Performance of QoS Agents for Provisioning Network Resources

Olov Schelén, Andreas Nilsson, Joakim Norrgård and Stephen Pink
Computer Science and Electrical Engineering
Luleå University of Technology, SE – 971 87 Luleå, Sweden
{olov, andreas, nogge, steve}@cdt.luth.se

Abstract

We have designed an agent-based architecture for quantitative service provisioning in differentiated services capable networks. For each link-state routing domain in the network there is a topology-aware QoS agent (also known as a bandwidth broker) responsible for admission control. The architecture provides resource reservations for aggregated virtual leased lines between network domains. In this paper, we present performance measurements for resource provisioning in a prototype QoS agent. This includes an evaluation of two data structures for advance reservations and accompanying algorithms. We also compare the cost for on-demand route computations with pre-computation of routes. The objective in this paper is to evaluate the performance of end-to-end admission control within a single link-state routing domain. In a domain with 15 routers, 28 transition networks and 64 stub networks, our prototype performs approximately 25 000 end-to-end admission decisions per second. The results show that an ordinary PC can be used for running a QoS agent that performs path-sensitive admission control and maintains per-link resource reservations in a link-state routing domain.

1 Introduction

Traditionally, the Internet was used by a homogeneous group of users running elastic applications that could adapt well to varying network performance. Then, it was sufficient to provide one single service class known as best-effort service. In the Internet today, there are users with different demands for service quality (e.g., business users, home entertainment users, etc). Also, there is a wide range of applications that require various levels of network performance. Interactive real-time applications (e.g., Internet telephony, multimedia conferencing, etc.) require low round-trip delay and a minimum throughput to work well. It is hard to determine a single level of service to fit all users and applications in terms of price and performance. Multi-level service differentiation can be an aid to meet these new demands.

There have been many proposals based on introducing per-flow reservations in the network to provide quality of service. For the Internet, a well known approach is the IETF standards for Integrated Services in routers [21, 7] and the end-to-end reservation setup protocol RSVP [4]. These proposals result in per-flow state in network nodes along the path of the flow. An advantage with such an approach is that fine-grain service agreements can be supported by having per-flow packet scheduling in routers. However, it has been argued that these models are too complex and do not scale well enough to be used in the backbone of the Internet [12].

This has resulted in a new approach being taken in the IETF, known as *differentiated services* [2]. The guideline is to provide scalable QoS support by having no per-flow state in routers. The basic idea is that IP packet headers include a small label (known as the *diffserv* field) that identifies the service (per-hop behavior) they should be given by the routers. Core routers are configured with a few per-hop behaviors and packet labels are used to map packets into the right service behavior. The architecture relies on packet markers and policing functions at the edges of the network to ensure that the services are used according to the service level agreements made with the clients. Differentiated services may provide a range of services, both *qualitative* (relatively defined) and *quantitative* (absolutely defined).

Qualitative services (better than best-effort) can be provided by only relying on differentiated services capable routers. To provide quantitative (minimum expectation service), however, resources must be administrated by separate mechanisms to make sure that the service commitments can be met. This can be arranged either manually through network managers or automatically through admission control units. Service agreements involving a minimum expected level of service must involve some kind of admission control to make sure that there are sufficient resources available in the network to meet the agreements. Admission control gives the provider an opportunity to reject service to new clients so that those already granted admission obtain committed service.

In earlier work [17, 18], we have designed a QoS architecture that can provide *virtual leased lines* over differentiated services by performing admission control in agents (also known as *bandwidth brokers*). For each routing domain in the network there is a QoS agent responsible for admission control in the domain. We have shown that path-sensitive admission control can be supported with favorable scalability properties. This is because reservation state for virtual leased lines is aggregated in agents as their paths merge towards the destination domains. Each agent along the path performs admission control within its own domain and aggregates several commitments into one commitment (per destination domain) with the downstream agent. Consequently, reservations form sink trees towards destination domains.

Path-sensitive admission control is possible as each QoS agent knows the topology and static link resources in its domain. The agent is an end-system that is configured for passively participating in a link-state routing protocol (OSPF) where each participating router has an identical topological database over the domain. Agents retrieve link properties, such as static bandwidths, by querying routers seen in the topological database. For this, we use a network management protocol (SNMP). Agents use the resource map to perform admission control in their domain. Admission requests contain the bandwidth to be reserved, a source and a destination address prefix.

1.1 Scope and layout

In this paper we present a prototype implementation of a QoS agent and focus on obtaining performance measures for admission control within a single link-state routing domain. The prototype includes functionality for:

- obtaining a topological database through the link-state routing protocol,
- obtaining link bandwidths through SNMP,
- performing path-sensitive end-to-end admission control within a single link-state routing domain.

Our evaluation focus on the following performance measures:

- Single-hop admission decision speed. This includes comparing the admission decision speed of two data structures for advance reservations. We compare their sensitivity to the total number of reservations, the maximum book-ahead time, the length of the requested durations, etc.
- Multi-hop admission decision speed. In addition, this includes comparing the time for routing decisions when paths are computed on-demand or pre-computed. We also evaluate the protocol processing cost for receiving admission requests over the network.

Our prototype support both immediate and advance reservations. Immediate reservations start as soon they are granted and are open-ended (i.e., clients may hold on to them as long as they like). Advance reservations are time-limited (i.e., they have a starting time and a finishing time). We use peak bandwidth as the quality-of-service metric for virtual leased line service. Other metrics such as delay and packet loss are not specified in admission requests. Instead, it is the responsibility of the provider (i.e., the owner of the QoS agent) to configure schedulers and to provision resources so that acceptable service is obtained in the traffic class for virtual leased lines.

The paper is organized as follows. In section 2, we motivate the need for advance admission control. In section 3, we introduce two data structures for advance admission control. In section 4, we describe immediate admission control. In section 5, we describe the prototype setup and experimental environment. In section 6, we evaluate single-hop admission control. In section 8, we evaluate multiple-hop admission control (including the cost for route calculation). In section 9, we evaluate the cost of request/reply transactions. In section 10, we present related work and finally we conclude the paper.

2 Resource provisioning over time

In the telephone network, admission control is performed as we initiate a call by dialing a number. Once admitted, clients know that they have sufficient quality for voice communication and that they may hold on to the call as long as they like.

This service model may not be appropriate for the Internet. First, there is a new set of real-time applications involving many users. Carrying out sessions (e.g., conferences, lectures, games, etc.) with these applications often require a certain degree of planning so that the participants are attending at the same time. Booking network resources should be part of this planning. It is unfortunate if a planned event has to be canceled with late notice because it turns out there are insufficient network resources at the time.

Second, there is an increasing demand for long-term provisioning of trunk network capacity. This is often provided through virtual leased lines and virtual private networks. These services today involve tunneling and manual configuration of routers along the paths. There is a demand for having automated setup of such services. Trunk bandwidth contracts generally involve planning of network resources over time (e.g., time-limited contracts, various bandwidths at different times of the day etc.). It is reasonable that admission requests can include timing information to allow this. We believe that the need for advance negotiation and time-limited contracts is especially desirable in the network core when providers negotiate QoS contracts between each other.

In our work we meet the demands for planning network resources over time by exploring support for advance reservations. In addition to the traditional so-called *immediate* admission control (where no timing information is included) we provide *advance* admission control. Advance admission control allows clients to plan their activities provided they give a duration (i.e., a starting point and a finishing point) of the service level agreement. This also allows providers to schedule their resources over time and predict when the total capacity will be insufficient.

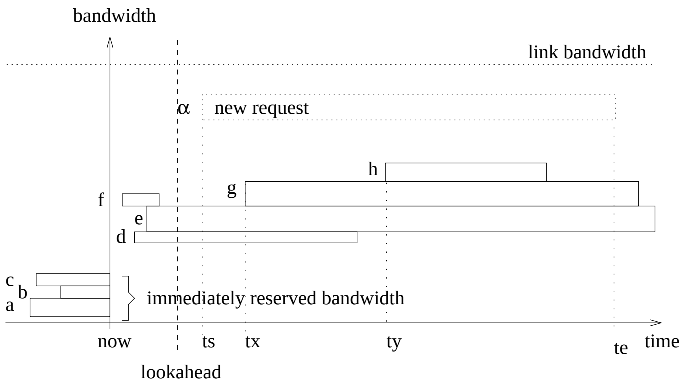

Figure 1: Advance reservations in a time-bandwidth domain

3 Advance admission control

In our model, advance admission control is two-dimensional problem involving bandwidth and time. Admission control can be seen as fitting boxes (i.e., rectangles) in a two-dimensional time-bandwidth diagram (figure 1). For this we have implemented two data structures supporting continuous or slotted time.

3.1 Continuous or slotted time

The overhead for advance reservations depend on whether we use continuous time or slotted time. With continuous time, we allow reservations to start and end at any time. With slotted time, we enforce a minimum granularity on the starting and finishing times. Slotted time allows more aggregation of reservation state. This results in faster admission decisions and less memory usage.

In our experiments we have used two data structures. The first one (section 3.2), supports continuous time for reservations, but benefits from having slotted time. The second data structure (section 3.3), relies on having slotted time. Below we give a short overview of the two data structures and accompanying algorithms used in our experiments.

3.2 A binary search tree over time

The first data structure is a binary search tree over time. Every reservation is represented by two nodes in the tree: one for the starting time and one for the finishing time. The node at the starting time contains a positive bandwidth *delta* and the node at the finishing time contains a corresponding negative bandwidth *delta*. If there are several reservations starting or finishing at the same time, the sum of bandwidth deltas is stored in a node representing that point in time.

To obtain efficient admission decisions, we have added some information to the nodes (figure 2). First, the aggregate bandwidth reserved in the left subtree (*ltot*) is stored in each node. This value is updated as new elements are inserted in the left subtree. Second, there are links (dashed arrows) to the closest right neighbor in the tree so that the list of consecutive nodes in time can be obtained starting from any point in time.

The admission decision involves a binary search to find the starting time of the requested duration. In this traversal, the total aggregate bandwidth reserved in left children is computed (i.e., for each step to a right child, the aggregate bandwidth of no-traversed left children is updated). Thus, as the starting time is found, the total aggregate bandwidth reserved at that time is known. Admission control involves a linear search through the requested duration. At each point of change, the total aggregate is updated by adjusting for the delta. At each positive delta, we check whether the total aggregate plus the new requested bandwidth is still below the link bandwidth.

The complexity for finding or inserting a given time in this data structure is on average $O(\log(n))$, where n is the number of nodes in the tree. The complexity for searching through the duration spanned by a new request is $O(m)$, where m is the number of nodes in the requested duration. Thus, the complexity of admission control is $O(m + \log(n))$. However, when slotted time is used, n is bound by the total number of slots and m is bound by the length of the duration divided by the granularity of slots (i.e., by the number of time slots spanned by the requested duration).

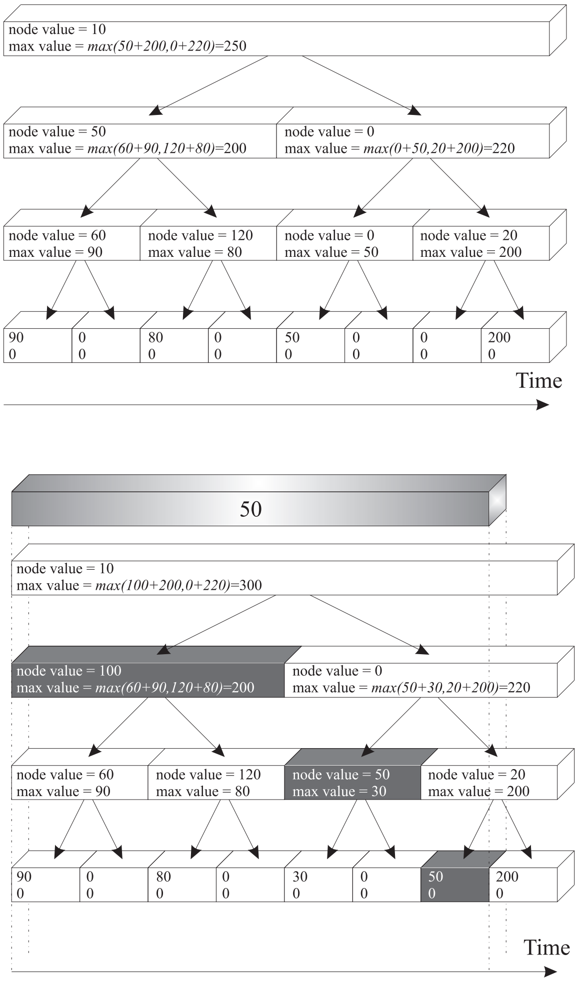

Figure 3: A segment tree before and after insertion

3.3 A segment tree over time

The second data structure, described in [14], is based on Segment Trees. Each node contains a time frame (duration) and the amount of reserved bandwidth during that time frame. The time frame of the top node equals the whole period of time that is covered for advance reservations. Each time frame is recursively divided into smaller, equally sized, time frames. The time frame of each leaf is equal to one time slot. Thus, the height of the tree is decided by the time frame of the top node, the branching degree, and the granularity of the leaf nodes.

Each node stores the aggregate bandwidth of all reservations spanning over the whole time frame represented by the node. This value is denoted the *node value*. Each node

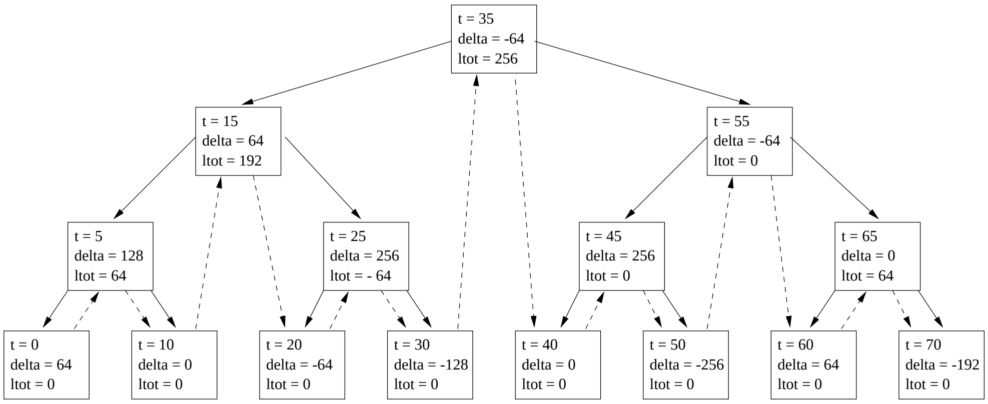

Figure 2: A binary search tree over time

also stores the maximum sum of node values in any of the branches (i.e., paths) below the current node. This value is denoted the *max value*. An admission request can be viewed as a block with a given duration that falls through the data structure (figure 3). The admission operation is performed as follows.

- If the duration of the falling block is within the time frame of the node, the block falls through to the level below.
- If the duration of the block intercepts with the time frames of two or more nodes at a given level, the block is split at the interception points before it falls to the next level.
- However, if the duration of a block fits perfectly within a time frame of a node (before or after a split) it does not fall further.

While a block falls through, the bandwidth reserved at the levels it falls through are added up. When a block fits perfectly in the time frame of the current node, the admission decision for that block is made. This is done by adding the reserved bandwidth in the fallen through levels, the bandwidth reserved in the current node, the max value of the children of the current node, and the bandwidth requested by the falling block. If this sum is smaller than the link bandwidth, the request can be granted (for a sub-duration) and inserted.

The details of the operations as well as some optimizations are found in [14]. All operations can be performed in constant time, i.e., they are independent of the number of reservations stored in the tree. The time depends, however, on the number of time slots and the length of the requested duration (see section 6).

4 Immediate admission control

Immediate reservations are open ended (i.e., they start immediately and can be held on to by clients as long as they like). Admission control for immediate reservations is performed by recording the aggregate bandwidth of all immediate reservations. In addition, admission control looks ahead a short period of time into the advance reservation state to find out about resources needed for advance reservations in the near future. The lookahead time depends on how much bandwidth that is requested for each advance reservation and the amount of bandwidth per time-unit that is returned by immediate reservations ending. In most cases the lookahead time is much shorter than the average duration of a reservation. This issue is covered in [17].

An admission decision is taken by adding the requested bandwidth to the aggregate of current immediate reservations. Then, the lookahead interval is checked for sufficient resources so that both advance reservations and immediate reservations can be served. This operation is similar to performing advance admission for the lookahead duration, with the exception that granted bandwidth is added to the aggregate immediate reservations instead of being recorded as an advance reservation. Since the lookahead time is relatively short, immediate admission decisions are on average performed faster than advance admissions.

5 Prototype Setup

5.1 Framework

We have implemented a prototype agent as an application program for the FreeBSD 2.2.6 operating system. All experimental results presented in this paper are obtained when running the software on an Intel Pentium II, 300 MHz PC, with 128 MB of memory. The machine is configured as a passive router to obtain the topological database for its link-state routing domain (i.e., the machine does not advertise any routes but it receives link-state advertisements from other routers). For this, we use the OSPF [13] implementation as provided in the *gated* 3.5.8 distribution [6]. Link bandwidths are collected by an agent from the routers by using SNMP [9].

5.2 Topology

The prototype runs in the network at Luleå University of Technology. The OSPF domain comprises 15 routers, 28 transit networks, and 64 stub networks. Transit networks and stub networks are mostly 100 Mb/s or 10 Mb/s Ethernets that may be switched or hubbed internally. To OSPF, the transit networks are perceived as clouds interconnecting at least two routers while stub networks connect one router to a number of end systems. The bandwidths available for admission control are determined by the speed of the interfaces at the routers (as found through SNMP).

The domain is flat (i.e., there are no OSPF sub-areas) and therefore one agent can perform admission control within the whole domain. In a larger OSPF domain with sub-areas, each sub-area would have an admission control agent (i.e., there would be a two level hierarchy).

6 Evaluation of single-hop admission control speed

We have performed experiments by using both the binary search tree (section 3.2) and the segment tree (section 3.3). Here we present performance measurements only for advance admission control. This is because immediate admission control is a less demanding special case of advance admission control. (section 4). The tests are performed by generating a large number of requests that go through admission decision as fast as possible. Granted requests are not cancelled or removed, because advance reservations automatically time out when their duration expires. Also, the computational cost for removing reservations is less than or equal to the cost of performing new admission decisions. We present our results in figures where each plotted point is obtained by averaging over 1 000 consecutive admission decisions.

Figure 4 shows the admission decision time when performing admission control over a single link where there are up to approximately 800 000 granted reservations spread over 30 days. The following parameters are used in this example:

- the link speed is 100 Mb/s,
- each request asks for a bandwidth of exactly 64 kb/s (we have chosen a small value to obtain a large number of reservations),
- requested durations are uniformly distributed in [20..180] minutes,
- requested starting times are uniformly distributed over 30 days (i.e., 43 200 minutes),
- time is slotted with a granularity of five minutes.

In this case, it is clear that advance admission decisions on average can be performed in less than $10\mu s$ when using the search tree implementation and less than $7\mu s$ when using the segment tree implementation. These results are for admission decisions over a single link (including decision time and storage time). When a reservation involves several hops, the corresponding time will be spent for admission decision over each hop.

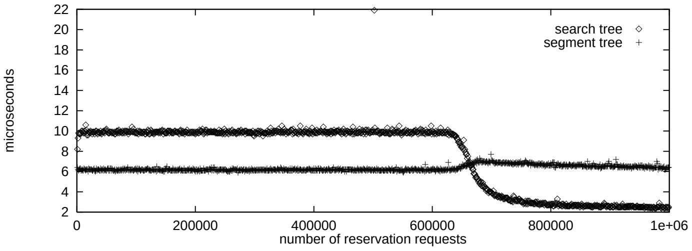

Figure 4: Admission decision speed for a single link

6.1 Dependence on the number of rejections

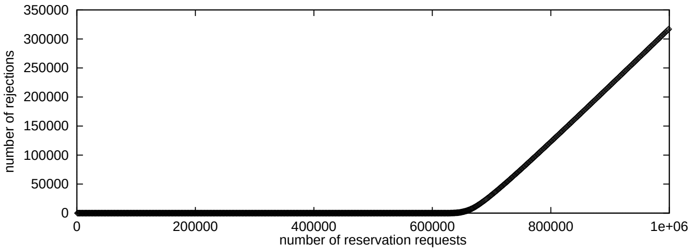

Figure 5: Rejections for a single link

In figure 5, we show that resources are reserved up to and above the point where we have rejections. The knee shows at which number of reservations we start getting rejections. From figure 4, it is clear that admission decision speed changes when there is a large number of rejections. In the binary search tree, admission decisions are taken before the corresponding reservations are stored. Also, each admission decision ends as soon as a point of insufficient resources is found. Therefore, the admission decision time decreases when there are rejections. In the segment tree, traversed segments are updated as an admission decision proceeds along a branch of the tree. If there is a rejection, the reservation state has to be restored. Therefore, execution time increases when there are rejections. The major difference is explained by the way we have designed our operations. There is a choice between optimizing either for fast acceptance or for fast rejection.

6.2 Dependence on the number of reservations

It is somewhat surprising that the binary search tree appears to have an almost constant execution time (figure 4). Intuitively, the execution time should have an $O(\log(n))$ component for finding the starting time and an $O(m)$ component for admission decision through the requested duration, where n is the number of nodes in the tree and m is the number of nodes in the requested duration. However, these effects are visible only in the range of very few reservations (e.g., $< 10\,000$). As the number of reservations grow large, the data structure grows to a maximum number of nodes. In this case there are never more than 8 640 nodes in the tree. This is because there are 8 640 slots of five minutes in a period of 30 days. Once the tree has reached the maximum number of nodes, each set of admission requests will on average take the same time even if the total number of reservations increase. Thus, aggregation of reservation state results in both data structures having execution times that are independent of the number of reservations stored (when the number of reservations is reasonably large).

6.3 Dependence on insertion order

The segment tree is always fully balanced independently of the insertion order. The search tree, on the other hand, may get unbalanced if the requested durations are ordered. In the worst case (i.e., when insertions are totally ordered), all nodes in the structure must be traversed. In our experiments here presented, the requested durations are generated randomly with equal probability over time. Thus, the search tree is well balanced. If requested durations sweep over time (e.g., if most reservations are made one day ahead while current time moves forward), the search tree will tend to be filled from the left hand side to the right hand side. Consequently, the tree would have to be re-balanced.

6.4 Dependence on the reservable time frame

The search time depends on the total number of slots (i.e., nodes) stored in the data structure. This number is determined by the reservable time frame, divided by the slot size. Using a slot size of five minutes, figure 6 shows how the search time depends on the reservable time frame. We measure the average admission decision time over granted requests only.

For both data structures, the search time is in $O(\log(n))$, where n is the total number of slots. However, the constants for admission decision over the given durations are smaller for the segment tree.

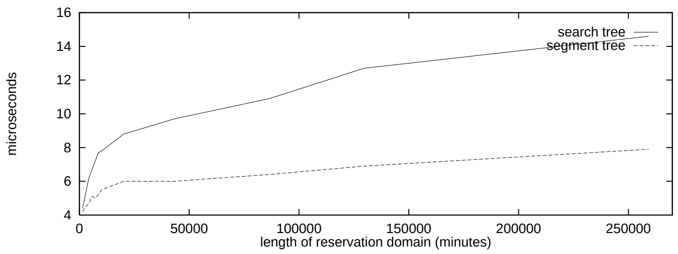

Figure 6: Decision time and reservable time frame

6.5 Dependence on the length of requested durations

The number of time-slots spanned by a requested duration is directly proportional to the length of that duration. In the search tree implementation, there is a linear search through the time slots spanned by the duration. This results in admission decision time for the search tree being directly proportional to the length of the requested duration, as verified in figure 7. (the curve represents the average admission time for a large number of requests having a particular average duration)

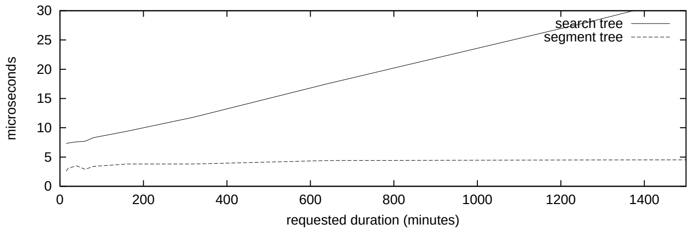

Figure 7: Decision time and length of requested durations

The segment tree shows a logarithmic curve (figure 7). The reason for this is that longer durations are more likely to be split at several levels of the tree (see section 3.3). There is some computational cost associated with each split. The worst case is that each end of the falling block is split exactly once at each level of the tree (where the top level is excluded). Thus the worst case is $2*(d-1)$ splits, where d is the depth of the tree. As described in section 3.3, the depth is in $O(\log(n))$, where n is the total number of slots in tree.

6.6 Memory requirement

Our segment tree implementation has one leaf per time slot. The branching degree is configurable (individually for each level). Assuming the worst case, i.e., a binary tree, there is a total of $2n-1$ nodes, where n is the number of time slots spanned. Each node contains one pointer to where the children are stored and four integers (start time, stop time, node value, max of children). Assuming four bytes per pointer and four bytes per integer, each node requires 20 bytes. When spanning 30 days using 5 minute slots (i.e., when there is a total of 8 640 time slots), the memory requirement adds up to $(2*8\,640-1)*20 = 345\,580$ bytes. A higher branching degree decreases the memory requirement.

The search tree implementation requires one node per time slot in the worst case. In the current implementation, each node contains four pointers (left child, right child, next, previous) and four integers (time, aggregation count, aggregated bandwidth delta, bandwidth of left child). This means that each node takes 32 bytes. For 8 640 time slots, this adds up to $8\,640*32 = 276\,480$ bytes. However, by allocating the pointers in a chunk and have one pointer to that chunk (as we did in the segment tree implementation), we would need only 20 bytes per node. Then, the total memory requirement for 8 640 time slots would be $8\,640*20 = 172\,800$ bytes. Note also that the search tree requires less memory if there are few reservations.

6.7 Discussion

Our experiments show that the segment tree generally has better performance than the search tree. The segment tree is the best choice for large backbone links where there is a large number of requests with long durations (e.g., for reserving trunk bandwidth). In addition, the segment tree is insensitive to insertion order. The drawbacks of the segment tree are that it requires slotted time and that is uses more memory. Having slotted time seems natural for advance reservations and should not be a problem. The higher memory consumption for the segment tree might be hard to motivate for narrow bandwidth links where there can only be a few reservations at any time. In that case, the search tree implementation is more efficient.

The segment tree must be pre-configured for covering a certain period of time (e.g., by having a different tree for each month). The search tree, on the other hand, can continuously adapt as time proceeds (with the risk of having to be re-balanced).

7 Routing calculations

Our architecture requires that link-state routing is used in agent domains. The server on which the agent is running participates in the routing protocol to obtain the topological database for the domain. Our prototype runs OSPF as implemented in the *gated* distribution.

The following basic functionality is part of OSPF [13]. Each OSPF node maintains an identical database describing the topology of the domain. From this database, a routing table is calculated by constructing a shortest-path tree. OSPF recalculates routes quickly in the face of topological changes, generating small amounts of routing protocol traffic. In an OSPF domain, each router generates its routing table from this graph by calculating a tree of shortest paths with the router itself as root. Obviously, the shortest-path tree depends on the router doing the calculation. The tree gives the entire path to any destination network or host in the domain. In routers, however, only the next hop to the destination is used in the forwarding process. As a result, packets sent from a node will follow the paths found in the shortest-path tree of that node.

An admission control agent must be able to calculate shortest-path trees starting at any router in the domain. For this, we use a modified Dijkstra's shortest path algorithm (i.e., with a binary heap implementation for the priority queue).

8 Evaluation of multi-hop admission control speed

End-to-end admission control within a link-state routing domain involves finding the path between the end points and performing an admission decision over each hop in the path. Here, we compare the cost for on-demand route calculations (i.e., performed for each admission request) with the cost for accessing pre-calculated routes.

We evaluate the average time to process an admission request within the topology used at Luleå University of Technology (section 5.2). For the end-to-end admission requests

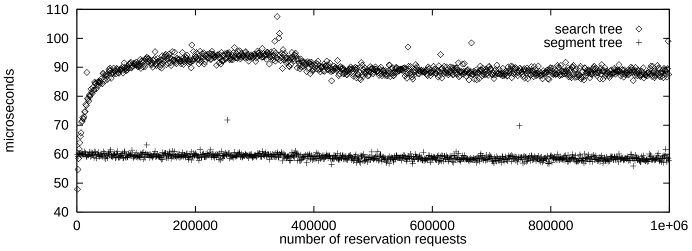

Figure 8: Admission decision speed within the LTU domain

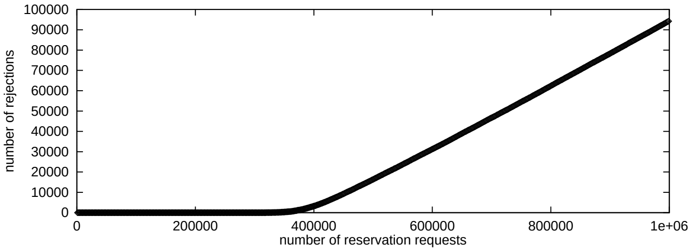

Figure 9: Admission rejections within the LTU domain

generated, the average number of hops is 3.2. In these experiments, admission control includes both finding the shortest-path and performing admission decisions over each hop in the shortest path. In our implementation, admission decisions are performed for each router interface along the path, i.e., twice for each intermediate hop and once for each of the two end hops. A path of h hops therefore involves $2 * (h - 1)$ admission decisions. An average path of 3.2 hops, as in our experimental domain, involves 4.4 admission decisions.

8.1 On-demand route computation

Figure 8, shows the performance when routes are computed on-demand. As shown in section 6, the time needed for admission decisions per-link in the segment tree is at most $7\mu s$. Thus, the average time spent for admission decisions over a path is $4.4 * 7 \approx 31\mu s$. Approximately $20\mu s$) is spent on shortest path first calculations to find out which links are involved in the request. The remaining $9\mu s$ is overhead for traversing the computed path.

In figure 9, we show that resources are reserved up to the point where we have rejections. From figure 8, it is again clear that the admission control speed does not suffer from having a large number of rejections.

We have implemented multi-hop admission control using a one-pass mechanism with cleanup. We reserve resources hop-by-hop and if there is a rejection at one hop, we reset the reservations at all previous hops. Since rejections on average happens halfway to the destinations, a rejection means setting up reservations along half the way and then tearing down those reservations. This is about the same cost as setting up a successful reservation end-to-end. In fact, when there is an extreme load of requests the rejection is likely to happen earlier and the average admission control time becomes shorter. This method is robust and amenable to parallelization. A two-pass solution where we first check availability along the whole path before reserving would be faster when there are many rejections.

8.2 Pre-computed paths

One way of saving time for route calculations would be to cache the most recently used shortest-path trees. However, caching works best if there is locality in the reservation requests. As we currently have no real data indicating locality for resource reservations, this is left for further studies. Another solution is to pre-compute all shortest path trees. Then we can save most of the processing time for route calculations.

In figure 10, we find that admission control over 4.4 interfaces can be performed in approximately $40\mu s$ when routes are pre-computed. The admission decisions take $4.4 * 7 \approx 31\mu s$ and traversing the pre-computed route takes approximately $9\mu s$.

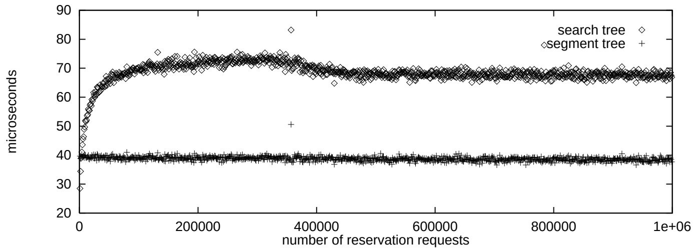

Figure 10: Admission speed using pre-computed paths within the LTU domain

8.3 Memory requirement

Storing all pre-computed shortest-path trees in our current implementation involves a total of n^2 nodes, where n is the number of routers in the domain. Each node in the shortest-path has two pointers to interface structures and one pointer to the parent node in the tree. In our example domain with 15 routers, this adds up to $15^2 * 12 = 2\,700$ bytes for shortest-path trees.

The 15 routers in the domain have a total of 48 interfaces. For each interface, there is an instance of the reservation data structure. Using segment trees, each instance requires 345 580 bytes when supporting reservations with five minutes granularity over 30 days. The total memory required for all 48 admission control data structures is therefore $48 * 345\,580 \approx 16\,200$ KB.

By measuring the memory consumption of the prototype agent we obtained a maximum of 20 400 KB in this topology (this involves more information about interfaces, runtime stack, debug strings associated with all objects, etc.).

9 The cost of request-reply transactions

In the experiments presented, admission requests are generated in the computer where the agent runs. In reality, an agent would receive reservation requests by using a reliable communication protocol. For this, we suggest using TCP for clients making frequent admission requests, and T/TCP [3] for clients making single admission requests. T/TCP is an extension to TCP for handling short reliable transactions efficiently. In T/TCP, clients can establish and close a virtual connection in the same packet as the request is sent. Also, the server can close its part of the virtual connection as the reply is returned. Therefore, the total delay from sending an admission request until receiving the reply equals the round-trip time plus the processing time of the admission request in the server (provided no packets are lost).

To evaluate the performance of an agent, we focus on the protocol processing time in the server only (i.e., we do not consider processing time in the client or round-trip delay). When admission requests are received from the network, the total time for processing an admission request must include local T/TCP protocol processing time. We have measured the T/TCP protocol processing time on our machine by running a T/TCP-capable dummy-server that immediately returns a reply to each incoming request. By running the program *tcpdump* [11], comparing the arrival time of requests with the departure time of replies, we have found the protocol processing times to be on average between $100\mu s$ and $110\mu s$, for data sizes between 400 bytes and 1 400 bytes.

Thus, with our current implementation of admission control, protocol processing time for admission requests is around twice as large as the time needed for performing end-to-end admission control in our example domain. However, the protocol processing time is independent of the size of the link-state routing domain, while admission control time is proportional to the path length.

10 Related Work

In [5], advance admission control is supported by partitioning network resources between advance reservations and immediate reservations. In our architecture, immediate admission control looks ahead into the advance reservation state to avoid pre-partitioning resources [17].

In [20], an overview of the field of advance reservations is given. In [16] , extensions for ST-2 to support advance reservations are presented. Our work provides advance reservations in a differentiated services Internet (i.e., in a datagram network).

In [1], an architecture that relies on topology-aware advance reservation servers is presented. The major difference compared to our architecture is that they use RSVP to setup per-flow reservation state in routers along the path when a session starts.

In [19], the theory of effective bandwidths is used to allow higher utilization by providing advance admission control with statistical guarantees. When resources are over-utilized, the most recently booked advance reservation is preempted.

In [8], a theoretical study of future reservations is performed. Their model includes functionality for finding time slots when resources are available and to make an incentive for video-on-demand users to receive movies at the same time (by multicast).

Within the Internet2 QBone project, a differentiated services test-bed is developed. Work on bandwidth brokers is carried out within a subgroup named the QBone Bandwidth Broker Advisory Council (QBBAC) [15]. In addition to our prototype, a number of independent prototypes are under development in this group. Among these are: the Merit, the BCIT, the UCLA, and the LBL/Chicago University bandwidth brokers [10]. The latter includes slot-based admission control that is similar to ours.

To our knowlege, none of these works have presented a thorough experimental evaluation of the cost for performing advance admission decisions in a real network topology.

11 Conclusion

We have evaluated two data structures for advance reservations, i.e., a segment tree and a binary search tree. The segment tree is faster and more indifferent to changing parameters such as average durations requested, insertion order, etc. The only drawback of the segment tree is that it requires more memory than the binary search tree. With the segment tree, an admission decision takes less than $7\mu s$ per hop when supporting reservations over a 30 day period with 5 minutes granularity.

We have also evaluated the time required for finding end-to-end paths in a link-state domain. Using on-demand route computations, the time is on average $20\mu s$ in our example domain. Using pre-computed routes and segment trees for admission control over each link in a topology with 15 routers, 28 transition networks and 64 stub networks, admission control takes approximately $40\mu s$. Thus, in this topology our prototype can perform around 25 000 end-to-end admission decisions per second. We have shown that protocol processing time in the server for receiving admission requests through T/TCP is substantial. Currently, the protocol processing time is more than $100\mu s$ per request.

The results show that it is feasible for an agent to manage per-link resource reservations and perform path-sensitive admission control in a link-state routing domain. The results also show that a general purpose computer is powerful enough to run the agent software.

References

[1] S. Berson, R. Lindell, and R. Braden. An architecture for advance reservations in the Internet. Technical report, USC Information Sciences Institute, July 1998.

[2] S. Blake et al. An architecture for differentiated services. RFC (Informational) 2475, IETF, December 1998.

[3] B. Braden. T/TCP – TCP extensions for transactions functional specification. RFC (Experimental) 1644, IETF, July 1994.

[4] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) – version 1 functional specification. RFC (Standards Track) 2205, IETF, September 1997.

[5] D. Ferrari, A. Gupta, and G. Ventre. Distributed advance reservation of real-time connections. In *Proceedings of NOSSDAV*, Lecture Notes in Computer Science, pages 15–26, Durham, New Hampshire, April 1995. Springer.

[6] GateD home page. http://www.gated.merit.edu/.

[7] R. Guerin, C. Partridge, and S. Shenker. Specification of guaranteed quality of service. RFC (Standards Track) 2212, IETF, October 1997.

[8] A. Hafid, G. V. Bochmann, and R. Dssouli. Quality of service negotiation with present and future reservations: A detailed study. *Computer Networks and ISDN Systems*, 30, 1998.

[9] D. Harrington, R. Presuhn, and B. Wijnen. An architecture for describing SNMP management frameworks. RFC (Standards Track) 2271, IETF, January 1998.

[10] G. Hoo, W. Johnston, I. Foster, and A. Ray. QoS as Middleware: Bandwidth Brokering System Design. Submitted to Eight IEEE Symposium on High Performance Distributed Computing.

[11] V. Jacobson, C. Leres, and S. McCanne. tcpdump, the protocol packet capture and dumper program.

[12] A. Mankin, F. Baker, B. Braden, S. Bradner, M. O'Dell, A. Romanow, A. Weinrib, and L. Zhang. Resource reservation protocol (RSVP) – version 1 applicability statement and some guidelines on deployment. RFC (Informational) 2208, IETF, September 1997.

[13] J. Moy. OSPF version 2. RFC (Standards Track) 2178, IETF, July 1997. (Obsoletes RFC1583).

[14] A. Nilsson, J. Chen, and S. Carlsson. An efficient data structure for advance bandwidth reservation on the Internet. Technical report, CSEE, Luleå University of Technology, 1998.

[15] QBone Bandwidth Broker Advisory Council home page. http://www.merit.edu/working.groups/i2-qbone-bb.

[16] W. Reinhardt. Advance reservation of network resources for multimedia applications. In *2nd International Workshop on Advanced Teleservices and High Speed Communication Architectures (IWACA'94)*, Heidelberg, Germany, October 1994.

[17] O. Schelén and S. Pink. Sharing resources through advance reservation agents. In *Proceedings of IFIP Fifth International Workshop on Quality of Service (IWQoS'97)*, New York, May 1997. Also: Resource Sharing in Advance Reservation Agents. In Journal of High Speed Networks, Special issue on Multimedia Networks, vol 7, no 3-4, 1998.

[18] O. Schelén and S. Pink. Resource reservation agents in the Internet. In *Proceedings of NOSSDAV*, Lecture Notes in Computer Science, Cambridge, United Kingdom, July 1998. Springer.

[19] D. Wischnik and A. Greenberg. Admission control for booking ahead shared resources. In *Proceedings of IEEE Infocom*, San Francisco, California, March 1998.

[20] L. Wolf, L. Delgrossi, R. Steinmetz, S. Schaller, and H. Wittig. Issues of reserving resources in advance. In *Proceedings of NOSSDAV*, Lecture Notes in Computer Science, pages 27–37, Durham, New Hampshire, April 1995. Springer.

[21] J. Wroclawski. Specification of the controlled-load network element service. RFC (Standards Track) 2211, IETF, October 1997.