Analysis of Data Structures for Admission Control of Advance Reservation Requests

Lars-Olof Burchard (baron@cs.tu-berlin.de)

Communication and Operating Systems Group
Technische Universitaet Berlin

Technical Report KBS-2004/1

Abstract
Advance reservations are a useful method to allocate resources of various kinds in many different environments. Among others, a major advantage of this kind of reservations is the improved admission probability for requests that are made sufficiently early. Applications can be found in many different environments, e.g., the field of grid computing which allows to co-allocate compute resources in different locations connected over the Internet. For that purpose, co-allocations in advance of resources, such as compute nodes or network bandwidth, are important to guarantee the correct functionality of the whole system, e.g., that a certain job is completed up to given deadline. In order to implement reliable admission control in those cases, it is important to store information about future allocations and to provide fast access to the available information. It will be shown that data structures for that purpose demand for the largest portion of the overall admission processing time and therefore are required to be fast to access and also memory efficient. In this paper, two data structures are examined. One of them is a tree specially designed to support advance reservations which is compared to arrays. Both structures are examined both analytically and by measurements in a realistic network management system. It turns out that arrays are far better suited to support the required operations, in particular when large time intervals need to be scanned.

1 Introduction

In contrast to immediate reservations, where the allocations are established immediately after the request is admitted, advance reservations allow to specify and request a given quality-of-service (QoS) for a resource a long time before the actual usage starts. Advance reservations are especially useful in environments that require reliable allocations of various resource types at different locations. Such co-allocations are necessary in order to assure that all the resources required are available at a given time.

Grid computing toolkits [9,10] support co-allocations based on advance reservation mechanisms and are used to build an infrastructure with the potential to
manage resources on a global scale. An example for such kind of co-allocation is given in Fig. 1: data received via satellite is transmitted to two cluster computers (step 3) where the data is preprocessed resulting in a real-time data stream from both sources which is then multiplexed and transmitted to the final destination where it is post-processed and shown on a video screen. In this example, it is required to make allocations for any resource involved, i.e., the satellite link, the cluster computers, the visualization component and the network connections between the different locations. The network connections must be reserved for both real-time transmissions (video stream) and the transfer of large data files from the satellite receiver to the cluster computers.

Especially the support for advance reservations on the network is essential to avoid delays or insufficient bandwidth for the transmissions which may impact the correct operation of the whole system. For this purpose, reservation agents [17] or bandwidth brokers [2,14] were proposed to manage network resources and to support advance reservations.

Independent of the actual application environment, e.g., cluster, grid, or network management, resource management systems are required to have the capability to process large amounts of requests at a time (in the order of several thousands of requests or more) in order to avoid delays in the reservation process for users. For example, network management systems are designed to handle around several thousands of simultaneous requests. Besides providing fast end-to-end response times, where also other factors such as message run-time are important, the main problem in this context is to provide high throughput in order to not keep clients waiting and to prevent the management from being overloaded. In particular network environments, where the management may reside on routers with naturally lower processing capabilities, require to address this issue.

One important task to be performed by a resource management in such an environment is admission control which means for new reservation requests it must be checked whether sufficient resources are available for the whole duration of the reservation. For this purpose, the aggregated resource usage is stored in data structures (see Fig. 2). In the advance reservation environment, the
checks must be made not only for a single point in time but for the duration of a reservation or a failure. Hence, the data structures must provide fast and efficient access to information about the resource utilization. The temporal dimension of this data structure related processing defines the difference between immediate and advance reservations and, as will be shown in this paper, in a sample network management system the time spent in the data structures adds up to about 60% of the total processing time required by the bandwidth broker, whereas routing requires only about 8% and the administrative tasks the remaining 32% of the total processing time. This holds only for the most simple reservation type. Requests that make use of the full potential of the advance reservation service, e.g., by allowing the network management to choose suitable transmission parameters, require even more time. Therefore, optimizations of these data structures are most likely to improve the processing speed of the bandwidth broker.

In this paper, two data structures that can be used in the given environment are compared. The first structure is a tree which was especially designed to support the admission control task in advance reservation environments. It is compared to arrays using both analysis and measurements. The comparison includes two aspects: the speed of the admission control process when using the data structures, and the memory efficiency, i.e., the amount of memory required to store reservation requests. The interesting result is that arrays are not only easier to implement but also have a significant performance advantage compared to the tree structure. This is not a result of hardware aspects, e.g., caching, but can also be shown in an analysis of the properties of both data structures.

Measurements using an actual network management system (bandwidth broker) with a realistic network topology show, that the usage of arrays leads to even faster admission times than determined by the analysis. In this paper, the focus is on the network environment since it is more challenging in the sense that many different links are involved for each of which advance reservations must be performed, i.e., a data structure is required for each of the links. However, the results presented here can be applied to any other environment supporting advance reservations, in particular the allocation of compute nodes in grid scenarios.

The rest of this document is organized as follows: firstly, after a description of the application environment, both data structures are introduced and their properties are described. This covers also aspects of actual implementations. Following that, the performance is analyzed using the average amount of
memory cells accessed during admission control as a metric to assess the perfor-
mance. The third part deals with actual performance measurements made with
real implementations of both data structures in an actual advance reservation
enabled network resource management system. The paper is concluded with a
discussion of related work and a summary of the results.

2 Application Environment

2.1 Advance Reservations

In the environment examined here, the management system needs to fulfill sev-
eral tasks related to admission control. These tasks include routing, authenti-
cation, and other management tasks such as employing recovery mechanisms in
case of resource outages or failures. In this context, the difference between an
immediate and an advance reservation environment is the check for sufficient
bandwidth during the whole requested transmission period which is the task of
the data structures examined in this paper.

![Diagram of slotted time and book-ahead interval](image)

Figure 3: Slotted time and book-ahead interval.

Before the data structures are described, some preliminary considerations
must be made concerning the environment in which the data structures are
implemented. Each data structure covers a certain period of time for which
requests can be issued, called book-ahead interval. For each point in time within
this period, the data structure keeps the amount of resources allocated by ad-
mitted reservations (see Fig. 3). In general, two implementation alternatives
exist to store this information. It is either possible to define time slots of equal
length such that requests can only be made for a certain number of consecu-
tive time slots (see Fig. 3), or each request defines its own time scale. This
approach has the advantage of restricting the amount of data that must stored,
i.e., the memory consumption is bounded, and furthermore, it can be easily
implemented. The majority of current implementations in the field of advance
reservations supports this scheme [5, 9, 12, 16]. The other alternative is to al-
low each request to specify its own time scale and to store the information for
any single request [18]. This approach however is not widely used and has the
disadvantage that the memory requirement depends on the amount of requests
stored by the data structure. Consequently, the data structures presented here were designed to support slotted time.

2.2 Importance of the Data Structures

Besides access to the data structures also other tasks must be fulfilled by the management system in order to perform admission control. However, the data structure related processing is important for a variety of other reasons. In general, short admission times are needed as they minimize the response time for clients. The response time of the management system to an individual client also depends on various other factors, e.g., the message run-time. However, this affects only the end-to-end processing time for a single request. Since typical management systems in the cluster or network environment are designed to deal with several thousand requests, e.g., 10,000 simultaneous requests can be handled by the management system described in [15], the throughput of the management system in terms of processed requests is also an important figure and depends to a large extend on the processing efficiency of the data structures as will be shown in Sec. 7.3. This is especially important when the bandwidth broker runs on a device with limited processing capability such as a router.

Furthermore, request types such as *malleable reservations* [2], which involve a possibly large number of scans of long time intervals in order to determine suitable transmission parameters, require a drastically increased amount of processing time spent in the data structures. Since such reservation types make up the full strength of advance reservations, they are likely to comprise a considerable amount of the overall requests and therefore are important to be considered also in terms of admission speed.

Due to their nature, failure recovery mechanisms demand for fast response times in order to reduce packet loss and performance degradation of admitted flows. Within this context, it is required to switch flows affected by a failure onto alternative paths [4], which can also be observed as admission control task and thus, requires accessing the data structures. Since failure recovery is made completely within the broker, no other task such as authentication or the request message run-time affects the processing speed in this case.

Also other off-line tasks, i.e., route changes in the background in order to increase the resource utilization, can result in network performance gains but may require also significant computing time. Therefore, the time spent in this process should be minimized. Moreover, any off-line mechanism, failure recovery or optimization, requires to block the on-line admission control and vice-versa in order to assure consistency of the stored information. Therefore, the time spent in the respective processes needs to be as short as possible in order to ensure both on-line or off-line processes remain operable.

Integrated resource management systems (RMS) for grid computing as described in [6] perform the admissibility checks on behalf of several underlying RMS. In particular when several rounds of admission control must be performed, in those cases the overhead for message run-time and authentication between different RMS is omitted and thus, the response time of the whole system is dominated by the processing time related to the data structures.

As will be shown in Sec. 7.3, the access to the data structures comprises the major amount of the processing time spent in the broker, in particular when scanning larger time intervals where the data structure related processing rep-
resents up to 95 percent of the overall processing time. Therefore, in order to minimize the processing time, improvements of the data structures are most likely affecting the overall processing time. In addition to the admission control tasks, in particular failure recovery and also some off-line optimization strategies, although not explicitly examined in this paper, are also making extensive use of the information stored in the data structures. These tasks can also be observed as admission control processes and therefore benefit in a similar way from the performance improvements of these parts.

2.3 Admission Control

In order to perform the basic admission control process, two operations must be supported. Before a reservation can be made, it is required to check each link on the path for sufficient bandwidth. Therefore, admission control involves a two-phase procedure:

- **CHECK**: Determine whether a reservation request can be fulfilled.
- **UPDATE**: The data structures for each link involved in a transmission are filled with the new values, i.e., the resource requirement of the newly admitted request is added to the existing entries of the data structure. The **UPDATE** phase is only performed in case the previous **CHECK** phase succeeded.

The application of the data structures examined in this paper is a network management system (bandwidth broker). The bandwidth broker has the purpose to process requests for network bandwidth (which is the resource to be managed in this case) and to determine a feasible path, i.e., a path with sufficient bandwidth in the requested transmission interval. In this process, admission control is implemented using the data structures mentioned before. In addition, in case of link failures, it is necessary to find alternative paths for transmissions affected by a failure. In this sense, the data structures are also a very important aspect of the system's performance.

3 Array

Using the slotted time model as assumed throughout this document, the application of arrays is straightforward. Each entry of the array represents a single time slot and stores the accumulated resource usage - in our case: **link bandwidth** - during the corresponding time interval. In order to limit the book-ahead interval, the array is implemented as a ring buffer using a pointer to the time slot representing the current time in the bandwidth broker (see Fig. 4). This allows to handle the advancing time in an elegant and efficient way.

The memory requirement of an array in this form is \( b \cdot s \), with \( b \) being the length of the book-ahead interval and \( s \) denoting the memory requirement for a single slot.
4 Segment Tree

In [16], a tree structure was developed for supporting the admission control task in advance reservation environments, the so-called segment tree which is described in the following.

A segment tree is essentially a complete binary tree with nodes representing the allocated bandwidth during certain time intervals. Each node of the binary segment tree represents one part of the overall book-ahead interval and stores the bandwidth reserved exactly for that part. Both start and stop time of the respective interval of a given node are stored within the node together with two values related to the reserved bandwidth. The first value (denoted by node value) is the bandwidth reserved during the particular period of time the node stands for. The second value (max value) denotes the maximum sum of node values in any of the subtrees below the current node (see Fig. 5). Starting at the top node, the two child nodes each represent one half of the duration of their father node. In order to perform admission control, each reservation request is processed as follows:

1. Starting at the top node, each reservation whose period is covered by the current node, "falls" through to the next level below the current node, i.e., the admission procedure is performed with the node at the level below which covers the particular period of the reservation.

2. If the duration of the request intersects with the duration of more than one node on the current level, the reservation is split at the intersection points and then each of the parts "falls" through, i.e., for each part the admission decision is made independently.

3. If a reservation or a part of it completely fits into the duration covered by a tree node (such a node is called final node), the "fall-through" process stops for the respective part or the whole reservation.

While the reservation or the parts of it "fall through" the tree, the node values of each visited tree node are added up. This happens also for final nodes. When "fall-through" process is finished, i.e., all final nodes are found and completely cover the duration of a part of the original reservation, it is checked whether the requested bandwidth together with the computed sum does not exceed the link bandwidth. In case the bandwidth is sufficient, the request can be admitted. This has to be checked for each path from the top node to the final nodes. In case sufficient bandwidth is available for any part of the reservation, the node

---

1 The final nodes are not necessarily leaf nodes.
values in the corresponding final nodes can be updated. The traversal process can be stopped in case insufficient bandwidth is detected at any of the visited nodes. Following a successful check, each path has to be traversed backwards in order to update the max values in each node previously visited. Fig. 5 shows a segment tree before and after a new request (bandwidth = 50, starting slot = 1, finishing slot = 6) is inserted, final nodes are denoted in gray.

4.1 Details of the Implementation

The application environment considered here is a network management system. In this context, it is necessary to find a suitable path between two network endpoints. Hence, in order to admit a given request, it is required to perform \texttt{CHECK} and \texttt{UPDATE} on any link of a given path.

In general, two alternatives are conceivable in order to implement the segment tree:

1. pointer based implementation
2. tree embedded in an array

Pointers - although suggested in [16] - are rather unsuited in this particular case because the segment tree represents a complete binary tree and hence embedding the tree in an array saves the memory space required for the pointers. On the other hand, a pointer based implementation allows one to use dynamic memory allocation of tree nodes during run-time as will be described in Sec. 4.2.
Figure 6: Tree embedded in array: root node is placed at array index 1.

Tree traversal as required by the **CHECK** and the **UPDATE** operation require to compute the respective array indices for father and child nodes. This can be supported by embedding the tree in the array starting with the root node of the tree at element with index 1 instead of index 0 (see Fig. 6. This allows to ascend or descend in the tree using only one shift operation (instead of multiplying or dividing by 2) and at most one addition. This means, when \( n \) is the array index of the currently visited node, the index of the father node is computed as \( n/2 = (n \gg 1) \) (where \( \gg \) denotes the shift right operation) and the left and right child nodes are computed as \( n \cdot 2 = (n \ll 1) \) and \( n \cdot 2 + 1 = (n \ll 1) + 1 \), respectively.

Furthermore, two other alternatives exist to implement the tree traversal: recursive and nonrecursive descend and ascend operations. However, recursions are costly (although the computational complexity is not affected) in practical implementations due to the computational overhead required for function calls and hence, this alternative is unsuited. Therefore, nonrecursive tree traversal was used for the following examinations. The complete admission control procedure requires at most one descend (**CHECK**) to the final nodes and one ascend (**UPDATE**) from the final nodes in the tree.

Figure 7: **CHECK** and **UPDATE** phase for segment trees, gray boxes denoting visited nodes. The final nodes (1, 5) for each descend/ascend path must be stored during **CHECK**.

Using the nonrecursive tree descend approach requires to keep those nodes where the left and the right subtree must be checked in a separate data structure ("split" nodes). For example, the root node 0 in the situation depicted in Fig. 7 needs to be stored like this. Since the number of those "split" nodes is limited by the total number of nodes visited during the traversal, it is feasible to store these nodes in an array. In the given network scenario, the **CHECK** phase must be performed for any link on a given path. This means, it is required to store the final nodes, respectively their indices in the array the tree is embedded in (see Fig. 7) after each successful **CHECK** phase in order to support efficient retrieval of those nodes during the **UPDATE** phase. For this purpose, final nodes are also stored in an array during the **CHECK** phase.

These considerations lead to the following implementation of the **CHECK** phase (tree descend). Two additional arrays are required, i.e., the array to store final nodes and one to temporarily store "split" nodes:
1. Descend to the next lower level in the tree, checking each node for sufficient bandwidth as described in Sec. 4.

2. In case a node is found where left and right subtree must be checked, insert this node’s index in the array used to store "split" nodes at the rightmost position. Then, scan the left subtree of this node.

3. In case a final node is found during the descend process, i.e., no further descend is required, insert this node’s index in the corresponding array storing final nodes. Remove the leftmost index from the array storing "split" nodes (with right subtree still to be checked) and restart descending the right subtree of this node. If this array is empty, the check is over. The index of each final node visited during the whole process is stored in another array. These indices are later used to start the UPDATE phase from.

4. In case, insufficient bandwidth is determined at any node visited during the descend process, the CHECK phase can be stopped immediately.

In a network management system as considered here, the CHECK phase has to be successful not only for a single tree but for the whole number of trees associated with the links on a given path. This means, the CHECK phase for the whole admission control process is successful only if sufficient bandwidth is available on all links on a given path.

Following a successful CHECK phase, the UPDATE process for each link is initiated, starting from the final nodes which were stored in an array during the CHECK phase. During this process, the node value of the final nodes and the max value of any other node is updated.

4.2 Dynamic Memory Allocation

One of the major drawbacks of the segment tree is the memory consumption. Although the memory consumption of the original implementation [16] can be reduced by not storing start and stop times with each node (instead they can be computed during tree traversal) which was the approach for the implementations presented here, the memory requirement is far worse than that of arrays.

Starting from the observation that only a limited number of tree nodes is actually required even in case many reservations are stored within the tree, the memory requirement can be reduced by only allocating the nodes that are actually required. This is done dynamically during run-time. At the beginning, only the root node is present. In case a new reservation is added, missing tree nodes are generated on-demand during the tree traversal.

In order to realize the dynamic memory allocation, the pointer-based implementation must be used.

5 Advancing Book-Ahead Interval

A drawback of the segment tree is its unsuitability for a dynamically advancing book-ahead interval. This means, that with advancing time but unchanged duration of the book-ahead interval, a new memory cell must be used for each new time slot.
In order to cope with that problem, two trees can be used as depicted in Fig. 8. At the time the book-ahead window completely covers one tree, the content of the other tree can be completely deleted or remain in place for re-usage in order to implement a mechanism similar to the ring buffer. However, in the latter case the node value and max value of each node in the tree must be reset to zero, which may require more time than the complete deletion. For the implementations described here, complete deletion was used. In any case, the worst-case memory consumption doubles. In addition to that, in case a single reservation spans more than one tree this does also increase the admission time since reservations must be split and an admission decision has to be made in both trees.

In contrast, the ring buffer implementation using arrays (see Fig. 4) does neither affect the admission speed nor the memory requirement of the array.

6 Performance Analysis

In order to analyze the behavior and performance of both data structures, it is of interest to determine the number of memory cells \( a(n) \) accessed during a single round of CHECK and UPDATE, with \( n \) being the duration of the corresponding request.

6.1 Array

The memory requirement of arrays is fixed and equals the size of the book-ahead interval counted in slots. The number of memory cells accessed during the CHECK and the UPDATE phase only depends on \( n \) and can be easily computed as

\[
a_{array}(n) = 2n.
\]  

6.2 Segment Trees

While arrays are easy to analyze, this gets more complicated in case of the segment tree. For a given request of duration \( n \) and book-ahead interval \( b \), \( a_{tree}(b, n) \) represents the average number of memory cells accessed during both phases. The average is computed over any possible start time of the request.

Figure 8: Usage of two trees for coping with the dynamically advancing book-ahead interval.
within the book-ahead interval $b$. Unlike arrays, in this case $a_{\text{tree}}$ also depends on $b$.

The approach for computing $a_{\text{tree}}(b, n)$ is to determine the average amount of final nodes and their length, i.e., the duration each final node covers, accessed during both phases. With this knowledge, it is possible to determine the average depth of each final node in the tree which is then used to determine the average amount of memory cells $a_{\text{tree}}(b, n)$.

In the following, the estimations will be made under the assumption that the start time of the reservation is uniformly distributed in the interval $[0, b]$, with $b$ being the book-ahead interval. This means, requests appear with the same probability at each position in the interval $[0, b]$. The general approach in this analysis is to determine for each such position in the tree the number of final nodes (see Sec. 4) and thus the total amount of final nodes $f$. Then, the average duration $l$ covered by each final node within the tree and the average amount of final nodes $\hat{f}$ for each position is computed. With these results, the distance of those final nodes from the root node in the segment tree can be determined which provides the number of nodes and hence memory cells that need to be accessed during the CHECK and the UPDATE phases.

![Diagram showing final nodes and possible positions for a request of duration 2 in a tree with $b = 4$.](image)

Figure 9: Final nodes (gray) and possible positions for a request of duration 2 in a tree with $b = 4$.

The following example illustrates the approach. For the situation given in Fig. 9, with $n = 2$ and $b = 4$ there is a total of 3 different positions of the request of duration 2 in the tree. There are in total 4 final nodes at the three positions. In order to reach a final node during the CHECK phase, 2 memory cells (node value and max value) are accessed for each node on the path. Then, each final node is stored in an array and recovered in the UPDATE phase which starts at those final nodes. During the UPDATE phase, only 1 memory cell is accessed per node, i.e., the node value at the final nodes and the max value at any other node.

The means, for position 0, the CHECK phase requires to access $2 \cdot 2 + 1$ memory cells and $1 + 2 \cdot 1$ memory cells during the UPDATE phase (see Fig. 10). The same holds for position 2. For position 1, for each of the two final nodes $32 + 1$ cells are access during CHECK and $1 + 3 \cdot 1$ cells during UPDATE. In total, this means 8 cells are accessed for position 0 and 2, respectively, and $2 \cdot 11 = 22$ cells for position 1. Thus, the average amount of memory cells accessed for CHECK and UPDATE at a single position is determined as $(8 + 8 + 22)/3 \approx 12.6667$.

In order to generalize this approach, firstly the total number of final nodes $f(n)$ as a function of the duration $n$ of a given request for all possible positions
Figure 10: Cells accessed during CHECK and UPDATE phase in position 0. The final node (black frame) is stored in an extra array.

is determined. As depicted in Fig. 9, the number of final nodes for a request of duration \( n \) depends on the actual position in the tree, i.e., the start time of the request. W.l.o.g., only the number of different positions is considered for this estimation. This means, for the example shown in Fig. 9 only position 0 and 1 would have been considered. This estimation can be made when \( n \) is much smaller than \( b \), which is a realistic assumption. The number of different positions depends on the duration of the requests, i.e., for a request of duration \( n \) there exist \( 2^{\lceil \log n \rceil} \) different positions until the initial position is reached again. In the following, it is assumed that \( n \ll b \), where \( b = 2^k \) denotes the book-ahead interval.

**Lemma 1 (Number of Final Nodes)** The total number of final nodes for a given duration \( n \) can be determined as

\[
f(n) = 1 + \sum_{i=0}^{\lfloor \log n \rfloor - 1} (i + 1)2^i + (n - 2^{\lfloor \log n \rfloor})
\]

\[
= 1 + 2^{\lceil \log n \rceil} \lfloor \log n \rfloor + n - 2^{\lfloor \log n \rfloor}
\]

**Proof.** (of Lemma 1) The formula will be proven in two steps: the first step is to prove the validity for \( n = 2^k \) for all \( k \in \mathbb{N} \). In the second step, the result will be generalized for all \( n \in \mathbb{N} \).

Step 1: (complete induction over \( k \)). In case \( n = 2^k \) for \( k \in \mathbb{N} \), the formula can be simplified to

\[
f(n) = 1 + \sum_{i=0}^{k-1} (i + 1)2^i = 1 + nk.
\]

For \( k = 1 \) follows \( f(2^1) = 3 \) and hence correctness of the formula (see also Fig. 9).

Let now be \( n = 2^{k+1} \). Compared to a duration of \( 2^k \) there are \( 2^k \) additional positions for the reservation in the tree until the initial position is reached again. For the other \( 2^k \) positions, the number of final nodes remains the same as for the case \( n = 2^k \), i.e., \( 1 + nk \), and only the depth in the tree of the respective final nodes changes.
Situation 1: only the tree depth of final node changes

Situation 2: "splitting" leftmost final node

Figure 11: Example: Node changes and new final nodes when the duration is doubled \( 2^k \rightarrow 2^{k+1} \).

The \( 2^k \) new positions can be constructed from the old positions by taking the position with the maximum number of final nodes, splitting the leftmost respectively rightmost final node (with duration of at least 2) and adding the split part to the rightmost respectively leftmost node. This is depicted in Fig. 11. For each of those \( 2^k \) new positions, there are \( \log 2^k + 1 = k + 1 \) nodes, and therefore

\[
f(2^{k+1}) = 2^k(k+1) + 1 + \sum_{i=0}^{k-1} 2^i(i+1) = 1 + \sum_{i=0}^{k} 2^i(i+1) = 1 + n(k+1).
\]

Step 2: In this step, the result from step 1 is generalized to arbitrary values of \( n \). It was already shown, that the formula is valid for \( n = 1 \). Assuming the validity of the formula was already shown for duration \( n \), for duration \( n + 1 \) with \( n \neq 2^k - 1 \) for \( k \in \mathbb{N} \), the number of positions does not change compared to duration \( n \). This means, when increasing the duration by 1, two cases can be distinguished:

1. The rightmost final node for duration \( n \) has a length of \( 2^i \) for some \( i \in \mathbb{N} \). Hence, increasing the length by 1 results in the creation of exactly 1 additional node. In total, there are \( n/2 \) such cases and hence \( n/2 \) new final nodes.

2. The rightmost final node for duration \( n \) has a length of 1. In this case, increasing the request duration by 1 has the effect, that some final nodes are merged. The reason is that 1 node of length 2 comes into existence, and in some cases this node of length 2 merges with another node, etc. The amount of nodes that actually "disappear" in this process is given by

\[
\log n - 1 + \sum_{i=0}^{\lfloor \log n \rfloor - 2} 2^i(\log n - 2 - i) = \log n - 1 + \frac{n}{2} - \log n = \frac{n}{2} - 1.
\]
The validity of the formula was already shown for $n = 2^i, i \in \mathbb{N}$ and therefore by combining the results from step 1 and 2, $f(n+1)$ with $n \in \mathbb{N}$ we obtain:

\[
 f(n+1) = f(n) + 1 + \sum_{i=0}^{\lfloor \log n \rfloor - 1} 2^i (i+1) + (n - 2^{\lfloor \log n \rfloor}) + 1
 = 1 + \sum_{i=0}^{\lfloor \log n + 1 \rfloor - 1} 2^i (i+1) + (n + 1 - 2^{\lfloor \log n + 1 \rfloor})
 = 1 + 2^{\lfloor \log n + 1 \rfloor} [\log(n+1)] + (n + 1) - 2^{\lfloor \log n + 1 \rfloor}
 = 1 + (n + 1) + 2^{\lfloor \log n + 1 \rfloor} ([\log(n+1)] - 1).
\]

This result can now be used to determine the remaining components that are required to determine the average amount of memory cells accessed during admission control.

The average length $l(n)$ of the final nodes can be easily computed as

\[
l(n) = \frac{n2^{\lfloor \log n \rfloor}}{f(n)}.
\]  

(2)

This is true because the duration of the request was assumed to be $n$ and a total of $2^{\lfloor \log n \rfloor}$ different positions in the tree exists. Furthermore, the average number of final nodes $\hat{f}(n)$ can then be computed as

\[
\hat{f}(n) = \frac{f(n)}{2^{\lfloor \log n \rfloor}}.
\]  

(3)

With these results, the average amount of memory cells accessed during a complete CHECK and UPDATE can now be calculated. It is assumed, that each CHECK phase requires to access the memory cells from the root node to each final node. During this phase, both the node value and the max value of each node must be accessed, i.e., 2 memory cells per node. During the UPDATE phase, in the worst case the same memory cells need to be accessed, in this case only 1 cell per node since either the node value (for final nodes) or the max value (for the nonfinal nodes) are updated.

The nature of the implementation requires that firstly all the CHECK phases for each link must be performed and only in case all these checks were successful the updates are made. This requires to store the final nodes found during the CHECK phase. The implementation presented here uses an additional array to store these final nodes. This means for each CHECK and UPDATE phase, $\hat{f}(n)$ additional memory cells must be accessed.

The number of nodes accessed during a complete CHECK and UPDATE phase depends on the book-ahead interval $b = 2^k$ and the request duration $n$:

\footnote{Final nodes are start nodes for the UPDATE phase.}
With given book-ahead interval, this formula allows to determine the minimal duration \( n \) for which the usage of trees pays off, i.e., faster admission times are achieved. For example, for \( n \approx 138.7489 \) with \( b = 65536 \), both structures achieve similar performance, i.e., \( a_{\text{array}}(n) = a_{\text{tree}}(b,n) \).

When using this formula for the example shown at the beginning of this section, we obtain \( a_{\text{tree}}(4,2) \approx 14.6323 \). The difference to the previously computed value of \( \approx 12.6667 \) results from including only different positions in (4). This requires \( n \) being much smaller than \( b \) which is not the case in our simple example.

Figure 12: \( a_{\text{tree}}(b,n) \) compared to \( a_{\text{array}}(n) \) for three different book-ahead intervals \( b \)

In Fig. 12, the values of \( a_{\text{tree}}(b,n) \) are depicted for \( b = 4096, 16384 \), and 65536 and varying \( n \) and compared to the respective numbers for arrays.

### 6.3 Support for Flexible Reservations

The previous sections only considered reservation requests with fixed parameters, i.e., fixed start and stop times. However, in many cases - especially in advance reservation environments - it is desired to support also more sophisticated allocation strategies such as finding the first suitable interval for a reservation of given length, or support for malleable reservations [2]. In such environments, it is required to access every single time slot within a given interval or to determine the first interval with sufficient resources. A detailed description can be found, e.g., in [2] and is omitted here due to the space limitation.

In order to support such request types, it is required to scan intervals larger than the requested duration \( n \) and also to determine the average utilization for those intervals [2]. Such functionality is also necessary to support routing algorithms based on other than simple shortest path routing [3]. In this section, the
considerations of sections 6.1 and 6.2 are extended to examine the suitability of both data structures for such reservations. In case a suitable transmission interval is found, the complexity of the \texttt{UPDATE} phase remains the same as described in the previous sections. Hence, in the following only the \texttt{CHECK} phase is examined which requires scanning the complete search interval.

Two additional request types are distinguished: \textit{malleable reservations} which require accessing each individual time slot within a given search interval and an \textit{interval search} which has the purpose to determine the first available transmission interval for a request of given duration and bandwidth.

\subsection{6.3.1 Array}

Using arrays, such functionality can be realized using a linear scan over the search interval. For given search interval length $t$, this means the scan can be implemented with at most $t$ memory cells of the array being accessed, independent of the request duration $n$. This holds for the worst case search for the first time interval with sufficient duration $n$ and bandwidth, and is also valid for support malleable requests, i.e., where the average or peak utilization within a large search interval needs to be determined. Formally, this means the average amount of memory cells accessed for malleable requests or the search within an interval $a_{\text{array/mall}}$ or $a_{\text{array/ival}}$ is given by:

\begin{equation}
  a_{\text{array/mall}}(t, n) = a_{\text{array/ival}}(t, n) = t.
\end{equation}

\subsection{6.3.2 Segment Tree}

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{segment_tree}
  \caption{Accessed nodes (gray) in the malleable scenario. Any node covering a part of the search interval must be accessed to obtain utilization for any single time slot within the search interval. The tree descend proceeds until the leaf nodes are reached. As described before, supporting malleable reservations may require knowledge about the link utilization at every single time slot. However, the tree was designed to hide this information as much as possible and only store information for larger periods of time in higher tree levels. In order to determine the utilization for every single slot with an interval of length $t$, $t/2^{k-i}$ nodes need to be accessed on tree level $i$ with $b = 2^k$ being the duration of the book-ahead interval (see Fig. 13). In contrast to the \texttt{CHECK} phase described in Sec. 4, in this case only the node value of each node needs to be accessed and the tree descend}
\end{figure}
always proceeds until the leaf nodes of the tree are reached. When $t$ denotes the duration of the search interval, the amount of memory cells accessed yields

$$a_{\text{tree/mall}}(k, t) = \sum_{i=0}^{i=k} \frac{t}{2^{k-i}} = 2t - \frac{1}{2^k}. \quad (6)$$

The search for the first time interval of duration $n$ within a larger search interval of duration $t$ which satisfies the bandwidth constraints requires less effort, since initially a CHECK phase as described in Sec. 4 can be performed for a request of duration $n$ starting at the beginning of the search interval, followed by checking each slot within the remaining slots of the search interval. In the worst case, every slot within this remaining interval needs to be checked. The initial check can be bounded using the formula from Sec. 6.2 for the UPDATE phase because only one cell needs to be accessed for this check. This leads to the following estimation

$$a_{\text{tree/ival}}(k, t, n) = \hat{f}(n) (k - \log l(n) + 1) + \sum_{i=0}^{i=k} \frac{t - n}{2^{k-i}} \quad (7)$$

where $t$ denotes the duration of the search interval, $n$ denotes the requested duration, and $k$ determines the length of the book-ahead interval with $b = 2^k$. While equation 6 represents an upper bound for the number of memory cells being accessed, in equation 7 only an average is given because $a_{\text{tree/ival}}(k, t, n)$ also depends on the requested duration $n$.

![Graph](image.png)

Figure 14: Memory cells accessed during CHECK phase: fixed search interval size $t = 500$ (left) vs. fixed request duration $n = t/3$ (right) for $k = 16384$.

In Fig. 14, the amount of memory cells accessed when admitting a flexible reservation request\textsuperscript{3} is given. When accessing any single time slot and fixed search interval $t$, the time required using trees is approximately twice as large as the time when using arrays, independent of the request duration (see equation 6). In contrast, the search for the first suitable interval achieves similar results

\textsuperscript{3}Flexible here refers to malleable or interval search.
than arrays when the request duration $n$ reaches the size of the search interval $t$. For malleable requests, the requirements using trees is more than doubled compared to arrays. The same holds for the case of variable search interval $t$ and request duration $n = t/3$. The performance advantage of arrays is even more evident in this case. Trees cannot achieve a similar performance at all. Due to the influence of the requested duration $n$, the curve for the interval search increases more slowly than the one related to malleable reservations.

### 6.4 Worst-Case Memory Requirement

The worst-case memory requirement of the data structures is another important performance aspect which can considerably impact the applicability of a data structure in a given environment. For the following computations, a memory requirement of 4 bytes per integer and 4 bytes per pointer is assumed.

When $b$ is the length of the book-ahead interval, i.e., the total number of time slots covered, arrays require $4 \cdot b$ bytes of memory. The original implementation of the segment tree uses 20 bytes per tree node: 4 bytes each for the max and node values, 4 bytes for the pointer to the child nodes and 4 bytes for start and stop time. The memory requirement can be reduced by 8 bytes by not storing the start and stop times within the nodes but computing them during the traversal. In addition to that, using dynamic memory allocation it is possible to further reduce the requirement during the run-time of the broker. However, in the worst case there is still a worst-case requirement of $(2b - 1) \cdot 12$ bytes when start and stop times are not stored. In case, the segment tree is stored using an array (i.e., each tree node corresponds to an element of the array), the memory for pointers to child nodes can be saved. This leads to a worst-case memory requirement of $(2b - 1) \cdot 8$ bytes which is 4 times the memory allocation to be made for arrays.

Concluding, the analysis shows that arrays have a significant advantage in terms of memory requirement (see Sec. 6.4). With respect to the number of memory cells accessed during the CHECK and the UPDATE phase, it was shown that trees only have an advantage when the requested duration is rather long. Arrays were competitive up to a duration of approximately $90 - 140$, depending on the duration of the book-ahead interval. However, when it is required to obtain the utilization for single time slots within a given search interval as is the case for scheduling malleable reservations, the segment tree cannot compete with arrays at all, i.e., the performance of arrays is never reached. This is independent of the request duration. Only in certain cases, when the first suitable time interval for a transmission is requested, trees can reach the performance of arrays (see Fig. 14).

### 7 Performance Measurements

In order to determine how the results previously described impact the performance in an actual application of the data structures, tests were made using a bandwidth broker, supporting also malleable reservations [2]. The results presented here may not be representative for any possible environment, in particular the measurements involving the whole bandwidth broker are very much implementation dependent. However, the figures presented here illustrate the
effects when applying the data structures in an actual environment and give an idea of how the performance can be affected.

7.1 Environment

Two types of tests were made. Firstly, the performance of only the single data structure (segment tree or array) was measured. The second test involved the measurement of the admission times in a multi link scenario, i.e., the admission speed of a bandwidth broker [2] in a realistic scenario. In addition to the admission control task, in the second scenario also the path computation in the network and further administrative tasks, e.g., storage of per-flow information, is included in the admission speed. In case the available bandwidth of a chosen link is not sufficient for the requirement of a given request, an alternative path must be found and checked. The measurements presented here were made on a 1 GHz Pentium IV PC with 1GB of main memory running Linux.

In order to test the performance of the bandwidth broker, request sets were generated with uniformly distributed source and destination nodes and exponentially distributed request inter-arrival times as suggested in [11]. These parameters are less important in this context as implied by the properties advance reservation environment, the key variable which determines the performance of the data structures is the requested duration. The results of the analysis and the previous measurements imply that request sets as used in [16], i.e., with a book-ahead interval of approximately 8,000 slots and extremely short reservation durations between 4 and 32 slots are inappropriate for a performance comparison because trees are not competitive at all under these conditions. In this evaluation, uniformly distributed request durations as used in [16] were assumed with varying mean. The resulting performance figures show clearly the limitations of trees up to a certain request duration.

Figure 15: Network topology used to measure the admission times of the bandwidth broker [1].

For each link in the network topology (see Fig. 15), the bandwidth broker keeps two data structures, i.e., each link is bi-directional and one data structure is required for each direction. This means, for the given topology a total of $2 \cdot 26 = 52$ data structures representing link utilization is kept. The results shown here may of course vary depending on the actual bandwidth broker implementation.

The segment tree implementation variant embedded in arrays was used for the time measurements. The tests related to the memory consumption (see Sec. 7.5) were made using the memory saving variant based on the pointer implementation with dynamic memory allocation.
7.2 Performance of the Data Structures

Initially, the admission time of the data structures as a function of the duration of the reservation request was measured. Every single reservation request was made on an empty data structure, i.e., no other reservations were present. For these results, only the time required by the functions accessing the data structures was measured, i.e., there was no overhead introduced by the bandwidth broker framework.

![Graph showing admission speed as a function of duration for different data structures.](image)

Figure 16: Admission speed as a function of the duration for book-ahead intervals of 4096, 16384, and 65536 slots.

Fig. 16 shows the measured admission times for the array and the tree implementation. In order to generate these results, successful requests of given duration were simulated, i.e., both the CHECK and UPDATE phase were performed. For each duration, a request was generated for any possible start time within the book-ahead interval. Thus, each sample point represents the average of the measured times for both phases at each possible slot and hence, these measurements are comparable to the analytic result depicted in Fig. 12.

In general, the curves are similar as those determined by the analysis with the only difference that arrays perform even better. It can be observed that arrays are competitive up to a reservation length of approximately 300 – 500 slots which is about three times the value computed in Sec. 6.2. This is due to the actual implementation which, besides the number of memory cells accesses, requires other operations such as function calls or index computations. In this context, trees are more complicated to implement and hence, more administrative framework is required which also affects the actual run-time of CHECK and UPDATE routines.

7.3 Multi-Link Admission Control

In order to obtain realistic results that reflect an actual application scenario, in this section measurements are presented for the complete admission control process in a bandwidth broker using a network with multiple routers. The broker implementation as used in [2, 4] was chosen for these tests. The bandwidth broker adopts a shortest feasible path algorithm [3] using the request parameters as described in Sec. 7.1.

The total processing time of the bandwidth broker for a single request consists of various tasks. The most important one in terms of processing time is the check for sufficient bandwidth on the links of the network and updating of
the internal data structures. Following that the routing, i.e., the search for a suitable path using the results from the bandwidth check, is the factor with the next lower amount of processing time in the broker. The remaining tasks each require only short amounts of time. These tasks are related to authentication, queuing of incoming requests, the framework around routing and bandwidth check, and the representation of the network in the bandwidth broker. Furthermore, besides updating the data structures, for each accepted flow some more information needs to be stored, such as the source and destination node, the path it takes, the start and stop time, etc. In the following, the tasks not directly associated with either routing or the data structures are all referred to as administrative tasks.

Figure 17: The processing time of the bandwidth broker divided into the different tasks routing, data structure related processing, and the remaining parts (administration) to be performed by the broker. The average request duration was 500 slots.

The processing time of the bandwidth broker is depicted in Fig. 17, decomposed into the time spent for each of the three main parts data structures, i.e., mostly CHECK and UPDATE operations, routing, and the remaining administrative operations to be performed, e.g., storing information such as source and destination node for admitted requests. The figure shows, that using segment trees almost 60% of the total processing time of the broker is required for data structure related operations, whereas the processing time can be reduced to approximately 30% when using arrays instead. The overall performance difference results completely from using different data structures, the remaining parts of the broker are not affected in their run-time. The results were measured using an average request size of 500 slots which denoted the "break-even" point of arrays and tree as indicated in Fig. 16.

Since this does not hold in the multi-link scenario, the average admission speed depending on the average request duration is depicted in Fig. 18. As indicated by the plots in Fig. 17, the actual length of the book-ahead interval is less important in this case. It can be observed, that using the multi link scenario, the "break-even" point is reached at a request duration of approximately 1,500 slots, i.e., three times more than measured for a single data structure.

Using an average request duration of 500, the broker is capable of handling about 8,500 simultaneous requests when using arrays and only about 5,000 simultaneous requests when using trees.
7.4 Flexible Reservations

7.4.1 Performance of the Data Structures

In order to examine the impact of the data structures in case of interval search and malleable reservations, time measurements were made that underline the analytical results presented in Sec. 6.3.

Figure 19: Single data structure: Measured time of only the CHECK phase for malleable requests: fixed search interval size $t = 500$ (left) vs. fixed request duration $n = t/3$ (right). The results were measured using a book-ahead interval of $b = 16384$, i.e., $k = 14$.

In Fig. 19, the average time required for the only the CHECK phase when scanning an interval of fixed size (500 slots) is depicted. Alike in case of fixed reservations, it can be clearly observed that trees have an even larger disadvantage than shown in the analysis (see Fig. 14). In order to obtain those results, the situation as analyzed in Sec. 6.3 was simulated, i.e., for each request the worst-case scenario was simulated, i.e., the maximal number of tree nodes had to be accessed. Using arrays, the actual times measured here are shorter than e.g., those shown in Fig. 16, since here only the CHECK phase was measured.

7.4.2 Multi-Link Admission Control

As depicted in Fig. 20, when processing malleable requests in the multiple link scenario, the percentage of the overall processing time spent with data structure related functions is significantly increased compared to the results shown in
Fig. 17. It can be observed, that the time spent in the data structures nearly completely determines the total processing time. The diagram shows the average processing time required for a whole request set. The request sets used to generate these figures only contained 30% malleable requests. These figures show, that the actual choice of the data structures plays an important role for the admission speed of the bandwidth broker, in particular, using segment trees results in about 10 times higher processing time compared to arrays.

Figure 20: Processing time of the bandwidth broker for malleable requests. The average search interval length was 1000 slots.

In Fig. 21, another important factor with considerable impact on the processing time is depicted. It can be observed that, the maximum processing time for a single malleable request, computed over the whole request set, is up to 100 times larger than the average. In particular, the maximal processing time was several seconds for a single request. The reason is that some malleable requests can be admitted very quickly without performing a large number of checks, whereas in the worst case, i.e., a rejection of the malleable requests, the amount of checks to be performed is significantly higher. Therefore, the actual network load is an important factor with respect to the admission time required for a single malleable request.

The dependency of the processing time on the average search interval length is depicted in Fig. 22. The diagram shows the similar effect as outlined Fig. 19: the advantage of array increases with the search interval length. The length of the book-ahead interval is less important. Although the processing times rise slightly in the scenario with 65536 slots, in general the difference between trees and arrays remains stable. The reason is, that using trees the malleable scenario...
requires frequently accessing leaf nodes, i.e., traversing 14 (using a book-ahead interval of 16384) or 16 (using a book-ahead interval of 65536) tree nodes. Hence, only two additional nodes must be accessed for each traversal which influence the overall processing time only moderately.

7.5 Memory Consumption

In the previous sections, only the admission speed using trees or arrays was examined. Memory consumption is another important factor for the applicability of a data structure in the given environment. In this case, using the tree statically embedded in an array has a great disadvantage over arrays with 4 times higher memory consumption (see section 6.4). In section 4.1, another implementation alternative was presented, based on pointers and dynamic allocation of tree nodes during run-time. Although this tree variant has an even larger disadvantage in terms of admission speed, for the sake of completeness it is also tested with respect to the memory performance. For that purpose, the same parameters as previously used were chosen to simulate the multi-link scenario with the given topology.

In Fig. 23, it can be observed that the usage of dynamic memory allocation has a significant advantage over the static tree implementation which requires 4 times more memory than arrays. However, trees are still not as memory efficient as arrays. On the left, the development of the overall memory consumption of all data structures in the bandwidth broker is depicted using different average request durations. The memory consumption heavily depends on this parameter.
as outlined on the right hand side. It can be observed that trees retain the worse memory performance up to an average duration of 4,500 slots. This means, in this case trees show an even worse performance than in terms of admission speed where equal performance could be achieve at approximately 1,500 slots.

8 Related Work

Advance reservations have been considered in different environments. Originally familiar from the field of flight or hotel reservations, this type of reservations has been applied in different areas mainly in the field of distributed computing. Advance reservations are mainly used when co-allocation of different resource types is requested, as is the case in grid computing environments such as Globus [10]. The resource management system used in Globus (GARA) has been described in [9]. GARA supports advance reservations for various resource types, e.g., compute nodes or network bandwidth, and uses slotted time. Besides computer networks and grid computing, in [7] advance reservations were also applied in the field of content management systems for distributed media servers. Such content distribution networks also benefit from the opportunity to reserve resources in advance since this allows the timely transfer of large amounts of data among different servers.

The field of advance reservations in networks, as is the focus of this paper, has also been examined in various publications. For example, fundamental requirements for supporting this reservation type in computer networks have been outlined in [8, 19]. Later considerations deal with various aspects such as routing [12] or admission control [13]. Fast and efficient access to the information stored in the data structures is not only important during the initial admission control process. In addition, when unexpected situations arise during the runtime of the reservation system, e.g., resource failures or reconfigurations, it is necessary to remap allocations to other resources. For example, in the network environment flows must be switched to alternative paths as fast as possible [4] in order to avoid noticeable degradation of the guaranteed quality-of-service. This requires fast access and update functionality. Instead of using slotted time [2, 9, 12, 17], another approach to this problem is to allow allocations for arbitrary periods of time without limiting the start and stop time to the granularity of slots. Such an approach was presented in [18]. However, for reasons already explained in previous sections, the usage of slotted time is favorable.

In addition to the segment tree discussed here, in [16] a binary search tree was presented which also uses slotted time. However, the number of tree nodes depends on the amount of admitted requests which results in poor scalability of the approach. Furthermore, this tree has the drawback of getting unbalanced over time. Hence, balancing is required periodically which results in significant performance disadvantages and therefore, this tree was not considered here.

9 Conclusion

In this document, arrays and a specially designed tree for admission control in advance reservation environments were examined with respect to their impact
on the admission control performance in advance reservation environments. The examinations were made both analytically and by measurements. Those data structures are important as they make up the major portion of the total processing time of the management system. Besides being required for admission control of requests, the data structures are also accessed by off-line processes, i.e., when failures occur and a fast remapping of the requests allocated to failed resources must be initiated.

The most important result of the examination of both data structures is that arrays, being one of the simplest conceivable data structures, have a significant advantage over segment trees below a certain request duration when examining the simplest reservation type with fixed parameters. The picture is different when investigating more advanced reservation methods where the full strength of advance reservations is used, e.g., malleable reservations. In those cases, it is required to scan larger intervals than in the simple case, in particular with the additional requirement of having access to the resource allocation of every single time slot. For this reservation type trees are unsuitable and - apart from some rather unlikely cases such as requests covering the whole book-ahead interval - always result in a worse performance. Since request types such as malleable requests are one of the key improvements in advance reservation environments compared to an immediate scenario, it is likely that in real-world implementations especially this request type will be of significant importance.

The easy implementation, e.g., support for the dynamically advancing time, is an additional factor in favor of arrays. It can be stated, that the usage of arrays is favorable in the examined environments. The analysis showed that the superiority of arrays results from the property, that the information is stored locally at each array element rather than distributed across several nodes as is the case for the tree.

The memory consumption of trees was far worse compared to arrays using any of the tested implementations. Even with dynamic memory allocation, trees were only competitive with very long average request durations. In particular, those durations were several times longer than those were the static tree implementation became as fast as arrays.

The measurements presented here were mostly made in network management systems. However, the analytical results and the measurements of individual data structures can also be applied to any other resource management system with support for advance reservations [9]. Although the measured times depend on the actual implementation and hardware platform, the general result that arrays provide a much better performance up to a certain average request duration can be generalized to other areas of applications.

Future work will deal with issues that could not be discussed here, e.g., the impact of the data structure related processing in a multi-resource environment such as a grid. In such a case, a single job consists of multiple sub-tasks which must be reserved in a coordinated manner.

References


