of unit capacity multicast data transfers in trees and meshes. In [3], we present an algorithmic framework for several efficient data structures which can be used for data transfer scheduling on single-link and path networks. In [4], the authors introduce a wide range of algorithmic techniques for scheduling data transfers in tree networks. Several heuristic methods for data request scheduling were presented in [5]. A scheduling model based on bandwidth reservations for critical data transfers was introduced in [6]. In this paper we presented several novel algorithmic techniques for the optimization of the decision process regarding multimedia data transfer request scheduling, in the context of a centralized resource manager. The techniques were developed in order to improve both the execution time and the accuracy of the decision algorithms. Their efficiency has been thoroughly examined from a theoretical point of view.

References

[1] 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, pp. 221-222, 1997.

[2] M. R. Henzinger and S. Leonardi, "Scheduling Multicasts on Unit-Capacity Trees and Meshes", J. of Comp. and Syst. Sci., vol. 66, pp. 567-611, 2003.

[3] M. I. Andreica and N. Ţăpuş, "Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling", Proc. of the 7th IEEE Intl. Symp. on Parallel and Distributed Computing, pp. 285-292, 2008.

[4] M. I. Andreica and E.-D. Tîrşa, "Towards a Real-Time Scheduling Framework for Data Transfers in Tree Networks", Proc. of the 10th IEEE Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing, pp. 467-474, 2008.

[5] M. D. Theys, H. J. Siegel, and E. K. P. Chong, "Heuristics for Scheduling Data Requests using Collective Communications in a Distributed Communication Network", J. of Parallel and Distributed Computing, vol. 61, pp. 1337-1366, 2001.

[6] A. Hangan, R. Marfievici, and G. Sebestyen, "Reservation-Based Data Flow Scheduling in Distributed Control Applications", Proc. of the 3rd Intl. Conf. on Networking and Services, p. 10, 2007.