Constrained Content Distribution and Communication Scheduling for Several Restricted Classes of Graphs

Mugurel Ionut Andreica, Nicolae Tapus

Constrained Content Distribution and Communication Scheduling for Several Restricted Classes of Graphs

Mugurel Ionuț Andreica, Nicolae Țăpuș
Computer Science and Engineering Department
Politehnica University of Bucharest
Bucharest, Romania
{mugurel.andreica, nicolae.tapus}@cs.pub.ro

***Abstract*—In this paper we address several problems regarding content distribution (broadcast) and communication optimization and scheduling for some restricted classes of graphs (trees, intersecting cliques). For the broadcast problem in trees we introduce some new extensions and present some new algorithmic results for determining optimal offline broadcast strategies. The communication scheduling problems are also addressed from an offline algorithmic perspective, considering mutual exclusion constraints or geometric aspects.**

Keywords-broadcast strategy; tree network; intersecting cliques; mutual exclusion; communication scheduling

I. INTRODUCTION

Efficient communication is very important in many domains, like scientific computing, Grid file transfers and computations, distributed web and Grid services, wireless networks and many others. However, with the world-wide development and deployment of many distributed services, applications and systems, achieving efficient communication is becoming more difficult, due to communication bottlenecks or inefficient usage of the available network resources (because of the lack of information or of intelligent techniques). In this paper we consider several offline optimization problems which are of general interest, like optimal broadcast strategies in tree networks under several models and considering several types of restrictions and communication scheduling techniques on some restricted classes of graphs (either communication graphs, mutual exclusion graphs or time graphs). Although in order to be of practical use, the techniques should be online, we believe that efficient offline algorithms are of a huge importance, in order to provide insight into the optimal solution of a problem and in order to facilitate a reliable performance evaluation of the online algorithms.

The rest of this paper is structured as follows. In Sections II, III and IV we study several variations of the constrained single-port tree broadcast problem. In Section V we discuss several communication optimization problems, for which we present efficient algorithmic solutions. In Section VI we consider the file transfer scheduling problem with a mutual exclusion graph consisting of intersecting cliques. In Section VII we present some communication scheduling problems which are modeled using some geometric aspects. In Section VIII we present related work and in Section IX we conclude.

II. MINIMUM TIME BROADCAST IN TREES WITH SENDING CONSTRAINTS

We consider 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 [1]. 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: *(i)* at any time moment $t \geq TM$, no vertex is blocked (we have $sendb(i,t)=0$ for $t \geq TM$); *(ii)* $sendb(i,t)=sendb(i,t-TM)$, $t \geq TM$ (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.

A. 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 *(i)*, 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 *(ii)*, $T_{min}(i,t \geq TM)=T_{min}(i,t \bmod TM)+(t \operatorname{div} TM)\cdot TM$, 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 $t \leq ts(i,t,1)<ts(i,t,2)<\ldots<ts(i,t,ns(i))$, such that $sendb(i,ts(i,t,j))=0$ and $sendb(i,t')=1$, for $ts(i,t,j)<t'<ts(i,t,j+1)$, $0 \leq j \leq ns(i)-1$ (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 $t'' \geq TM$ (and $ns'(i)<ns(i)$ moments were found) - in case *(i)*, we add the moments $TM, \ldots, TM+ns(i)-ns'(i)-1$; in case *(ii)*, we can obtain $P \cdot ns'(i)$ extra moments for any $P \geq 1$, 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 $T_{min}(i,t) \leq T_{max}$. It is obvious that the $T_{min}(v,*)$ values of a vertex v are non-decreasing, i.e. $T_{min}(v,t') \leq T_{min}(v,t'+1)$. 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 $T_{min}(s(i,j),tl(s(i,j), T_{max})+1) \leq T_{max}$, 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 aforementioned property of the $T_{min}(v,*)$ values: *(i)* $TBOUND=n+TM-1$; *(ii)* $TBOUND=n \cdot TM-1$. 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 $tl(s(i,1),T_{max}) \leq tl(s(i,2), T_{max}) \leq \ldots \leq tl(s(i,ns(i)),T_{max})$. 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 $tl(s(i,j), T_{max}) \geq ts(i,t,j)$, for all the values of j ($1 \leq j \leq ns(i)$). 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, $ts(i,t+1,j)=ts(i,t,j)$ ($1 \leq j \leq ns(i)$) and we do not need to perform other computations.
- $t=ts(i,t,1)$: in this case, $ts(i,t+1,j)=ts(i,t,j+1)$ ($1 \leq j \leq ns(i)-1$) 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: *(i)* $tcs(i,j>ntcs'(i))=TM+(j-1-ntcs'(i))$; *(ii)* $tcs(i,j>ncts'(i))=tcs(i,1+((j-1) \bmod ntcs'(i))+((j-1) \operatorname{div} ntcs'(i)) \cdot TM$). 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 a 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.

B. 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 $s(i,1), s(i,2), \ldots, s(i,ns(i))$, 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, $T_{latest}(i)=tsend(i,ns(i))$.

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 downto 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"
```

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 $tcs(i,k) \le t$ (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 $O(n \cdot log(TBOUND))$. A further improvement consists of computing a function $tprev(i,t)$=the largest time moment t' such that $t' \le t$ and $sendb(i,t')=0$ (for $t \ge TM$, we have: *(i)* $tprev(i,t)=t$; *(ii)* $tprev(i,t)=max\{tprev(i,TM-1) + ((t \operatorname{div} TM)-1) \cdot TM, tprev(i,t \bmod TM)+(t \operatorname{div} TM) \cdot TM\}$). We can tabulate $tprev(*,-1 \le t < TM)$ in $O(n \cdot TM)$ time: $tprev(i,-1)=-\infty$; $tprev(i, 0 \le t' < TM)=(if\ (sendb(i,t')=0)\ then\ t'\ else\ tprev(i,t'-1))$. 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 $O(n+n\cdot TM)$ preprocessing time or $O(n\cdot TM)$ storage.

III. 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 ((i) and (ii)). 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 $T_{min}(i,t)=(if\ (recvb(i,t)=1)\ then\ +\infty\ else\ t)$. For a non-leaf vertex i, we will determine the list of time moments $tcs(i,t)$, such that $t\le tcs(i,t,1)<tcs(i,t,2)<\ldots<tcs(i,t,ntcs(i,t))$ and $sendb(i,\ tcs(i,t,j))=0$ ($1\le j\le ntcs(i,t)\le TBOUND$). 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 $s(i,1),\ \ldots,\ s(i,ns(i))$ 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 [6] 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, too. We define $tprecv(i,t)=$**if** $(t\ge TM)$ **then** $\{\ (i)\ t\ ;\ (ii)\ max\{tprecv(i,TM-1)+((t\ div\ TM)-1)\cdot TM,\ tprecv(i,t\ mod\ TM)+(t\ div\ TM)\cdot TM\}$ **else if** $(t<0)$ **then** $-\infty$ **else if** $(recvb(i,t)=0)$ **then** t **else** $tprecv(i,t-1))$ (we tabulate the values $tprecv(i,t)$, $0\le t<TM$). For a leaf i, $T_{latest}(i)=tprecv(i,\ T_{max})$; for a non-leaf vertex i, we binary search $T_{latest}(i)$ 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 time moments $T_{cand},\ \ldots,\ T_{max}$), from which we remove the edge weights and the edges $(s(i,j),t)$, with $t\ge T_{latest}(s(i,j))$. The feasibility test checks if the maximum matching has cardinality $ns(i)$.

IV. 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 $w(i)\ge 0$. 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 $t\le T$). The content is not sent further at time moments $t\ge T$ 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 $O(n\cdot T^4+n^2\cdot T^3+n^3\cdot T^2)$.

V. Some Communication Optimization Problems

In the first four problems in this section we are given an undirected weighted tree with n vertices, where every edge (u,v) (vertex v) has a weight $we(u,v)$ $(wv(v))$.

A. Maximum Weight Path in a Tree

The weight of a path $v(1),\ \ldots,\ v(k)$ (where $(v(i),v(i+1))$ is an edge in the tree, $1\le i\le k-1$) is the sum of all the values in the multiset $\{we(v(p),v(p+1))|1\le p\le k-1\}\ \cup\ \{wv(v(p))|1\le p\le k\}$. In order to compute the maximum weight path in the tree, we will

use dynamic programming. We root the tree at an arbitrary vertex r, thus defining parent-son relationships. We compute the values: $lmax(i,1)$=the largest weight of a path in $T(i)$ (vertex i's subtree) which ends at vertex i and $lmax(i,2)$=the largest weight of a path in $T(i)$ which passes through vertex i. For a leaf l, we have $lmax(l,1)=lmax(l,2)= wv(l)$. For a non-leaf vertex i, we first compute the index p_1, such that $we(i,s(i,p_1))+lmax(s(i,p_1),1)=max\{we(i,s(i,j))+ lmax(s(i,j),1)|1\leq j\leq ns(i)\}$. We have $lmax(i,1)=wv(i)+max\{ we(i,s(i,p_1))+lmax(s(i,p_1),1), 0\}$. If $ns(i)=1$, $lmax(i,2)= lmax(i,1)$. Otherwise, if $ns(i)>1$, we compute the index p_2 such that $we(i,s(i,p_2))+lmax(s(i,p_2),1)=max\{we(i,s(i,j))+ lmax(s(i,j),1)|1\leq j\leq ns(i), j\neq p_1\}$. Then, $lmax(i,2)=lmax(i,1)+ max\{we(i,s(i,p_2))+lmax(s(i,p_2),1),0\}$. The largest weight of a path is $max\{max\{lmax(i,1),lmax(i,2)|1\leq j\leq n\},0\}$. Finding the actual path can be performed easily, by (re)computing at each vertex of the path the index p_1 (and, possibly, the index p_2) and then tracing the path in $T(p_1)$ (and, possibly, $T(p_2)$).

B. Maximum Weight Path Cover in Trees

A path cover of a graph is a set of vertex disjoint paths, such that every vertex of the graph belongs to exactly one path. A classical problem in graph theory is to compute a path cover consisting of a minimum number of paths [9]. We want to compute a path cover for which the sum of the weights of the edges composing the paths is maximum. We will use dynamic programming. We root the tree at an arbitrary vertex r and compute the values $WA(i)$ and $WB(i)$ for each vertex i: $WA(i)$ is the largest total weight of a path cover of $T(i)$ such that i is an endpoint of some path in the path cover. $WB(i)$ is the same as $WA(i)$, except that i is an inner node of some path of the path cover. For a leaf vertex l we have $WA(l)=0$ and $WB(l)=-\infty$. For a non-leaf vertex i we perform the following actions. For each son $s(i,j)$ ($1\leq j\leq ns(i)$) we compute $W_{dif}(s(i,j))=WA(s(i,j))+we(i,s(i,j))-max\{WA(s(i,j)),WB(s(i,j))\}$ and we let $W_{dif,max}(i)=max\{ W_{dif}(s(i,j))|1\leq j\leq ns(i)\}$. We also compute $WAB_{sum}(i)$=the sum of the values in the multiset $\{max\{WA(s(i,j)),WB(s(i,j))\}|1\leq j \leq ns(i)\}$. If $W_{dif,max}(i)>0$, then we have $WA(i)=WAB_{sum}(i)+ W_{dif,max}(i)$ (we connect vertex i to its son $s(i,j)$ with the maximum value $W_{dif}(s(i,j))$); otherwise, $WA(i)=WAB_{sum}(i)$ (we construct a path composed only of vertex i). If $ns(i)=1$, then $WB(i)=-\infty$. If $ns(i)\geq 2$, then we choose the two distinct sons $s(i,a)$ and $s(i,b)$ for which $W_{dif}(s(i,a))$ and $W_{dif}(s(i,b))$ are the two largest values (among all the $ns(i)$ sons). We have $WB(i) = WAB_{sum}(i) + W_{dif}(s(i,a)) + W_{dif}(s(i,b))$. The optimal total weight of the path cover is $max\{WA(r),WB(r)\}$. The time complexity is $O(n)$.

We can use the same technique for computing the minimum delay path cover consisting of a minimum number of paths. Let's assume that each edge (u,v) of the tree has a delay $d(u,v)\geq 0$. We set its weight $w(u,v)$ to $-d(u,v)+C$, where C is a large constant (for instance, $C>n\cdot max\{d(u,v)|(u,v)$ *is an edge in the tree*$\}$). Using the algorithm described above on this set of weights will maximize the number of edges which belong to the path cover (this is equivalent to minimizing the number of paths). It will also maximize the sum of the negated delays (i.e. minimize the sum of the delays) of the edges composing the paths of the path cover.

C. Maximum Weight Matching in Trees

A maximum weight matching is a matching for which the sum of the weights of the edges in the matching is maximum. Such a matching can be computed in linear time, using a bottom-up approach. We compute for each vertex i the values $WA(i)$ ($WB(i)$), representing the maximum weight of a matching in $T(i)$ which contains (does not contain) an edge adjacent to i. For a leaf vertex l, $WA(l)=-\infty$ and $WB(l)=0$. For a non-leaf vertex i, we have $WB(i)$=the sum of the values in the multiset $\{max\{WA(s(i,j)), WB(s(i,j))\}|1\leq j\leq ns(i)\}$. In order to compute $WA(i)$, we will use the same approach as in the case of the optimal path cover problem. We compute for each son the value $W_{dif}(s(i,j))=WB(s(i,j))+ we(i,s(i,j))-max\{WA(s(i,j)),WB(s(i,j))\}$. Then, we compute $W_{dif,max}(i)$ and $WAB_{sum}(i)$ as before, and set $WA(i)$ to $WAB_{sum}(i)+W_{dif,max}(i)$.

D. A Peeling Algorithm for the Center of a Tree with Non-Uniform Edge Lengths

The center of a tree is one of the vertices v for which $max(dist(v,x))$ is minimum, with x ranging over all the tree vertices and $dist(v,x)$ representing the distance between the vertices v and x (the sum of edge weights of the unique path between v and x). Computing the center of a tree has important applications in generating optimal broadcast strategies in the multiple-port model. If a vertex may send a piece of content to any number of neighbors at the same time and the edge lengths represent the durations of transmitting the content along the edges, then the tree center is a vertex for which a minimum time broadcast strategy can be achieved. When all the edges have equal length, a well-known linear algorithm for computing the center (or bi-center) of a tree is the peeling algorithm: remove all the leaves of the tree T, thus obtaining a smaller tree T_1; remove all the leaves of T_1, thus obtaining a smaller tree T_2 and so on, until we obtain a tree with only one or two vertices. We can implement this algorithm by computing the leaf layer number $layer(i)$ of every vertex i. We insert all the leaves of the initial tree in a queue Q and set their layer number to 1; then, for each element x of Q, we decrease the degree (number of vertices) of its only (remaining) neighbor y; if vertex y's degree becomes 1, then we insert y into Q and set $layer(y)=layer(x)+1$. The vertex (vertices) with the largest layer number(s) is (are) the tree center(s). In the case of edge weights we insert all the leaves into a priority queue Q. We will maintain a value $d(i)$ for each vertex i of the tree, which is initialized to 0 for all the vertices. $d(i)$ represents the maximum distance from vertex i to a leaf of the tree, passing only through vertices which were previously inserted in Q. After a vertex x decreases the degree of its only neighbor y, if y's degree is ≥ 1, we adjust $d(y)$, by setting $d(y)=max\{d(y), d(x)+we(x,y)\}$ (if y's degree becomes 1, we insert y into Q). We will always extract the vertex x with the

minimum value of $d(x)$ from Q. The vertex (vertices) i with the largest value(s) $d(i)$ is (are) the tree center(s). The time complexity of this algorithm is $O(n \cdot log(n))$, if we implement Q as a (binary) heap.

Although our algorithm is interesting from a theoretical point of view, a more efficient $O(n)$ algorithm exists [7]. This algorithm roots the tree at a vertex r and then computes, in a first traversal of the tree, the values $d_1(i)$, representing the maximum distance from a vertex i to a leaf in its subtree and $d_2(i)$ representing the second maximum distance from a vertex i to a leaf in its subtree. We have $d_1(i)=d_2(i)=0$ for a leaf vertex i and $d_1(i)=max\{we(i,j)+d_1(j)\}$ for a non-leaf vertex i, where j ranges over all the sons of i. Let's assume that $d_1(i)$ is obtained by going through a son $f_1(i)$. Then, $d_2(i)=max\{0, we(i,k)+d_1(k)\}$, where k ranges over all the sons of i, except $f_1(i)$. In a second traversal, the algorithm computes $dmax(i)$=the maximum distance from a vertex i to any other vertex in the tree. For the root r, $dmax(r)=d_1(r)$. For the other vertices i, $dmax(i)=max\{d_1(i), we(i,parent(i)) + (\textbf{if } (f_1(parent(i))=i) \textbf{ then } d_2(parent(i)) \textbf{ else } d_1(parent(i)))\}$.

E. Finding Edge Weights from the Shortest Path Matrix

We are given an arbitrary undirected graph with n vertices and m edges, together with its shortest path matrix D, where $D(i,j)>0$ (for $i \neq j$) is the length of the shortest path between the vertices i and j. We want to assign a (non-negative) weight $we(u,v)$ to every graph edge (u,v), such that the length of the shortest path between u and v (using the assigned weights) is equal to $we(u,v)$. We present here two solutions. For the first one, we sort all the $O(n^2)$ distances in increasing (non-decreasing) order and we will initialize a matrix E with $E(i,j)=+\infty$ (for every ordered pair (i,j), $i \neq j$) and $E(i,i)=0$. We will now traverse the sequence of distances (in sorted order). At any moment, $E(i,j)$ will be the shortest distance between i and j, considering only the distances traversed so far. Let's assume that we reached the distance $D(i,j)$ between the vertices i and j. If ($E(i,j)<D(i,j)$) or ($E(i,j)>D(i,j)$ and there is no edge (i,j) in the graph), then no feasible edge weight assignment exists. If $E(i,j) \geq D(i,j)$ and the edge (i,j) exists in the graph, then we set $we(i,j)=we(j,i)=D(i,j)$. We will now update all the entries in the matrix E which are modified by the new edge weight assignment. We consider every (ordered) pair of vertices (a,b) and set $E(a,b)$ to $min\{E(a,b), E(a,i)+we(i,j)+E(j,b), E(a,j)+we(j,i)+E(i,b)\}$. If no contradiction was encountered, then we were able to find a feasible edge assignment. The time complexity of this algorithm is $O(n^2 \cdot log(n)+n^4)$.

A much easier $O(n^3)$ time algorithm is the following. We assign to each edge (i,j) of the graph the weight $we(i,j)=D(i,j)$. Then, we compute the shortest paths between every pair of vertices (e.g. using the Floyd-Warshall algorithm). We denote the length of the shortest path between vertices i and j (using the weights $we(i,j)$) by $E(i,j)$. If $E(i,j)=D(i,j)$ for every (ordered) pair (i,j), then the assignment was feasible. Otherwise, no feasible assignment exists. In this case, we can have $D(i,j)=0$ ($i \neq j$).

VI. Scheduling File Transfers with a Mutual Exclusion Graph – M Intersecting Cliques

We are given 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 $M \geq 2$ 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:

$$PM_j(i,t_1,t_2,\ldots,t_{2k}) = \max\left\{\begin{array}{c} PM_j(i-1,t_1,t_2,\ldots,t_{2k}) \\ p_{X_j(i)} + PM_j(i-1,t_1-d_{X_j(i)},t_2,\ldots,t_{2k}), \text{if } (t_1-d_{X_j(i)} \geq 0) \\ p_{X_j(i)} + PM_j(i-1,t_1,t_2+d_{X_j(i)},\ldots,t_{2k}), \text{if } (t_2+d_{X_j(i)} \leq t_3) \\ \ldots \\ p_{X_j(i)} + PM_j(i-1,t_1,t_2,\ldots,t_{2k-1}-d_{X_j(i)},t_{2k}), \text{if } (t_{2k-1}-d_{X_j(i)} \geq t_{2k-2}) \\ p_{X_j(i)} + PM_j(i-1,t_1,t_2,\ldots,t_{2k-1},t_{2k}+d_{X_j(i)}), \text{if } (t_{2k}+d_{X_j(i)} \leq T) \end{array}\right\}. \quad (1)$$

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 $Spe(1)<Spe(2)<\ldots<Spe(q)$. 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.

VII. Geometric Scheduling Problems

In this section we discuss three communication scheduling problems with relevance to middleware scheduling applications (i.e. those applications that receive requests from a higher layer and use the services of a lower layer), for which we use geometric models.

A. K-Interval Cover

An application receives *n* communication requests represented by *n* request intervals *[ts(i), tf(i)]*, meaning that the i^{th} request asks for communication services between the time moments *ts(i)* and *tf(i)*. For now, we shall assume that all the intervals are pair-wise disjoint. The scheduling application knows all the requests in advance. In order to offer communication services to the higher layer, it needs to use the services of the lower layers. In our problem, we assume that the lower layer services offer functions like: *begin* the communication and *end* the communication. Between one *begin* call and one *end* call, the scheduling application has exclusive access to the communication line. In order to fulfill all the *n* requests, the scheduling application needs to have exclusive access to the communication line during each interval. This can easily be achieved by performing a *begin* call at the smallest left endpoint of an interval and an *end* call at the largest right endpoint. However, this is inefficient, as the scheduling application may block other applications using the communication line. Another solution consists of calling *begin* at the beginning of each interval and calling *end* at the end of each interval. This is also inefficient, because performing a *begin* and an *end* call implies some non-negligible overhead. The compromise solution we consider in this section is to allow the scheduling application to perform (at most) *k begin* and *k end* calls ($1 \le k \le n$) in such a way that the total duration during which the scheduling application has exclusive access to the communication line is minimized (but it has exclusive access during the *n* given time intervals).

We will solve this problem by first sorting the intervals according to their left endpoint (because the intervals are disjoint, they are also sorted according to their right endpoint): $ts(1) \le ts(2) \le \ldots \le ts(n)$. The *k reserved* time intervals during which the scheduling application is allowed to get exclusive access to the communication line must include the *n* time intervals of the requests. Furthermore, each of the *k* reserved intervals will start at the left endpoint of a request interval and end at a right endpoint of a request interval. The *k* reserved intervals are separated by *k-1 free* time intervals during which the communication line is not occupied by the scheduling application. Since the endpoint of the first reserved interval is *ts(1)* and the right endpoint of the last reserved interval is *tf(n)*, the problem actually translates into maximizing the sum S_{free} of the free intervals (because the length of the *k* reserved intervals will be $tf(n)-ts(1)-S_{free}$. Moreover, the *k-1* free intervals are a subset of the *n-1* intervals located in between two consecutive request intervals. Thus, by choosing the *k-1* largest such intervals, we can solve our problem. We will build an array *d*, where $d(i)=ts(i+1)-tf(i)$ ($1 \le i \le n-1$); after computing the values of the array *d*, we sort them, such that $d(o(1)) \le d(o(2)) \le \ldots \le d(o(n))$. The *k-1* free intervals will have lengths: *d(o(n)), d(o(n-1)), …, d(o(n-k+2))*.

If the *n* request intervals are not disjoint, we can find a minimum length set of disjoint intervals which include all the request intervals. In order to do this, we sort the left and right endpoints of the intervals (obtaining $2 \cdot n$ sorted points). Then, we traverse these points from left to right and maintain a counter *nopen*, representing the number of open intervals. Whenever we encounter a left endpoint, we increase the counter and when we encounter a right endpoint we decrease a counter. If, after a left endpoint, we have *nopen=1*, we store the value of the left endpoint into a variable *x*. If, after a right endpoint, we have *nopen=0*, we store the value of the right endpoint into a variable *y* and add the interval *[x,y]* to the set of disjoint intervals we are constructing (*x* was previously assigned the value of a left endpoint). This set of disjoint intervals represents, in fact, the union of the *n* request intervals. Now we can use the algorithm we presented for the set of disjoint intervals that we computed.

B. Piercing Intervals with a Given Number of Points

An application has *n* time intervals *[a(i),b(i)]* ($1 \le i \le n$) during which it can receive data from a remote application (*a(i)* and *b(i)* are time slot indices). For each time interval *i* ($1 \le i \le n$), we need to choose (at least) *c(i)* time slots during which the communication line is reserved exclusively for the application. The total number of time slots during which the communication line is reserved should be minimum. In geometric terms, we want to compute a set *S* of points located at integer coordinates (we consider a discrete model), such that the intersection of *S* with every interval *[a(i),b(i)]* consists of at least *c(i)* points and the cardinality of *S* is minimum. The case *c(i)=1* is well-known: We sort the right end-points of the intervals in ascending order. We maintain a variable *xp* (initialized to $-\infty$), the x-coordinate of the rightmost point added so

far to *S*. We traverse the intervals in the sorted order. Whenever $xp<a(i)$, we add a new point at *b(i)* (and, thus, we set *xp* to *b(i)*). When the *c(i)* values are bounded by a small value *U*, we can easily extend the approach presented previously. We maintain the rightmost $m \le U$ selected points. Then, while traversing the sorted intervals, we linearly (in *O(U)* time) compute the number *x(i)* of selected points which are located inside the current interval. Then, we select the remaining $extra(i)= max\{0,c(i)-x(i)\}$ points as close as possible to *b(i)* (we consider every integer coordinate from *b(i)* towards *a(i)* and select the first *extra(i)* integer coordinates which were not already selected). Then, we update the set of rightmost *U* selected point. This approach takes $O(n \cdot log(n)+n \cdot U)$ time.

In the general case, we will maintain the set *S* as a stack of sorted maximal intervals *[p(1),q(1)]*, …, *[p(k),q(k)]*, where *k* is the total number of intervals and $q(i)+1<p(i+1)$ $(1 \le i \le k-1)$. We will also maintain a stack *np*, where *np(i)* is the sum of the lengths of the intervals *[p(j),q(j)]* $(1 \le j \le i)$ (we will consider $np(0)=0$). We will sort the intervals just as in the $c(i)=1$ case, in ascending (non-decreasing) order of their right endpoints. We then initialize *S* to the empty set and traverse the intervals according to the sorted order. Let's assume that we arrived at an interval *i*. We first need to compute how many points of *S* are already placed inside *[a(i),b(i)]*. If *S* is empty, we place the *c(i)* points at coordinates *b(i)-c(i)+1, …, b(i)*. Thus, we have $[p(1)=b(i)-c(i)+1,q(1)=b(i)]$ and $np(1)=c(i)$. If $a(i)>q(k)$, then, again, we need to place *c(i)* points at the same coordinates, and then add the interval *[b(i)-c(i)+1,b(i)]* at the top of the stack; we increment *k* and set $np(k)=np(k-1)+c(i)$. If we now have $p(k)=q(k-1)+1$, we need to combine the two topmost intervals, decrement *k* and set *np(k)* accordingly. In the other cases, we will binary search the smallest index *j* $(1 \le j \le k)$, such that $a(i) \le q(j)$. The number of points located inside the interval *i* is $x(i)=np(k)-np(j)+(q(j)-max\{a(i), p(j)\}+1)$. If $c(i)>x(i)$, then we still need to add $extra(i)=c(i)-x(i)$ points to *S*, which are strictly located inside *[a(i),b(i)]*. We will try to add these points as close as possible to *b(i)*. The pseudocode below describes this part.

AddExtraPoints(i):

```
if (b(i)=q(k)) then {xleft=p(k)-1; S.pop(); k=k-1} else xleft=b(i)
while (extra(i)>0) do
  if (S.isEmpty() or (q(k)+1<xleft-extra(i)+1)) then
    S.push([p(k+1)=xleft-extra(i)+1,q(k+1)=b(i)])
    np(k+1)=np(k)+(b(i)-xleft+extra(i)); k=k+1; extra(i)=0
  else
    extra(i)=extra(i)-(xleft-q(k))
    xleft=p(k)-1; S.pop(); k=k-1
```

The overall time complexity of the algorithm is $O(n \cdot log(n))$, because of the sorting stage and because of the binary search performed for each left endpoint of each interval. Let's consider now a variation of the problem we just discussed. We have the same problem parameters, except that the intervals are open on the left side, i.e. *(a(i),b(i)]*) and we can place points anywhere, not only at integer coordinates. This simplifies the problem solution considerably. We sort all the $2 \cdot n$ left and right endpoints *a(i)* and *b(i)* of the intervals, maintaining their type (left or right) and the index of their interval (*i*). If several endpoints are located at the same coordinate, we place the right endpoints before the left endpoints at that coordinate. We then traverse the endpoints in the sorted order, maintaining a counter *np* representing the number of points we placed. When we encounter the left endpoint of an interval *i*, we store the current value of *np* in *npc(i)*. Then, when reaching the right endpoint of an interval *i*, we compute $extra(i)=c(i)-(np-npc(i))$. If $extra(i)>0$, we place *extra(i)* points infinitesimally close to *b(i)* (strictly inside the interval *(pb(i),b(i)]*, where *pb(i)* is the largest previous coordinate strictly smaller than *b(i)*).

C. *Piercing d-dimensional Identical Hyper-Rectangles with Exactly One Point per Hyper-Rectangle*

We are given *n* requests for *d* linearly ordered resources (e.g. time, frequency). Each request *i* is specified by an equally-sized d-dimensional *acceptability* range $[xr(i,1), xr(i,1)+L(1)] \times \ldots \times [xr(i,d),xr(i,d)+L(d)]$. We need to find a subset of points *S* in the d-dimensional resource space, such that there is exactly one point in every acceptability range.

In geometric terms, we are given *n* d-dimensional identical hyper-rectangles, of side lengths *L(j)* $(1 \le j \le d)$ (*L(j)* is the side length in dimension *j*). Hyper-rectangle *i* has its "lower" corner at coordinates *(xr(i,1), …, xr(i,d))* (thus, its upper corner is at $(xr(i,1)+L(1), \ldots, xr(i,d)+L(d))$). We want to select a subset of points *S* such that every hyper-rectangle is pierced by exactly one point. We are not interested in minimizing the cardinality of *S*. It turns out that this problem is, in fact, quite easy. We will sort the $2 \cdot n$ coordinates of the hyper-rectangles in every dimension and compute δ, the minimum difference between two consecutive distinct coordinates in any dimension. Then, we consider a d-dimensional orthogonal grid *G*, whose origin is at $(o_1,\ldots,o_d)$, with $o_j=min\{xr(*,j)\}+\delta/2$ $(1 \le j \le d)$ and the distance between two grid points is *L(j)* in dimension *j* $(1 \le j \le d)$. Then, every hyper-rectangle *i* contains exactly one grid point *p(i)* of *G* inside itself. The coordinates of *p(i)* in *G* are $xp(i,j)=(xr(i,j)+L(j)-o_j) \ div \ L(j)$ $(1 \le j \le d)$. The coordinates of *p(i)* in the original system of coordinates are $xop(i,j)=o_j+L(j) \cdot xp(i,j)$ $(1 \le j \le d)$.

VIII. Related Work

Optimal broadcast strategies in trees in the single-port model have been studied in [1,8]. In [2], the problem was enhanced with non uniform edge transmission times and an $O(n \cdot log(n))$ algorithm was proposed. A dynamic programming algorithm was

presented in [3] for the minimum time broadcast in directed trees, under the single port line model. Efficient algorithms for the maximum reliability k-hop multicast strategy in directed trees, as well as exact, exponential algorithms for minimum time multicast in directed graphs have been presented in [4]. File transfer scheduling problems considering tree and clique mutual exclusion graphs were studied in [5].

IX. CONCLUSIONS

In this paper we studied several extensions of the single port model broadcast in trees. The extensions are motivated by practical applications and we presented efficient algorithms for computing optimal broadcast strategies. We also considered communication optimization and scheduling problems on several graphs (trees, intersecting cliques, time graphs), for which we presented exact algorithms which are quite efficient in certain situations.

REFERENCES

[1] P.J. Slater, E.J. Cockayne, and S.T. Hedetniemi, "Information Dissemination in Trees," SIAM J. on Computing, vol. 10, 1981, pp. 692-701.

[2] J. Koh and D. Tcha, "Information Dissemination in Trees with Nonuniform Edge Transmission Times," IEEE Trans. on Computers, vol. 40 (10), 1991, pp. 1174-1177.

[3] B. D. Birchler, A.-H. Esfahanian, and E. K. Torng, "Information Dissemination in Restricted Routing Networks," Proc. of the Intl. Symposium on Combinatorics and Applications, 1996, pp. 33-44.

[4] M. I. Andreica and N. Ţăpuş, "Maximum Reliability K-Hop Multicast Strategy in Tree Networks," Proc. of the IEEE International Symp. on Consumer Electronics, 2008.

[5] M. I. Andreica and N. Ţăpuş, "High Multiplicity Scheduling of File Transfers with Divisible Sizes on Multiple Classes of Paths," Proc. of the IEEE International Symp. on Consumer Electronics, 2008.

[6] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs," SIAM J. on Computing, vol. 2 (4), 1973, pp. 225-231.

[7] A. Rosenthal and J. A. Pino, "A generalized algorithm for centrality problems on trees," J. of the ACM, vol. 36 (2), 1989, pp. 349-361.

[8] J. Cohen, P. Fraginaud, and M. Mitjana, "Polynomial-Time Algorithms for Minimum-Time Broadcast in Trees," Theory Comput. Syst., vol. 35 (6), 2002, pp. 641-665.

[9] D. S. Franzblau, "Optimal Hamiltonian Completions and Path Covers for Trees, and a Reduction to Maximum Flow," ANZIAM Journal, vol. 44, 2002, pp. 193-204.