Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control

Mugurel Ionut Andreica, Nicolae Tapus

Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control

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

Abstract— **In this paper we present Time Slot Groups (TSG), a novel, efficient data structure for QoS-constrained advance bandwidth reservation and admission control. The data structure divides the time horizon into T equally sized time slots and can be used for serving efficiently complex bandwidth reservation requests specifying the duration of the reservation, the minimum required bandwidth, the earliest possible starting time and the latest possible finish time. The data structure supports reservation queries in time O(k+(T/k)·log(k)) and reservation updates in time O(k+(T/k)), where k is a user-defined parameter.**

I. Introduction

In this paper we present a novel, efficient data structure which is used for offering bandwidth guarantees to non-preemptive data transfers on a single network link, subject to time constraints, in the following context: applications submit bandwidth reservation requests to a bandwidth broker which either satisfies the requests or rejects them. The data structure divides the time horizon upon which bandwidth reservations are performed into T discrete equally-sized time slots and supports efficiently the following types of operations: ***find***(s_1,s_2,D,B) - finds a time slot interval $[s,s+D-1]$, where at least a given amount of bandwidth B is available during every time slot of the interval, subject to the following QoS constraints: the length of the interval is D time slots, the earliest possible starting time slot is s_1 and the latest possible finish time slot is s_2 (i.e. $s_1 \le s \le s+D-1 \le s_2$); ***reserve***$(s_1,s_2,B)$ - decreases by B the available bandwidth for each slot within the time slot interval $[s_1,s_2]$ (if the value of B is negative, an increase takes place). The reserve (update) operation takes $O(k+(T/k))$ time and the find (query) operation takes $O(k+(T/k)\cdot log(k))$ time, where $1 \le k \le T$ is a user-defined parameter (e.g. a constant value or a function $f(T)$). Some situations where this functionality is useful are the transfer of multimedia streams to customers who are only available within some specific time intervals or the transfer of large data files in Grids and other distributed systems.

The rest of this paper is organized as follows. In Section II we present related work. In Section III we present an enhanced version of the standard time slot array, which is the building block for the *Time Slot Groups* data structure presented in Section IV. The performance of the data structure is tested in Section V, where we also conclude.

II. Related Work

Many resource reservation and scheduling techniques [5] make use of efficient data structures capable of improving the response time. The simplest one is an array storing the available bandwidth for each time slot, but this takes $O(T)$ time per operation. The segment tree [1] and the bandwidth tree [6] provide a time complexity of $O(log(T))$ per operation, but only for simple requests: for $D=1$ ($D=s_2-s_1+1$) we need a range maximum (minimum) query operation, together with a range addition update. A dynamic version of an augmented segment tree is proposed in [2] and a linked-list data structure is presented in [7]. None of the data structures we mentioned (or that we are aware of) can efficiently support the complex requests our structure does. All of them exhibit linear ($O(T)$) or superlinear ($O(T\cdot log(T))$) time for at least one of the two operations, while our data structure takes sublinear time for both. We note that we incorrectly claimed in [1] that a segment tree or block partition can solve a more relaxed version of this problem.

III. The Enhanced Time Slot Array

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

reserve TSA(s_1, s_2, B):
for $s = s_1$ **to** s_2 **do** $availbw[s] = availbw[s] - B$

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

$$B_{interval} = \min_{sa \le s \le sb} \{availbw[s]\}. \quad (1)$$

In the *find* function we are looking for an interval of D slots, with a bandwidth greater than or equal to B. We will traverse the $[s_1,s_2]$ interval with a sliding window consisting of D time slots and maintain a min-heap with the available bandwidths of

the time slots in the window. When we move the right end of the window one position to the right, from s ($s \geq s_1+D-1$) to $s+1$, we remove from the heap the (leftmost) time slot $s-D+1$ (which now falls outside of the window) and insert into the heap the time slot $s+1$. We can reduce the time complexity from $O(T \cdot log(T))$ to $O(T)$, if we replace the min-heap by a double-ended queue (deque) [4] which stores *(time slot, available bandwidth)* pairs. The elements of the deque are sorted increasingly according both to the bandwidth and the time slot. The first element of the deque is always the one with the minimum bandwidth among the slots inside the current window. As the right end of the window slides to the next slot s, all the pairs at the end of the deque whose bandwidths are larger than the available bandwidth of slot s are removed. The element at the front of the deque is removed when it falls outside of the sliding window. This takes $O(T)$ time, because we insert each time slot once and remove it at most once from the deque.

find_TSA(s_1 , s_2 , D, B):

```
deque = empty
for s = s1 to s2 do
  while ((not deque.isEmpty()) and (deque.getLast().value≥
                                    availbw[s])) do deque.removeLast()
  deque.addLast((time_slot = s, value = availbw[s]))
  if ((s–s1+1>D) and (deque.getFirst().time_slot=s-D)) then
    deque.removeFirst()
  if ((s–s1+1≥D) and (deque.getFirst().value≥B)) then
    return [s-D+1,s]
return "no interval found"
```

The enhanced time slot array handles differently only the query and update function calls referring to all the slots (between $s_1=0$ and $s_2=T-1$). These calls will be named *full-period* calls. A *full-period update* needs to decrease the bandwidth of each time slot by the same value B. Instead of doing this, only the value of a variable called *globalbw* is modified. Thus, the real available bandwidth of each slot s will be $availbw[s]+globalbw$. For each *full-period query*, we will find the answer in $O(1)$ time, by using a previously computed array *sibw*. *sibw[L]* contains the maximum bandwidth of an interval of L slots, as well as the actual interval. The *sibw* array will be computed after every non-full-period update. An efficient way to compute the *sibw* array would be to sort the values of the available bandwidths of the T time slots increasingly into an array called *values*. We will maintain a data structure (balanced tree) of (disjoint) time slot intervals which, initially, contains only one time slot interval consisting of all the T time slots. Then we will traverse the sorted *values* array. Every value will split the time slot interval inside which it is located into two time slot intervals (or one if it is located at the end of some time slot interval, or zero if the time slot interval consisted of just one slot). We will also have a binary max-heap with the length of the current time slot intervals. Using the balanced tree, we can retrieve easily the time slot interval into which a given time slot resides. Before performing a split at the i^{th} value, we will retrieve the maximum value L from the max-heap, meaning that a time slot interval of L time slots having a bandwidth equal to $values[i]$ exists. We will store this interval at the position $sibw[L]$. After performing the split, the time slot interval which was split is removed both from the heap and the balanced tree and will be replaced by the resulting smaller intervals (which are inserted in the tree and the heap). After traversing all of the values (and performing all the splits), we traverse the array *sibw* from the largest length to the smallest one; if no interval was stored for a length L, then we use the time slot interval for length $L+1$, removing from it the leftmost or the rightmost time slot. The overall time complexity is $O(T \cdot log(T))$, because each of the T splits takes $O(log(T))$ time.

We can compute the *sibw* array more efficiently, in $O(T)$ time. If we consider the available bandwidth of a time slot s as the "height" of that time slot, we obtain a histogram. We can find all the $O(T)$ maximal area rectangles inside the histogram in $O(T)$ time, by adapting an algorithm presented in [3] for finding the largest area rectangle full of ones in a binary matrix. The algorithm maintains a stack of *(time slot, bandwidth)* pairs, sorted increasingly both according to the slot number and the bandwidth value. If, after processing a slot s, the stack contains some pair (s',B), then the time slot interval $[s',s]$ has bandwidth B and is the longest interval ending at slot s having this bandwidth. The pseudocode of the (first version of the) functions is given below:

find_ETSA_v_1(s_1 , s_2 , D , B):

```
if (s1=0) and (s2=T-1) then
  if (sibw[D].bw + globalbw ≥ B) then
    return [sibw[D].s1 , sibw[D].s2]
  else return "no interval found"
else return find_TSA(s1 , s2 , D , B-globalbw)
```

reserve_ETSA_v_1(s_1 , s_2 , B):

```
if ((s1=0) and (s2=T-1)) then globalbw=globalbw–B  else
  reserve_TSA(s1 , s2 , B)
  computeSibw()
```

computeSibw():

```
stack=empty; availbw[T]=-∞
sibw[L]=undefined, for each  L=1,…,T
```

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

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

find_ETSA_v2(s_1 , s_2 , D, B):

```
if ((s1 = 0) and (s2 = T-1) and (dirtyFlag.isSet())) then
  computeSibw()
  dirtyFlag.clear()
return find_ETSA_v1(s1 , s2 , D , B)
```

reserve_ETSA_v2(s_1 , s_2 , B):

```
if ((s1 = 0) and (s2 = T-1)) then globalbw = globalbw – B  else
  reserve_TSA(s1 , s2 , B)
  dirtyFlag.set()
```

The enhanced time slot array is augmented with three extra functions, *getMinBw*, *getMinBwLeftToRight* and *getMinBwRightToLeft*, computable in *O(1)* time. *getMinBw* returns the minimum bandwidth of any time slot. The minimum value in the *availbw* array, *minbw*, is computed in the *computeSibw* function (in *O(T)* time). The real minimum bandwidth is the sum of *minbw* and *globalbw*. In the *computeSibw* function we also compute in *O(T)* time two arrays: *minBwLtoR* and *minBwRtoL*, defined as follows:

$$\text{minBwLtoR}[i] = \min_{0 \le j \le i} \{\text{availbw}[j]\}\ , \qquad (2)$$

$$\text{minBwRtoL}[i] = \min_{i \le j < T} \{\text{availbw}[j]\}\ . \qquad (3)$$

The pseudocode of the *getMinBwLeftToRight* function is shown below. We define the other function (*getMinBwRightToLeft*) in a similar manner.

getMinBwLeftToRight(p):

```
if (p<0) then return +∞
else return minBwLtoR[min{p,T-1}]+globalbw
```

IV. The *Time Slot Groups* Data Structure

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

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

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

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

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

reserve_TSG(s_1 , s_2 , B):

compute G_1 , G_2 , sr_1 , sr_2
if $(G_1=G_2)$ **then** $tsg[G_1].reserve\_ETSA\_v_{(1/2)}(sr_1,sr_2,B)$ **else**
$tsg[G_1].reserve\_ETSA\_v_{(1/2)}(sr_1 , k-1 , B)$
$tsg[G_2].reserve\_ETSA\_v_{(1/2)}(0 , sr_2 , B)$
for $G=G_1+1$ **to** G_2-1 **do** $tsg[G].reserve\_ETSA\_v_{(1/2)}(0, k-1 , B)$

For the *find* function, the time slot interval $[s_1,s_2]$ can also have one of the two types of structures presented previously. We will also compute the group numbers of s_1 and s_2 (G_1 and G_2) and the values sr_1 and sr_2. If $G_1=G_2$, we just call the *find* function of G_1, with (sr_1, sr_2, D, B) as parameters. This call takes $O(k)$ time. If $G_1<G_2$, then we will first deal with a particular sub-case. If D is at most k, then the desired slot interval could be completely located within one of the groups $G_1+1,\ldots,G_2-1$. For each such group, we call the *find* function, with parameters $(0, k-1, D, B)$. Each

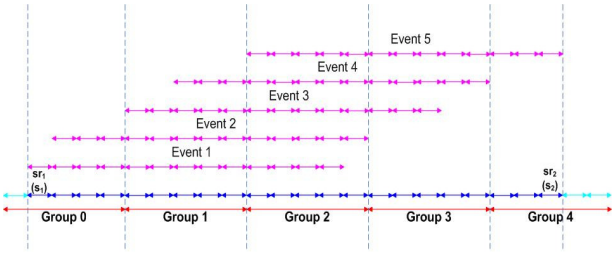

Figure 1. All the events for: T=25, k=5, s_1=1, s_2=22, D=13.

call takes $O(1)$ time. The time complexity of this particular case is $O(T/k)$, as there are $O(T/k)$ groups between G_1 and G_2. The interval could also lie in G_1 (between sr_1 and $k-1$) or G_2 (between 0 and sr_2), if D is at most $k-sr_1$, or at most sr_2+1, respectively. Calling the appropriate functions for G_1 and G_2 takes $O(k)$ time. The case where $D>1$ and the desired slot interval might cross several groups is presented next.

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

The slot interval $[s_1,s_2]$ contains at most $(T/k)-2$ groups between G_1 and G_2. Thus, the number of events is $O(T/k)$. For each event E, we will find the candidate interval having the maximum bandwidth, with the first time slot located between the first slot of E (inclusive) and the first slot of the next event (exclusive). If E is the last event, the next event is considered one slot to the right. Let's assume that for the current event E, the first time slot is in the group G_{begin} and its relative slot number in that group is s_{begin}. Similarly, the ending time slot's group is G_{end} and its relative time slot number in that group is s_{end}. If E is not the last event (i.e. the ending slot of E is not s_2), then we will also assume that s_{end} is not the last time slot in the group G_{end} (if it is, we consider $s_{end}=-1$ in the group $G_{end}+1$, i.e. right before the first time slot of $G_{end}+1$; after this, we also set $G_{end}=G_{end}+1$). If $G_{begin}<G_{end}$, we compute in $O(1)$ time the distance $dist$ (in terms of time slots) between E and the next event (if E is the last event, then $dist=1$). If we slide the event interval by any number of time slots between 0 and $dist-1$, the values of G_{begin} and G_{end} will remain the same (although s_{begin} and s_{end} would increase). Let's call the groups from $G_{begin}+1$ to $G_{end}-1$ *interior groups* and let's assume that we already know their minimum bandwidth B_{groups} (the minimum of the values returned by the *getMinBw* function of each group). We will define $cand(p)$=the candidate interval obtained by sliding the current event interval p positions to the right. For $0\le p\le dist-1$, $cand(p)$ contains all the interior groups of the event interval. Only the intersections with the groups G_{begin} and G_{end} change.

TABLE I. RUNNING TIMES (IN SECONDS): T=262,144 ; K=512(=$T^{1/2}$).

Total # of opera-tions	Number of time slots per query	Number of time slots per update	Time Slot Array	TSG without dirty flag	TSG with dirty flag
10,000	0	1 - 3000	0.13	1.4	0.1
10,000	0	150,000 - 262,144	8.75	1.17	0.1
10,000	10 - 2000	0	0.31	0.29	0.28
10,000	190,000 - 262,144	0	45.18	0.31	1.16
10,000	210,000 - 262,144	0	55.49	0.45	0.5

10,000	10 - 262,144	20,000 - 262,144	18.74	0.62	0.8
Sum of running times :			128.6	4.24	2.94

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

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

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

find_TSG(s_1, s_2, D, B):

compute G_1, G_2, sr_1, sr_2
if $(G_1=G_2)$ **then return** $tsg[G_1].find\_ETSA\_v_{(1/2)}(sr_1,sr_2,D,B)$ **else**
  handle the particular sub-cases: $D \le k$, $D \le k-sr_1$, $D \le sr_2+1$
  $fts=s_1$; $lts=s_1+D-1$; ***compute*** G_{begin}, s_{begin}, G_{end}, s_{end}
  intGDQ=empty
  add to intGDQ the groups G in $[G_{begin}+1, G_{end}-1]$ *(in this order)*
  while *(the last event has not been processed)* **do**
    if $(s_{end}=k-1)$ **then** // move to $G_{end}+1$
      if $(G_{begin}<G_{end})$ **then**
        remove *all the elements from the end of intGDQ having a bandwidth larger than the minimum bandwidth of the group* G_{end}
        $intGDQ.addLast((grp=G_{end}, bw=tsg[G_{end}].getMinBw())$
      $s_{end}=-1$; $G_{end}=G_{end}+1$
    dist=the distance between the current event and the next event
    compute p_{opt} // $\min\{B_{begin}(p_{opt}),B_{end}(p_{opt})\}$ is maximum
    if (**not** *intGDQ.isEmpty()*) **then** $B_{groups}=intGDQ.getFirst().bw$
    else $B_{groups}=+\infty$
    if $((min\{B_{begin}(p_{opt}), B_{end}(p_{opt}), B_{groups}\} \ge B)$ **and** $(G_{begin}<G_{end}))$ **then return** $[fts+p_{opt}, lts+p_{opt}]$
    move *to the next event* (***update*** G_{begin}, G_{end}, s_{begin}, s_{end}, *fts, lts*)
    if ((**not** *intGDQ.isEmpty()*) **and** $(intGDQ.getFirst().grp=G_{begin}))$ **then** $intGDQ.removeFirst()$
  return *"no interval found"*

V. Experimental Results and Conclusions

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

References

[1] 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.

[2] A. Brodnik and A. Nilsson, “An Efficient Data Structure for Advance Bandwidth Reservations on the Internet,” Proc. of the 3rd Conference on Comp. Sci. and Electrical Eng., 2002.

[3] D. Vandevoorde, “The Maximal Rectangle Problem,” Dr. Dobb’s Journal, vol. 23, 1998, pp. 30-32.

[4] P. Berman, et al., “Fast Optimal Genome Tiling with Applications to Microarray Design and Homology Search,” J. Comput. Biol., vol. 11, 2004, pp. 766-785.

[5] L. Marchal, P. V.-B. Primet, Y. Robert, and J. Zeng, “Optimizing Network Resource Sharing in Grids,” Proc. of the 48th IEEE Global Telecomm. Conf., IEEE Press, 2005, pp. 123-132.

[6] T. Wang and J. Chen, “Bandwidth Tree - a Data Structure for Routing in Networks with Advanced Reservations,” Proc. of the 21st Intl. Perf., Computing, and Comm. Conf., IEEE Press, 2002, pp. 37-44.

[7] Q. Xiong, C. Wu, J. Xing, L. Wu, and H. Zhang, “A Linked-List Data Structure for Advance Reservation Admission Control,” Lecture Notes in Computer Science, vol. 3619, 2005, pp. 901-910.