Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control
Mugurel Ionut Andreica, Nicolae Tapus

To cite this version:

HAL Id: hal-00789151
https://hal.archives-ouvertes.fr/hal-00789151
Submitted on 3 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control

Mugurel Ionuț Andreica, Nicolae Tăpuș
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 \(O(k+(T/k) \cdot \log(k))\) 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: \(\text{find}(s_1, s_2, D, B)\) - 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. \(s_1 \leq s \leq s+D-1 \leq s_2\)); \(\text{reserve}(s_1, s_2, B)\) - 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 \(1 \leq k \leq T\) 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), \(\text{availbw}\), 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 \(\text{reserve}\) function below:

\[
\text{reserve}_{\text{TSA}}(s_1, s_2, B) \quad \text{for } s = s_1, \ldots, s_2 \text{ do availbw}[s] = \text{availbw}[s] - B
\]

The bandwidth of a time slot interval \([s_a, s_b]\) is:

\[
B_{\text{interval}} = \min_{s \in [s_a, s_b]} \{\text{availbw}[s]\}
\]

In the \(\text{find}\) 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 \( s \) \((\leq s_2+D-1)\) 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 \( O(T \log(T)) \) 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**\((s_1, s_2, D, B)\):

```java
deque = empty
for s = s_1 to s_2 do
  while (!deque.isEmpty() and deque.getLast().value >= availbw[s]) do deque.removeLast()
  if (s-s_1+1>D) and (deque.getFirst().time_slot==s-D) then
deqe.removeFirst()
if (s-s_1+1>2D) 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 \( i^\text{th} \) 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 \( O(T \log(T)) \), 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 ETA**\(\_V \_2\)\((s_1, s_2, D, B)\):

```java
if (s_1=0) and (s_2=T-1) then
  if (sibw[D].bw + globalbw >= B) then
    return [sibw[D], s_1, sibw[D], s_2]
  else return "no interval found"
else return find_TSA(s_1, s_2, D, B-globalbw)
```

**reserve ETA**\(\_V \_2\)\((s_1, s_2, D)\):

```java
if ((s_1=0) and (s_2=T-1)) then globalbw = globalbw-B else
  reserve_TSA(s_1, s_2, B)
computeSibw()
```

**computeSibw()**:

```java
stack=empty; availbw[T]=∞
sibw[L]=undefined, for each \( L=1,\ldots,T \)
```
for $s=0$ to $T$

$L\leftarrow s$
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].s = stack.top().leftmost_slot
        sibw[L].s = s
        $lslot=stack.top().leftmost_slot; stack.pop()$
        stack.push((leftmost_slot = lslot, bw=availbw[s]))

for $L=T-1$ downto $0$ do

    if (sibw[L] = undefined) then
        sibw[L].s = sibw[L+1].s; $sibw[L].bw=sibw[L+1].bw$
        $sibw[L].s=sibw[L+1].s-1$
    compute minbw, minBwLtoR and minBwRtoL.

Because by using ETSA_v, 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_vy) is shown below:

**find ETSA_vy(s, s₂, D, B):**

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

**reserve ETSA_vy(s₁, s₂, D, B):**

if ($(s₁ = 0)$ and $(s₂ = T-1)$) then $globalbw = globalbw - B$
else
    reserve TSG(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:

$$
\text{minBwLtoR}[i] = \min\{\text{availbw}[j] \mid \text{minBwLtoR}[i] \}
$$

$$
\text{minBwRtoL}[i] = \min\{\text{availbw}[j] \mid \text{minBwRtoL}[i] \}
$$

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 $+\infty$
else return $\text{minBwLtoR}[\text{min}[p, T-1]] + globalbw$

IV. THE TIME SLOT GROUPS DATA STRUCTURE

We divide the $T$ time slots into $n_g = O(T/k)$ 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 $n_g-1$. Group $g$ contains the slots numbered from $(i-1)k$ 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 $[s_i, s_j]$ of the reserve function can have one of the two types of structures:

- **Type A:** $s_j$ and $s_j$ lie inside the same group $G_i$.
- **Type B:** $s_j$ is located inside some group $G_i$ and $s_j$ is located inside a group $G_{j+1}$.

First, we will compute the group numbers ($G_j$ and $G_{j+1}$) and $s_{r_j}$ and $s_{r_{j+1}}$, the values of the time slots $s_j$ and $s_{j+1}$, relative to their groups:

$$
G_i = \left\lfloor \frac{s_j}{k} \right\rfloor \cdot x_k = s_j \mod k \quad (i = 1, 2).
$$

In the case of a Type A structure, we will simply call the reserve function of the group $G$ with parameters ($s_{r_{j+1}}$, $s_{r_{j+2}}$, $B$), which will be executed in $O(k)$ time. For a Type B structure, we will update the interval of time slots $[s_{r_j}, k-1]$ in $G_i$, the interval $[0, s_r]$ in $G_i$ and the intervals $[0, k-1]$ for each group between $G_i$ and $G_{j+1}$. Updating $G_i$ and $G_{j+1}$ takes $O(k)$ time, while updating the other $O(T/k)$ groups takes $O(1)$ time for each group. The overall complexity is $O(k+T/k)$.

**reserve TSG(s₁, s₂, B):**
compute $G_1$, $G_2$, $s_{r1}$, $s_{r2}$
if ($G_1$ = $G_2$) then $tg[G_1$.reserve_ETSA($sr_1$, $sr_2$, $B$) else
$tg[G_2$.reserve_ETSA($sr_1$, $k-1$, $B$)
for $G=G_1+1$ to $G_2-1$ do $tg[G$.reserve_ETSA($0$, $sr_2$, $B$)

For the find function, the time slot interval $[s_{r1}, s_{r2}]$ can also have one of the two types of structures presented previously. We will also compute the group numbers of $s_{r1}$ and $s_{r2}$ ($G_1$ 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 $G_1$, $G_2$. If for each such group, we call the find function, with parameters $(0$, $k-1$, $D$, $B$). Each call takes $O(1)$ time. The time complexity of this particular case is $O(T/k)$, as there are $O(T/k)$ groups between $G_1$ and $G_2$. The interval could also lie in $G_1$ (between $sr_1$ and $k-1$) or $G_2$ (between $0$ and $sr_2$), if $D$ is at most $k-sr_1$, or at most $sr_2+1$, respectively. Calling the appropriate functions for $G_1$ and $G_2$ takes $O(k)$ time. The case where $D > k$ 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 $[s_{r1}, s_{r2}]$. There are $s_{r1}$-$D+2$ = $O(T)$ candidate intervals, one for each possible starting slot. We will traverse the $[s_{r1}, s_{r2}]$ 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 $\{[s_{begin}, s_{end}] \mid [s_{begin}, s_{end}] \subseteq [s_{r1}, s_{r2}] \text{ and } (s_{begin}=s_{r1} \text{ or } s_{end}=\text{the first time slot of some group} \text{ or } s_{begin}=s_{r2} \text{ or } s_{end}=\text{the last time slot of some group}) \text{ and } (s_{end}=s_{r2}+D-1)\}$. The events can be sorted from left to right, according to their leftmost time slot (see Fig. 1).

The slot interval $[s_{r1}, s_{r2}]$ 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 $s_{r2}$), then we will also assume that $s_{end}$ is not the last time slot in the group $G_{end}$ (if it is, we consider $s_{end}$ as $-1$ in the group $G_{end}+1$). Thus, the first time slot of $G_{end}$ is $1$, after this, we set $G_{begin}$ as $G_{begin}+1$. If $G_{begin}$ < $G_{end}$, we compute in $O(1)$ time the distance $\text{dist}$ (in terms of time slots) between $E$ and the next event (if $E$ is the last event, then $\text{dist} = 1$). If we slide the event interval by any number of time slots between $0$ and $\text{dist} - 1$, 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 $G_{begin}$ to $G_{end}$ interior groups and let’s assume that we already know their minimum bandwidth $\text{getMinBw}$ function of each group. We will define $\text{cond}(p)$ the candidate interval obtained by sliding the current event interval $p$ positions to the right. For $\text{dist} = 1$, $\text{cond}(p)$ 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).

<table>
<thead>
<tr>
<th>Total # of operations</th>
<th>Number of time slots per query</th>
<th>Number of time slots per update</th>
<th>Time Slot Array</th>
<th>TSG without dirty flag</th>
<th>TSG with dirty flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0</td>
<td>1 - 3000</td>
<td>0.13</td>
<td>1.4</td>
<td>0.1</td>
</tr>
<tr>
<td>10,000</td>
<td>0</td>
<td>150,000 - 262,144</td>
<td>8.75</td>
<td>1.17</td>
<td>0.1</td>
</tr>
<tr>
<td>10,000</td>
<td>10 - 2000</td>
<td>0</td>
<td>0.31</td>
<td>0.29</td>
<td>0.28</td>
</tr>
<tr>
<td>10,000</td>
<td>190,000 - 262,144</td>
<td>0</td>
<td>45.18</td>
<td>0.31</td>
<td>1.16</td>
</tr>
<tr>
<td>10,000</td>
<td>210,000 - 262,144</td>
<td>0</td>
<td>55.49</td>
<td>0.45</td>
<td>0.5</td>
</tr>
</tbody>
</table>
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.

We define \( B_{\text{beg}}(p) \) as the minimum bandwidth among the time slots located in the intersection of \( cand(p) \) and \( G_{\text{beg}} \). \( B_{\text{beg}}(p) \) is computed as \( \text{getMinBwRightToLeft}(s_{\text{beg}}+p) \) (in \( O(1) \) time), called for the group \( G_{\text{beg}} \). Analogously, we define \( B_{\text{end}}(p) \) as the minimum bandwidth among the time slots located in the intersection of \( cand(p) \) with the group \( G_{\text{end}} \). \( B_{\text{end}}(p) = s_{\text{end}} + p \) for all the updates and \( s_{\text{end}} + p \) for queries, where \( p \) is the highest value.

The value of \( p \) can be found using a simple binary search. Since \( dist \leq k \), the binary search takes \( O(\log(k)) \) time. Then, the value \( \max \{ \min \{ B_{\text{beg}}(p), B_{\text{end}}(p) \} \} \) is found either for \( p=\text{pw} \) or for \( p=\text{pw}+k \).

Once the optimal value for \( p \) is found, \( B_{\text{groups}} \) compute the bandwidth of the candidate interval determined, which is \( \min \{ B_{\text{beg}}(p), B_{\text{end}}(p) \} \) and compare it to the required bandwidth. In order to efficiently find the value \( B_{\text{groups}} \) 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 \( G_{\text{end}} \) may become an interior group and the leftmost interior group may fall outside of the interval of interior groups. Thus, \( B_{\text{groups}} \) can be computed in \( O(1) \) time for each event. The time complexity of the \( \text{find} \) function is dominated by the computation of the best candidate intervals and is of the order \( O(T/k \log(k)+k) \). For the case of simple requests (i.e. \( D=1 \) or \( D=s_{x}-s_{x}+1 \)), the complexity of the query operation is only \( O(k+T/k) \).

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 \( -B_{\text{min}} \) and \( +B_{\text{max}} \) (the initial value of the available bandwidth of each time slot). For all the queries, \( D \) was at most 75% of the length of \( [s_{x}, s_{x}] \) and \( B \) was between 0 and \( B_{\text{max}} \). 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 \( O(k+T/k \log(k)) \) 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


