Bell Laboratories

subject: **Lightweight Resource Reservation Signaling: Design, Performance and Implementation**
Work Project No. MA30034004
File Case 20564

date: **July 6, 2000**

from: **Ping Pan**
Dept. 10009669
HO 4F-506A
+1 732 332 6744
pingpan@bell-labs.com
10009669-000615-03TM

Henning Schulzrinne
Columbia University
Dept. Computer Science
NY, NY 10027
+1 212 939 7042
hgs@cs.columbia.edu

Technical Memorandum

Recent studies take two different approaches to admission control. Some argue that due scalability limitations, using a signaling protocol to set up reservations is too costly and CPU-intensive for routers. Instead, end users should apply various end-to-end measurement-based mechanisms to run admission control. Several other proposals have recommended to reduce the number of reservations in the network by using aggregation algorithms, and, thus, reduce the number of signaling messages and states.

We study the signaling cost factors, propose several solutions that achieve good performance with reduced processing cost, and evaluate an implementation of a lightweight signaling protocol that incorporates these solutions. First, we identify some the protocol design issues that determine protocol complexity and efficiency, namely the choice of a two-pass vs. one-pass reservation model, partial reservation, and the effect of reservation fragmentation. We also explore several design options that can speed up reservation setup and quickly recover from reservation fragmentation. Based on the conclusion of these studies, we developed a lightweight signaling protocol that can achieve good performance with low processing cost. We also show that with careful implementation and by using some of basic hashing techniques to manage flow states, we can support up to 10,000 flow setups per second (or about 300,000 active flows) on a commodity 700 MHz Pentium PC.

1 Introduction

The RSVP-IntServ model [1, 2] provides quality of service to individual *flows*. To enable such a service, network routers need to implement per-flow queueing and scheduling in the data plane, and per-flow reservation state management in the control plane. In a network where there are many flows, the processing overhead associated with real-time scheduling and queueing becomes non-negligible [3] . Furthermore, based on evaluations of several RSVP implementations [4, 5] , it was found that in some implementation, the per-flow processing overhead increases linearly with the number of flows. Based on these experiments, the scalability of the RSVP-IntServ model has been questioned.

It is important to realize that two scalability concerns arise, namely packet forwarding and signaling. Packet forwarding overhead is caused by maintaining queues for each "micro" (per-user) flow and assigning packets to each such queue. To reduce per-flow queueing overhead, several alternative architecture have been proposed, including the IETF DiffServ model [6] and the *dynamic packet state* [7] architecture. In the DiffServ model, routers simply implement a set of buffer management and priority-like queueing disciplines for each of a very small number of traffic classes, providing them with coarse-grain rate guarantees [8] .

However, even with DiffServ's superior data plane scalability, the network still needs to control admission. The first approach ([9, 10]) is based on a core stateless network, where no per-flow QoS state is maintained at network routers. The end host actively probes the network by sending probing packets at the data rate it would like to reserve, and admits user's flows based on the resulting packet loss or ECN marking [11] . A key assumption in this approach is that processing reservation messages at routers is expensive, therefore, admission control has to be delegated to end users.

In the second approach, admission control and QoS provisioning are supported inside the network. The second approach has two flavors, a distributed and a centralized model. Several proposals [12, 13, 14, 15, 16] have suggested a control message reduction approach of using *reservation aggregation*. In these designs, routers "sum" up the individual reservations at the network edge so that the total number of reservations at core routers is small. These approaches also acknowledge the importance of developing a hierarchical reservation system in the network to further reduce the reservation processing overhead. At the same time, in an effort to reduce or eliminate the involvement of routers during admission control, a centralized model based on bandwidth brokers [17] has been proposed. Here, bandwidth brokers are servers within the network that are responsible for admission control and resource management. The routers send flow information and resource usage data to the brokers, and the brokers send admission decisions to the routers.

To evaluate whether these approaches are necessary, we evaluate the factors that influence signaling cost and scalability The paper is organized as follows. In Section 2, we discuss several reservation signaling design issues. In Section 3, we evaluate some of the design alternatives through simulation. Section 4 outlines the implementation of a lightweight signaling protocol that incorporates the design ideas that we describe. Section 5 presents performance results of our implementation. We conclude in Section 6.

2 Reservation Signaling Issues

In this section we discuss some of the design issues related to reservation signaling, and use these considerations to motivate a set of design choices that will then be simulated in Section 3.

2.1 Scalability Factors

Some of the factors that strongly influence signaling scalability are:

Protocol complexity: Protocol complexity can be measured by the number of messages that routers need to process in order to complete one reservation, and the number of tasks that need to be scheduled during processing. Generally, there is a trade-off between end-user flexibility, and the processing complexity at routers.

QoS state management efficiency: This metric encompasses for example the time that is required to search, add or delete reservation states. To support a reasonably large number of sessions (or flows), CPU-intensive search algorithms, such as linear-search, have to be avoided.

Simplicity in configuration: This parameter is governed by the number of parameters that are required to setup a protocol on routers. The trade-off is between deployability and providing some "knobs" for fine tuning on routers.

Below, we evaluate several mechanisms that improve reservation performance while reducing a protocol's complexity. We will address the state management issue in Section 4.

2.2 One-Pass vs. Two-Pass Reservation

In previous work [5] , we proposed a lightweight reservation protocol called YESSIR. We claimed that it had two key features that could simplify reservation processing on routers, namely one-pass reservation and partial reservations. We reiterate these features here only as an example to motivate design alternatives. There are other light-weight reservation protocols such as SDP [18] .

The one-pass reservation model in the YESSIR proposal works as the following: Sender S initiates a reservation by sending a flowspec to all receivers D. The message that carries the flowspec propagates through the network toward the receivers. Each router along the way attempts to perform a resource reservation upon the reception of the flowspec. If there is an admission error, the router caches the flowspecs for future reservation retries, tags a notification to the flowspec message and passes the flowspec to the next router.

In comparison, RSVP employs a two-pass reservation model. In RSVP, an "offered" flowspec is first initiated by the sender S, and propagated along the routing path to receivers D. Each router along the way records the flowspec. The receivers get the "offered" flowspecs, adjust them to their needs, and propagate the resulting "requested" flowspecs back along the same routes to the senders. At each router, a local reconciliation must be performed between the "offered" and the "requested" flowspecs to create a reservation, and an appropriately modified "requested" flowspec is passed on. In case of reservation admission failure, the router terminates the reservation and returns an explicit error message back to the receivers. At the same time, the router keeps a copy of the failed flowspecs, and retries for reservation during the next refresh cycle. This latter process is referred as "killer reservation prevention". The failed flowspecs that routers keep for reservation retries are called "blockade states".

Receiver-oriented protocols such as RSVP allows for receiver diversity, at the expense of processing more messages. However, in the absence of intelligent packet filters, receiver diversity is not likely to be useful. (Both sender- and receiver-oriented protocols can support shared reservations, e.g., for multimedia audio conferences.)

2.3 Partial Reservation

We define the partial reservation as the following: for a reserving flow, $f_{a \to b}$, let $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ as the set of network links at the flow traverses, and $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}, m \leq n$ as the set of the network links that have made reservation for the flow.

if $\mathcal{R} = \mathcal{L}$

$f_{a \to b}$ is *fully* reserved.

else

$f_{a \to b}$ is *partially* reserved.

Both RSVP and YESSIR can result in partial reservations, although the mechanism that causes this is different. The distribution of partial reservation state also differs. In RSVP, a reservation request that fails admission control creates blockade state and proceeds no further. The corresponding reservation is left in place in nodes downstream (towards the receivers) of the failure point. In YESSIR, if a reservation request is denied, the reservation request still advances to the next hop. Thus, for the same network, YESSIR will create more reserved links than RSVP does.

Generally, the main reason for a reservation failure is the lack of sufficient resources, i.e., bandwidth or buffer space, to to accommodate the reservation at the time of the request. In a network with a large number of flows, reservations start and terminate at a high rate, causing the resource shortage likely to be temporary one. This suggests keeping the partial reservations instead of retrying the reservation from scratch at a later time.

The same rationale argues against the way that RSVP is handling partial reservations via blockade states. Just because a reservation has failed on one link along a reservation path does not mean the rest of the links will fail, too. In RSVP's killer reservation prevention mechanisms, the reservation process stops at the failure node. On the other hand, in YESSIR, each request tries to make reservations on as many links on its path as possible. This approach helps obtaining more resources for the requesting flow and potentially speeds convergence to a fully-reserved path. Section 3.2. will show simulation results to illustrate this point.

2.4 Reservation Retry Methods

Partial reservations do not provide the service quality that end users have originally requested. Hence, it is desirable for routers to "fill in" the missing reservations as soon as possible, since the tolerance for session set up delay is limited to a few seconds. We call the process of attempting to complete the reservations along the path as *retry*.

A simple mechanism combines retry and soft-state refresh. Since the routers periodically send flow states to the neighbors, the routers can retry the reservations at each refresh cycle. However, a refresh cycle can be quite lengthy. For example, the default refresh timer is 30 seconds for RSVP. This may be too long in applications such as Internet telephony where human users are waiting for the results and session life times are only a few minutes. Also, it has been suggested [19, 20] to dynamically adjust the refresh frequency based on network condition to improve the reliability in soft-state messaging. Hence, retrying failed reservations only at soft-state refresh intervals may not be good enough.

We propose a more aggressive method for partial reservation retries. Assume that a flow f_i needs to reserve b_i resources, while the reservable link resource is $\mathcal{B}$. At reservation setup time,

if $b_i < \mathcal{B}$
  update the reservation
  $\mathcal{B} \leftarrow \mathcal{B}$ - b_i
  Reserve(b_i)
else
  queue the request
  Enqueue($\mathcal{Q}$, b_i)
return

When a flow f_j with resource b_j is deleted or timed out, the router checks for the pending requests in queue $\mathcal{Q}$ and retries the reservations:

$\mathcal{B} \leftarrow \mathcal{B}$ + b_j
search through the queue
$b_x \leftarrow head[\mathcal{Q}]$
while $b_x \neq$ NIL and $b_x \leq \mathcal{B}$
do
  $\mathcal{B} \leftarrow \mathcal{B}$ - b_x
  Reserve(b_x)
  $b_x \leftarrow next[\mathcal{Q}]$
return

Thus, routers retry failed reservations as soon as extra resource becomes available. We refer to this scheme as *resource grabbing* and will demonstrate its effectiveness in Section 3.2.

Note that partial reservations and fragmentation (see below) are only likely to occur in heavily loaded networks or under overload conditions. However, this is exactly when reservation is needed at all – best effort service works fine in an under-utilized network.

2.5 Reservation Fragmentation

Partial reservation can lead to reservation fragmentation, where a large number of flows all have partial QoS, but all with unacceptable quality. An analogy to this is the deadlock problem in operating systems, where multiple processes try to access the same set of resources, and are all waiting for others to release them first. Since no process is willing to release the resource, a deadlock occurs.

In case of partial reservation, if all partially reserved flows refuse the give up their network resource, then no flow will get adequate resource. Section 3.3 shows the effect of reservation fragmentation[1].

Generally, common solutions to resolve deadlock include:

Preemption: A flow with high priority can take resources away from lower-priority flows holding these resources.

Rollback: All flows withdraw their partial reservations, and re-request at some random time later.

Suspend misbehaving flows: Flows that have failed their end-to-end reservation attempt too many times are simply ignored by routers, leaving resources for other flows.

The first two solutions require cooperation from end users, and thus more messaging between routers and end users. Plus, they do not prevent "impolite" users from persistently asking for reservations and obtaining as many network resources as they can. We plan to compare end-user reservation preemption and rollback in the future. Here, we present a simple algorithm that allow routers to limit the number of retries. It is based on the *resource grabbing* algorithm in Section 2.4. We define a threshold, $\mathcal{T}$, as the maximum number of retries that a flow can exercise during its entire duration at a single router. Each flow f_i needs to maintain a retry count c_i in addition to the reservation amount b_i.

During reservation and retry process:

if $b_i < \mathcal{B}$
    update the reservation
    $\mathcal{B} \leftarrow \mathcal{B}$ - b_i
    Reserve(b_i)
else
    queue the request and increment retry count
    Enqueue($\mathcal{Q}$, b_i)
    $c_i \leftarrow c_i + 1$
return

After a flow f_j with resource b_j has been deleted, the router does the following:

[1]Note, however, that the analogy is not complete. An operating system task can make no progress as long as it is missing one resource, while a flow may decide to "risk" the QoS degradation at a small number of routers, in the hope that the admission control mechanism is conservative and that there is enough best-effort bandwidth available there.

$\mathcal{B} \leftarrow \mathcal{B} + b_j$
search through the queue
$b_x \leftarrow head[\mathcal{Q}]$
while $b_x \neq$ NIL and $b_x \leq \mathcal{B}$ and $c_x \leq \mathcal{T}$
do
    $\mathcal{B} \leftarrow \mathcal{B}$ - b_x
    Reserve(b_x)
    $b_x \leftarrow next[\mathcal{Q}]$
return

By adjusting the value of $\mathcal{T}$ on routers during congestion, the fragmentation effect can be reduced. Here, we only limit the retry attempts during the resource grabbing process; flows can continue to retry failed reservations every refresh cycle.

3 Performance

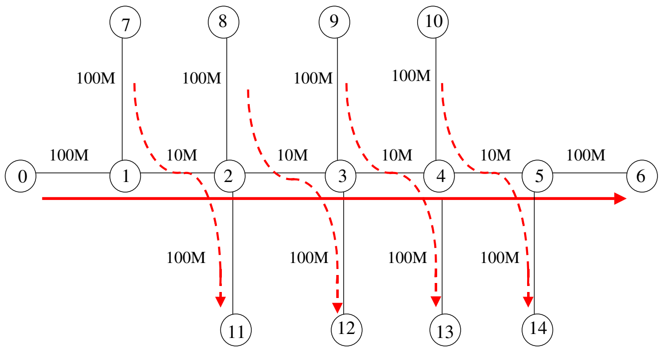

Figure 1: The network topology used in the simulation.

3.1 Simulation Methodology

To evaluate the design ideas described in Section 2, we used the *ns* simulator and its RSVP module, and extended it to support partial reservation functionality required for our experiments.

Figure 1 shows a 15-node simulation network topology. Nodes 1, 2, 3, 4 and 5 are backbone nodes, and the remainder are end systems. All backbone links have 10 Mb/s bandwidth and 20 ms propagation delay; all access network links have 100 Mb/s bandwidth and a propagation delay of

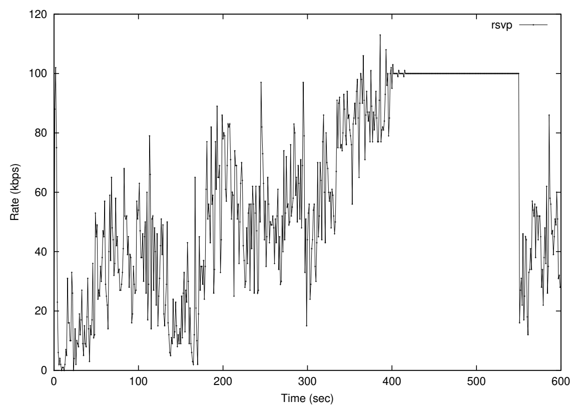

Figure 2: **Regular Load:** RSVP reservation; reservation completed after 150 s and 5 tries.

10 ms. Network links are reliable. We assume that each link has a high-priority queue reserved for reservation messages so that they are never lost. Up to 50% of the link bandwidth is reservable.

In the simulations that follow, network nodes 7, 8, 9 and 10 generates best-effort data flows as well as reserved data flows to nodes 11, 12, 13 and 14, respectively. All data packets are 125 bytes long. The best-effort data flows are modeled as exponential on/off traffic source, with on-time 1 s, off-time 0.5 s, a burst rate of 500 kb/s and an exponentially distributed flow duration with a mean of 150 s. The reservation flows are all CBR traffic with rate r of 100 kb/s and a token bucket size B of 5,000 bytes. We can create various network congestion conditions by adjusting the number of best-effort and reservation flows.

We assess the effectiveness of different reservation algorithms by monitoring a CBR flow from node 0 to node 6 traversing several congested links. In each simulation experiment, node 0 starts transmitting a 100 kb/s flow at time 0 to node 6. 250 s later, node 0 then tries to reserve resources for the flow. The reservation session lasts 300 s. We monitor this test flow at node 6 by capturing the data rate received. When the end-to-end reservation has been completed, we see a fixed rate of 100 kb/s (that is, a flat line on the traffic trace diagrams).

Data for the first 50 seconds in each simulation are discarded to obtain steady-state result. Each simulation has been run several times with different random seeds.

3.2 Basic Scenario

In a first experiment, labled "regular load" in the figures, we created a mildly congested network with 27 best-effort flows and 85 reservation sessions in the background. Since the total number of reservation sessions exceeds what the backbone routers can handle, we expect to see many reservation rejections, where rejected flows wait and retry.

All reservation protocols tested are soft-state based with a 30-second average refresh interval. To avoid synchronization, refresh intervals are randomly varied between 21 and 39 s.

Figure 2 shows the packet rate received at node 6 in a 600 s simulation. All nodes in the network use RSVP for reservation. A rejected flow can only retry for the reservation at the next refresh cycle. The test flow takes about 150 seconds and 5 tries to complete the reservation.

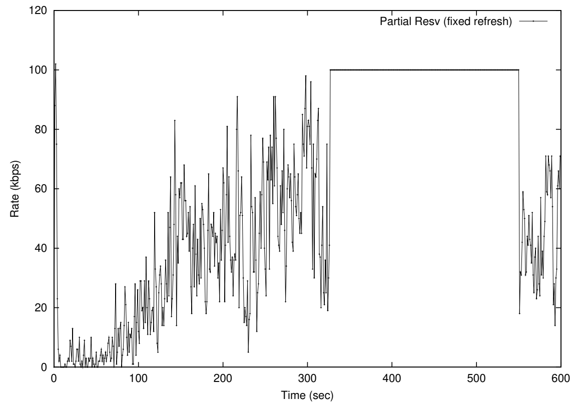

Figure 3: **Regular Load:** One-pass reservation without grabbing; reservation completed after 77 s and 3 tries.

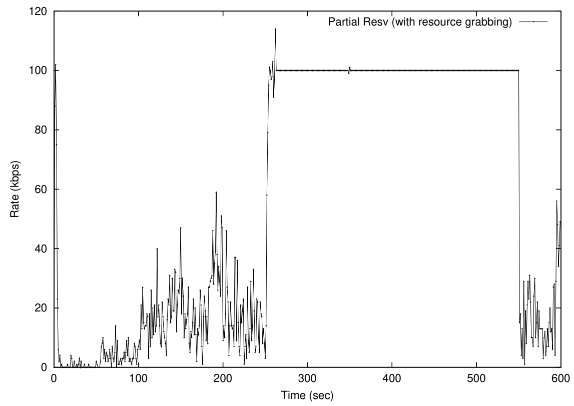

Figure 4: **Regular Load:** one-pass reservation with resource grabbing; reservation completed after 12 s and 19 tries.

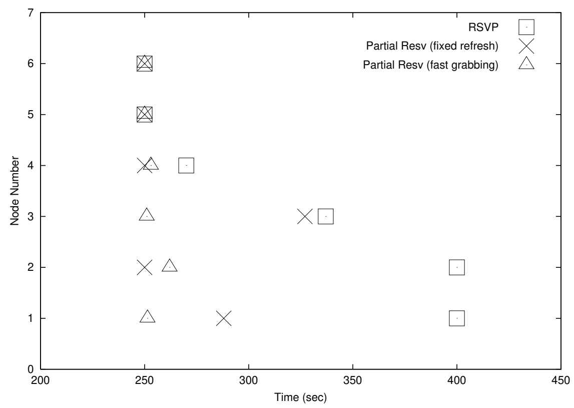

Figure 5: **Regular Load:** Reservation sequence for RSVP, one-pass reservation with refresh, and one-pass reservation with refresh plus resource grabbing. The ordinate shows the node number in the simulation network of Figure 1.

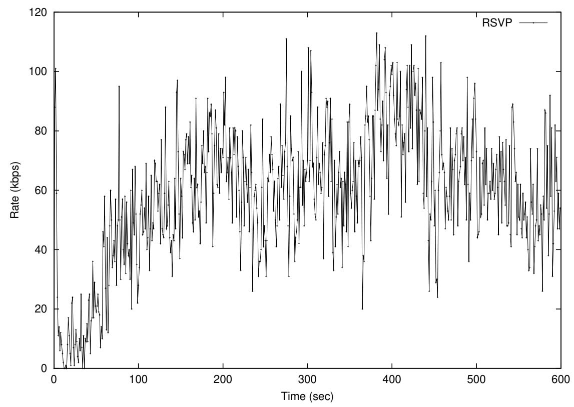

Figure 6: **High Load:** RSVP; no reservation made in 10 tries.

We then ran the same identical testing scenario with an one-pass reservation mechanism that uses soft-state refresh to retry the failed reservations. As shown in Figure 3, the reservation completes after 77 seconds and 3 tries.

Figure 4 shows a scenario where all the nodes use one-pass soft-state reservation and, in particular, the resource grabbing mechanism that we have described in Section 2.4. The testing flow reservation takes only 12 seconds to complete, but retries 19 times.

To study the reservation sequence, we collected reservation failure and success data from the backbone nodes. Figure 5 shows how the flow had completed the reservation in three testing scenarios above.

Using RSVP, the flow received reservations from node 5 and 6 during its first reservation attempt, but was rejected at node 4. At the next refresh cycle, the flow made the reservation on node 4, but was rejected from its immediate upstream node, node 3. It took two refresh cycles for the flow to eventually get a reservation from node 3. During the fifth refresh cycle, the flow made reservation from node 2 and 1, and thus completed the reservation.

Though, due to blockade states, the flow does not have to start reservations from scratch after each reject, it takes a long time for the flow to make its way toward the sender almost one hop at a time. In comparison, the one-pass reservation scheme can perform much better. At the reservation initiation time, the request message passes through all the nodes on the reservation path, and tries to make reservation on each node. As shown in the figure, the flow made the reservations on nodes 2, 4, 5 and 6 during its first reservation attempt. It took two more refresh cycles to complete the reservation.

Given that the background flows come and leave frequently, the reservation scheme armed with resource grabbing mechanism performed the best. It completed the reservation in 12 seconds, and retried reservations 19 times in total on all 6 nodes during this period.

3.3 High Load Condition: The Effect of Fragmentation

To evaluate the fragmentation effect on the network, we increased the number of background reservation flows to 120, that is to request 240% more resources than what the network can provide.

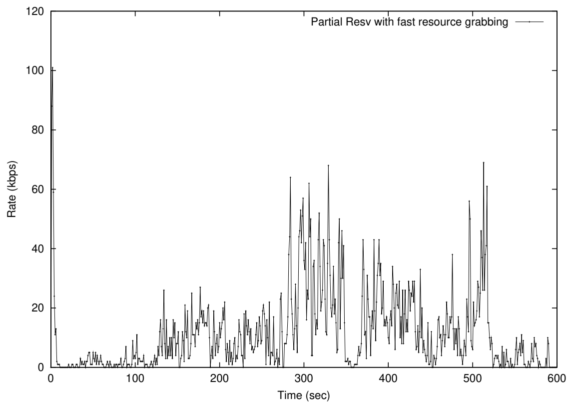

Figure 7: **High Load:** One-pass reservation with resource grabbing; reservation made in 184 tries.

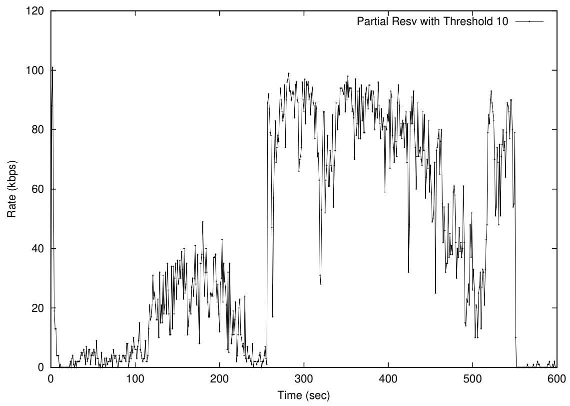

Figure 8: **High Load:** One-pass reservation, resource grabbing and a retry threshold of 10. No reservation after 46 tries.

Figure 6 and 7 show the received rate for the test flow that uses RSVP, and one-pass reservation with the resource grabbing algorithm, respectively. Neither protocol could make the end-to-end reservation. In particular, the one-pass reservation flow received far less user data than the one that uses RSVP. An explanation is that, with the resource grabbing algorithm, all reservation flows in the network try to take as much resource as possible. As a result, few flows get the full reservations. Since there was no reservable bandwidth left for best-effort traffic to share, all user flows suffer.

3.4 High Load Condition: Fragmentation Recovery

Here, we have applied the threshold algorithm that we had defined in Section 2.5. We ran the simulation with one-pass reservation with the resource grabbing algorithm on all network nodes with different a retry threshold value $\mathcal{T}$.

Figures 8, 9, 10 and 11 show the results with values for $\mathcal{T}$ of 10, 8, 3, and 1, respectively.

Table 1 collects the total number of reservation retries from the backbone nodes.

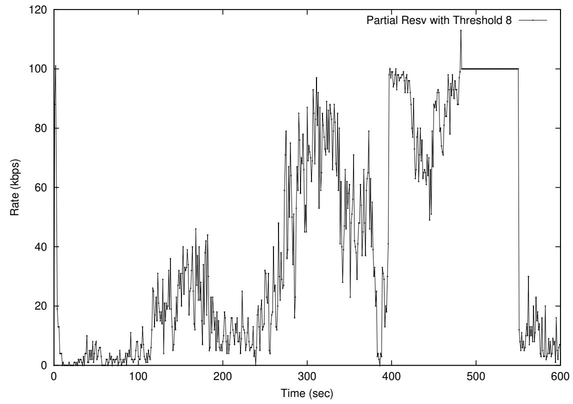

Figure 9: **High Load:** One-pass reservation, resource grabbing and a retry threshold of 8; reservation completed after 232 s and 32 tries.

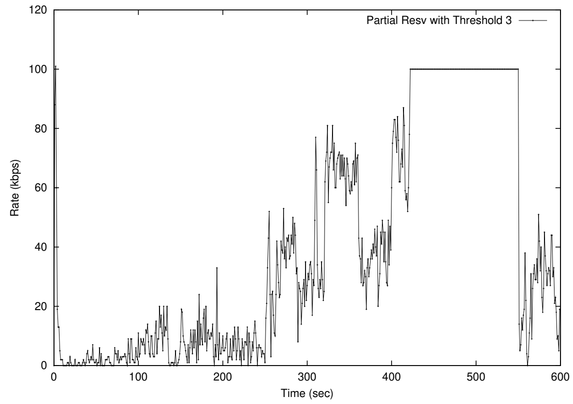

Figure 10: **High Load:** One-pass reservation with resource grabbing and a retry threshold of 3; reservation completed after 172 s and 14 tries.

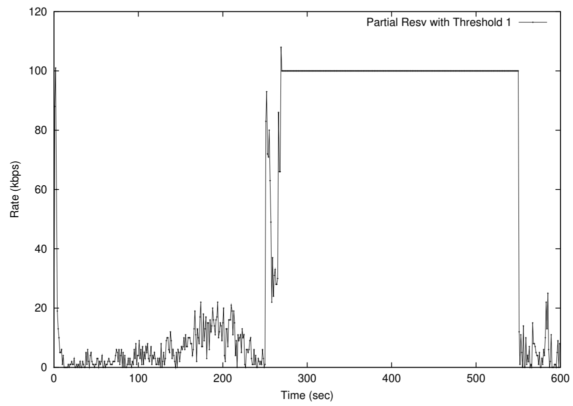

Figure 11: **High Load:** One-pass reservation with resource grabbing and a retry threshold of 1; reservation completed after 19 s and 1 retry.

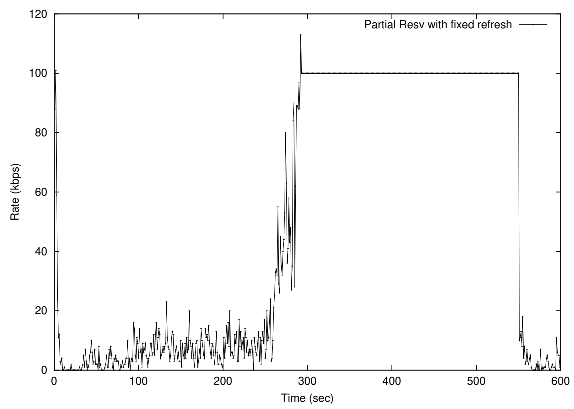

Figure 12: **High Load:** One-pass reservation; reservation completed after 43 s and 3 tries.

Description	# of retries	Test flow data
One pass, $\mathcal{T} = \infty$	28,314	Figure 7
One pass, $\mathcal{T}$=10	8,534	Figure 8
One pass,, $\mathcal{T}$=8	7,137	Figure 9
One pass, $\mathcal{T}$=3	4,382	Figure 10
One pass, $\mathcal{T}$ = 1	2,685	Figure 11
One pass, no grabbing	2,588	Figure 12
RSVP, fixed refresh	3,409[2]	Figure 6

Table 1: Number of retries in the network, measured for a 550 s simulation duration.

With a proper threshold value, the user flow can successfully make the reservation in a highly congested network. From our simulation, the best scenario is the one with $\mathcal{T}$ = 1 (Figure 11). The worst scenario is the one with a higher threshold, $\mathcal{T}$ = 10 (Figure 8).

We suspect that with lower threshold number, the flows are less aggressive to retry for the link resource, and therefore allow other flows to complete their reservations. This conclusion is supported by the results collected in Table 1.

Does it mean that reservation retry would perform the best with zero threshold (that is, only rely on the soft-state refreshes to perform reservation retry)? Figure-12 shows the simulation results of using only fixed refresh, and is not as good as the one with resource grabbing ($\mathcal{T}$ = 1). In the previous section, we had also observed that in a mildly congested network, resource-grabbing improves the reservation retry performance.

RSVP is also less aggressive in grabbing resource with its blockade state algorithm, then why does it perform so poorly? From our collected data, out of 3,409 reservation retries, there were only 739 successful flows in the 550-second simulation interval. Since the blockade states make partial reservations only on the nodes that are downstream from the failure node, RSVP flows thus receive resources from the network, and thus perform poorly.

[2] We have also monitored the total number of new RESV messages being received at the end nodes, which is 739. This is the same as the total number of successful RSVP reservations.

This leads to reasonable conclusion that, in using one-pass reservation protocol, there is a trade-off between the ability to obtain resource quickly, and the likelihood of causing reservation deadlocks.

4 Implementation Overview

We introduce a lightweight reservation protocol implementation for routers, in order to evaluate the performance with a large number of reservations. The implementation is based on the following considerations:

A simplified signaling protocol: From the studies that we have conducted in the previous sections, YESSIR qualifies as a lightweight signaling protocol.

An efficient state management scheme: The implementation needs to be able to support thousands of flows efficiently. At the same time, the algorithms that we choose should be simple and generic.

YESSIR is a simple signaling protocol, and is designed to provide resource reservation to real-time flows that use RTP [21] . It has two operating modes: explicit, and measurement-based. In the explicit reservation mode, YESSIR piggybacks traffic flowspecs in RTCP messages. Upon reception, routers make the reservation according to the flowspecs. In the measurement-based mode, the routers simply intercept the RTCP Sender Report messages, and make reservations based on the traffic statistics and timing information provided in the messages.

Since RTP is an end-to-end protocol, to allow the routers to intercept and process reservation requests, YESSIR uses the router alert option [22] in the IP header.

4.1 Kernel Extension

Current UNIX system have no clean mechanism to intercept IP option packets, including router alerts, through the socket interface. Thus, we designed and developed a new socket family, `PF_IPOPTION`, on FreeBSD. With the new socket family, it is easy to capture RTCP messages through a socket:

```
int sock, on = 1;
sock = socket (PF_IPOPTION, SO_RAW, IPOPT_RA);
setsocketopt(sock, IPOPT_IP, IP_RECVRTCP,
                &on, sizof(on));
```

We optimized the kernel code to speed the delivery of IP option messages [23] .

4.2 State Management

We used hash tables to manage the reservation states in our implementation. The motivation behind using hashing comes from the observation that any RTP session in the network can be *uniquely* identified by its IP source and destination addresses, *SA*, *DA*, and its UDP source and destination port numbers, *SP*, *DP*. In our implementation, the hash key for an RTP flow f_i is k_i = SA_i + DA_i + SP_i + DP_i. Our goal was to support up to 1,000 flows efficiently, so we had selected the hash table size, M, to be 1,537 to reduce the chance of hash collisions [24] . To solve the potential hash collision problem, we put all the flow entries that hash to the same slot in a linked list.

Our hash function is

$$h(k_i) \leftarrow k_i \bmod M.$$

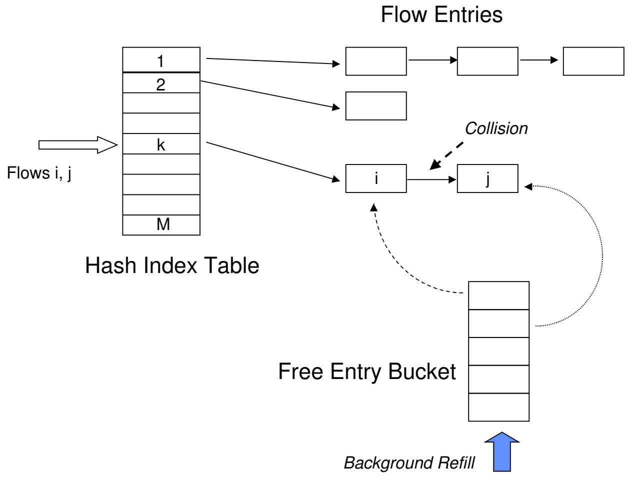

Figure 13: An example of reservation state management in our implementation

Figure 13 illustrates the reservation state handling in our implementation. There are three tables: hash index table, free entry bucket and flow entries. When a reservation request for flow f_i is received, we perform the hash function to find a hash slot, k, in the hash index table. After getting a flow entry from the free entry bucket, we copy the reservation data into the entry, and insert it into the linked list that is hanging off the hash slot, k.

A collision occurs if a new flow f_j arrives and hashes to the same slot as f_i. We simply insert the new entry into k's list behind the flow entry for f_i. Obviously, too many collisions will cause poor performance. (More sophisticate dynamic hashing schemes can limit the depth of the linked list.)

To improve memory usage and make easy for debugging, we have designed a free entry bucket (FEB) table. At system initiation time, we pre-allocate a small number of entries into the FEB. During flow processing, if the number of free entries is less than a threshold number, we will allocate another chunk of entries into the FEB. If there are too many free entries, the process will free the extra entries.

The hash table and memory management take about 1,400 lines of C code to implement.

4.3 Reservation Processing

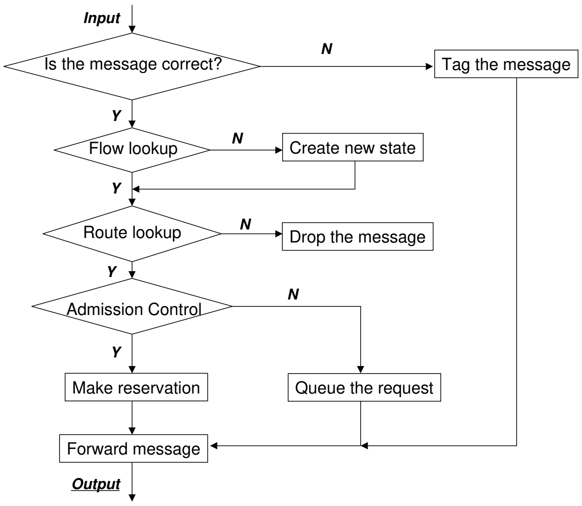

Figure 14: Reservation processing flowchart.

Figure 14 shows the reservation processing sequence at routers. If there is an admission control failure, we queue the request for future retries. Except in case of routing failure, we always forward the reservation message.

5 Experimental Results

We have implemented YESSIR on FreeBSD[3]. The latest version of the YESSIR implementation requires about 6,000 lines of C.

We tested and measured the implementation on a 700 MHz Pentium PC with several Ethernet interfaces. We have modified the author's rtptools package to generate RTCP messages with IP Router Alert option from end users.

We first examined the efficiency of the QoS state management with hashing. Provided that the hash table size is 1,537, we expect that the flow entry searching and creating time would be more or less the same when there are less than 1,000 flows in the system. As the number of the flows increases, more hash collisions will occur.

[3]The source and object code for the PF_IPOPTION kernel extension and YESSIR are available at http://www.cs.columbia.edu/~pingpan/software.

Number of flows in the router	flow build time (μs)
50	6.4 ± 1.35
100	6.3 ± 1.25
200	6.3 ± 0.95
500	6.2 ± 1.32
800	6.4 ± 0.97
1,000	6.1 ± 0.99
2,000	7.3 ± 1.25
5,000	7.1 ± 1.20
8,000	8.1 ± 0.99
10,000	8.0 ± 1.56

Table 2: Hash table performance with collision resolution by chaining. The hash table size is 1537.

To verify this, we had generated 10,675 new reservation flows from multiple sources. On the router, we had recorded the flow entry creation time on each flow. To ensure measurement accuracy, we shut off the refresh timers during the test. The results are shown in Table 2. As expected, the processing time is constant if the number of flows is below 1,000 and gradually rises above that threshold.

Description	Processing time (in μs)
Message integrity checks	6.6 ± 0.52
Flow lookup/creation	6.5 ± 0.53
Route lookup	27.1 ± 1.29
Admission control (call for reservation)	6.1 ± 0.57
Kernel I/O	97.8 ± 8.24

Table 3: Timing for a lightweight reservation protocol implementation on a 700 MHz Pentium.

We measured the time for processing a one-pass reservation request message. The measurement was taken both at user space and in the kernel. During the measurement, we generated YESSIR messages used for measurement-based reservation. As shown in Table 3, the overall processing time in the user space is about 46 μs. However, the time between when the packet is received from the device driver until it is sent to the device driver in the kernel is 98 μs, i.e., approximately half the processing time is spent in the kernel.

6 Conclusion

We have investigated the design and performance of reservation signaling through simulation. To gain a better understanding of reservation costs, we have implemented a lightweight reservation protocol on the FreeBSD platform.

With a detailed comparison of the reservation performance achieved from two-pass and one-pass reservation protocols with different parameters and network conditions, we verified that a reservation protocol that is based on one-pass reservation model has better performance and lower processing cost, comparing with a regular two-way signaling protocol. Routers can employ resource grabbing to help speed up reservation convergence. To recover from the fragmentation effect of partial reservations, routers need to control the retry process of the failed flows. One such mechanism is to limit the number of reservation retries on failed flows.

From evaluating our implementation results, we believe that with proper protocol design and implementation, routers can support a large number of user flows, while providing admission control. Actually, the processing bottleneck may not come from the signaling protocols after all, instead it may be from the limitations in router operating systems. Thus, a dedicated processor with a light-weight operating system may be helpful.

Finally, we believe that, the control-plane scalability problem is *not* an issue of the number of the flows that routers can process, but rather the number of the flows that the network providers can manage for authorization, accounting and billing. We think that much future research is needed on understanding network resource manageability.

7 Acknowledgements

Andreas Rosblom helped simulate some of the earlier test cases.

References

[1] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation protocol (RSVP) – version 1 functional specification," Request for Comments 2205, Internet Engineering Task Force, Sept. 1997.

[2] R. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture: an overview," Request for Comments 1633, Internet Engineering Task Force, June 1994.

[3] T. cker Chiueh, A. Neogi, and P. Stirpe, "Performance analysis of an RSVP-capable router," in *Proc. of 4th Real-Time Technology and Applications Symposium*, (Denver, Colorado), June 1998.

[4] M. Karsten, "Design and implementation of rsvp based on object-relationships," in *Proceedings of Networking 2000*, (Paris, France), May 2000.

[5] P. P. Pan and H. Schulzrinne, "YESSIR: A simple reservation mechanism for the Internet," in *Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSS-DAV)*, (Cambridge, England), pp. 141–151, July 1998. also IBM Research Technical Report TC20967.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated service," Request for Comments 2475, Internet Engineering Task Force, Dec. 1998.

[7] I. Stoica and H. Zhang, "Providing guaranteed service without per flow management," *ACM Computer Communication Review*, vol. 29, pp. 81–94, Oct. 1999.

[8] R. Guerin, L. Li, S. Nadas, P. Pan, and V. Peris, "The cost of QoS support in edge devices: An experimental study," in *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, (New York), Mar. 1999.

[9] L. Breslau, E. Knightly, S. Shenker, I. Stoica, and H. Zhang, "Endpoint admission control: Architectural issues and performance," in *Proceedings of Sigcomm 2000*, (Stockholm, Sweden), Aug. 2000.

[10] V. Elek, G. Karlsson, and R. Ronngren, "Admission control based on end-to-end measurements," in *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, (Tel Aviv, Israel), Mar. 2000.

[11] K. Ramakrishnan and S. Floyd, "A proposal to add explicit congestion notification (ECN) to IP," Request for Comments 2481, Internet Engineering Task Force, Jan. 1999.

[12] P. Pan, E. Hahne, and H. Schulzrinne, "The border gateway reservation protocol (BGRP) for tree-based aggregation of inter-domain reservations," *Journal of Communications and Networks*, June 2000.

[13] R. Guerin, S. Herzog, and S. Blake, "Aggregating RSVP-based QoS requests," Internet Draft, Internet Engineering Task Force, Nov. 1997. Work in progress.

[14] Y. Bernet, R. Yavatkar, P. Ford, F. Baker, L. Zhang, M. Speer, B. Braden, B. Davie, J. Wroclawski, and E. Felstaine, "A framework for integrated services operation over diffserv networks," Internet Draft, Internet Engineering Task Force, May 2000. Work in progress.

[15] F. Baker, B. Davie, F. L. Faucheur, and C. Iturralde, "RSVP reservations aggregation," Internet Draft, Internet Engineering Task Force, Mar. 2000. Work in progress.

[16] O. Schelen and S. Pink, "Aggregating resource reservation over multiple routing domains," in *Proc. of Fifth IFIP International Workshop on Quality of Service (IwQOS)*, (Cambridge, England), June 1998.

[17] K. Nichols, V. Jacobson, and L. Zhang, "A two-bit differentiated services architecture for the internet," Request for Comments 2638, Internet Engineering Task Force, July 1999.

[18] W. Almesberger, J.-Y. L. Boudec, and T. Ferrari, "Scalable resource reservation for the internet," in *Proc. of IEEE Conference Protocols for Multimedia Systems – Multimedia Networking (PROMS-MmNet)*, (Santiago, Chile), Nov. 1997. Technical Report 97/234, DI-EPFL, Lausanne, Switzerland.

[19] P. Pan and H. Schulzrinne, "Staged refresh timers for RSVP," in *Proceedings of Global Internet*, (Phoenix, Arizona), Nov. 1997. also IBM Research Technical Report TC20966.

[20] P. Sharma, D. Estrin, S. Floyd, V. Jacobson, P. Sharma, D. Estrin, S. Floyd, and V. Jacobson, "Scalable timers for soft state protocols," in *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, (Kobe, Japan), p. 222, Apr. 1997.

[21] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: a transport protocol for real-time applications," Request for Comments 1889, Internet Engineering Task Force, Jan. 1996.

[22] D. Katz, "IP router alert option," Request for Comments 2113, Internet Engineering Task Force, Feb. 1997.

[23] P. Pan and H. Schulzrinne, "Pf_ipoption: A kernel extension for ip option packet processing," in *Bell Labs Technical Memorandum 10009669-02TM*, June 2000.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

Bell Laboratories

Document Cover Sheet for Technical Memorandum

Title: Lightweight Resource Reservation Signaling: Design, Performance and Implementation

Authors	Electronic Address	Location	Phone	Company (if other than Lucent–BL)
Ping Pan	pingpan@bell-labs.com	HO 4F-506A	+1 732 332 6744	
Henning Schulzrinne	hgs@cs.columbia.edu	NY, NY 10027	+1 212 939 7042	Columbia University

Document No.	Filing Case No.	Work Project No.
10009669-000615-03TM	20564	MA30034004

Keywords:
Kernel, Socket

MERCURY Announcement Bulletin Sections

CMM– Communications CMP– Computing

Abstract

Recent studies take two different approaches to admission control. Some argue that due scalability limitations, using a signaling protocol to set up reservations is too costly and CPU-intensive for routers. Instead, end users should apply various end-to-end measurement-based mechanisms to run admission control. Several other proposals have recommended to reduce the number of reservations in the network by using aggregation algorithms, and, thus, reduce the number of signaling messages and states.

We study the signaling cost factors, propose several solutions that achieve good performance with reduced processing cost, and evaluate an implementation of a lightweight signaling protocol that incorporates these solutions. First, we identify some the protocol design issues that determine protocol complexity and efficiency, namely the choice of a two-pass vs. one-pass reservation model, partial reservation, and the effect of reservation fragmentation. We also explore several design options that can speed up reservation setup and quickly recover from reservation fragmentation. Based on the conclusion of these studies, we developed a lightweight signaling protocol that can achieve good performance with low processing cost. We also show that with careful implementation and by using some of basic hashing techniques to manage flow states, we can support up to 10,000 flow setups per second (or about 300,000 active flows) on a commodity 700 MHz Pentium PC.

Pages of Text 1 Other Pages 21 Total 22
No. Figs. 14 No. Tables 3 No. Refs. 24

Mailing Label

Complete Copy	**Cover Sheet Only**
Directors in 112, 113, 118	Members of 112, 113, 118

Future Lucent Distribution Release to any Lucent employee (excluding contract employees)

Author Signatures

Ping Pan	Henning Schulzrinne	

Organizational Approvals

Tom La Porta	Krishan K. Sabnani	

For Use by Recipient of Cover Sheet:

Computing network users may order copies via the *library -1* command;
for information, type *man library* after the UNIX® system prompt.

Otherwise:
Enter your Company ID number. ____________________
Return this sheet to any ITDS location.

Internal Technical Document Service

( ) AK 2H-28	( ) IH 7M-103	( ) DR 2F-19	( ) NW-ITDS
( ) ALC 1B-102	( ) MV 3L-19	( ) INH 1C-114	( ) PR 5-2120
( ) CB 3O-2011	( ) WH 3E-204	( ) IW 2Z-156	
( ) HO 4F-112		( ) MT 3B-117	