# Processing Overhead Studies in Resource Reservation Protocols

Ping Pan[a] and Henning Schulzrinne[b]

[a]Juniper Networks, 1194 N. Mathilda Avenue, Sunnyvale, CA 94089, U.S.A.
pingpan@juniper.net

[b]Computer Science Department, Columbia University, New York, NY 10027, U.S.A.
schulzrinne@cs.columbia.edu

We study the signaling cost factors from two aspects: reservation retry process (or how to recover from reservation blocking), and one-pass signaling. Through simulation, we discover that the reservation retry mechanism used in RSVP is not only process intensive, but also the reservations are slow to converge. We thus explore several design options that can speed up reservation setup and quickly recover from partial reservations. To gain a better understanding of reservation overhead, we have implemented a lightweight one-pass reservation protocol, YESSIR. We show that with careful implementation and by using some of basic hashing techniques to manage flow states, we can achieve good performance with low processing cost. Our YESSIR implementation 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 queuing 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 queuing becomes non-negligible [3]. Furthermore, based on evaluations of several RSVP implementations [4], it was found that in some implementation, the per-flow processing overhead increases linearly with the number of flows. Therefore, 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 queuing overhead, several alternative architecture have been proposed, including the IETF DiffServ model [5], where routers can simply implement a set of buffer management and priority-like queuing disciplines for each of a very small number of traffic classes, providing them with coarse-grain rate guarantees [6].

However, even with DiffServ's superior data plane scalability, the network still needs to control admission. The first approach 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, for instance. A key assumption in this approach is that processing reservation messages at routers is too *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 [7,8] 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. In an effort to reduce or eliminate the involvement of routers during admission control, a centralized model based on *bandwidth brokers* [5] has been proposed, where 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 back to the routers.

Here, we notice that the common thread in all these proposals is that they are aimed to move away from the original RSVP framework and concentrate on avoiding or reducing signaling overhead at routers. So what is the signaling overhead at routers? In this paper, we investigate some of the design issues that effects reservation signaling performance. To overcome these deficiencies, we introduce the design and the implementation of a lightweight reservation protocol, YESSIR, that can process a large number of reservations at run-time.

## 2. Reservation Signaling Issues

### 2.1. RSVP

RSVP (ReSerVation Protocol) was the first widely developed [9] resource reservation protocol in the Internet. It was originally designed as the signaling protocol to support Integrated Services (IntServ), where end users can trigger RSVP to establish simplex reservation "flows" in the network. Each flow is defined as a source-destination pair, and each destination can be either a unicast user, or a multicast group.

RSVP is a two-pass signaling protocol, that it takes two messages, PATH and RESV, to complete a reservation. To setup a reservation between source $S$ and destination $D$, $S$ first adds a reservation description or *flowspec* in a PATH message. The PATH messages are periodically sent toward $D$. Each router along the way records the flowspec from $S$. Upon reception of a PATH, $D$ adjusts the flowspec to its needs, and puts a modified flowspec, in a RESV message. RESV messages are periodically sent toward $S$ along the path traveled by the PATH messages. At each router, a local reconciliation must be performed on the flowspec's from $S$ and $D$. If the router can accommodate the resulting flowspec, a reservation is made and the RESV is passed on. Here, we notice that RSVP's two-pass reservation technique can be processing intensive if there are many flows in the networks.

During RESV message processing, if a router cannot meet the required reservation, it returns an explicit error message, back to $D$. At the same time, the router has to keep a copy of the failed flowspec from $D$, and retry for reservations during the next refresh cycle.

This latter process is referred as *killer reservation prevention* since the failed reservation request must never deny service to new requests. The failed flowspecs that routers keep for reservation retries are called *blockade states*.

### 2.2. YESSIR

In a previous work [10], we proposed a lightweight reservation protocol called YESSIR. The new protocol has two key features that could simplify reservation processing on routers, namely one-pass reservation and allowing partial reservations.

YESSIR establishes a reservation 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.

Since the one-pass reservation model in the YESSIR proposal requires only one message to setup a reservation, it simplifies the processing sequence at routers, and can ultimately improve router's signaling performance. We will detail YESSIR's message handling in Section 4. In the remaining section, we will concentrate on the other performance bottleneck, that is, the handling of reservation rejection (or blocking) at routers.

### 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 $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$, $m \le n$ 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.

Note that partial reservations are only likely to occur in resource-constraint 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.

Both RSVP and YESSIR can result in partial reservations due to admission failures, 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 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 3.1 will illustrate this point with simulation.

### 2.4. Reservation Retry

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. Hence, retrying failed reservations only at soft-state refresh intervals may not be good enough.

A more aggressive method is to have routers to retry failed reservations as soon as extra resource becomes available. However, as we will identify in Section 3.2, if every router aggressively seeking resources, it will only create more partial reserved flows in the network.

Partial reservation can lead to a situation, 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[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.

[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.

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 very simple algorithm that allow routers to limit the number of retries.

Assume that a flow $f_i$ needs to reserve $b_i$ resources, while the reservable link resource is $\mathcal{B}$. 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 also to maintain a retry count $c_i$. 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 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:

$\mathcal{B} \leftarrow \mathcal{B} + b_j$
▷ 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, the partial reservation effect can be reduced. Here, we allow flows to continue to retry failed reservations at every refresh cycle.

## 3. Simulation Verification

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 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.

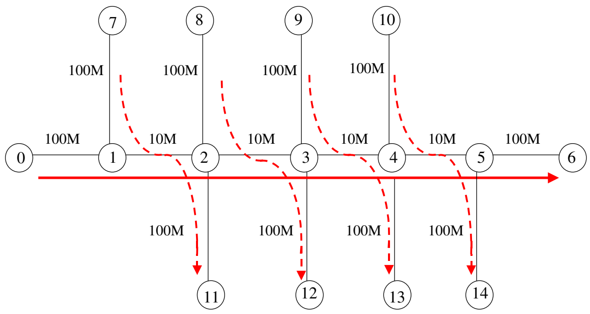


Figure 1. The network topology used in the simulation.

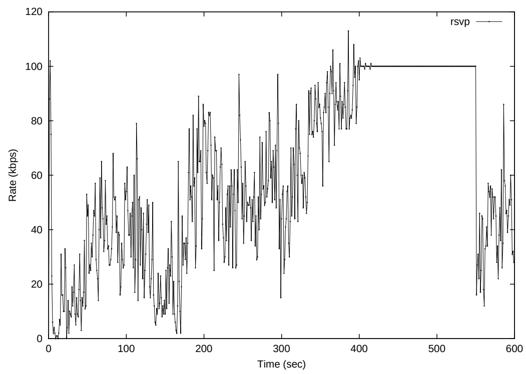


Figure 2. RSVP reservation completed after 150 s and 5 tries.

### 3.1. Partial Reservation: RSVP vs. YESSIR

In a first experiment, labeled “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.

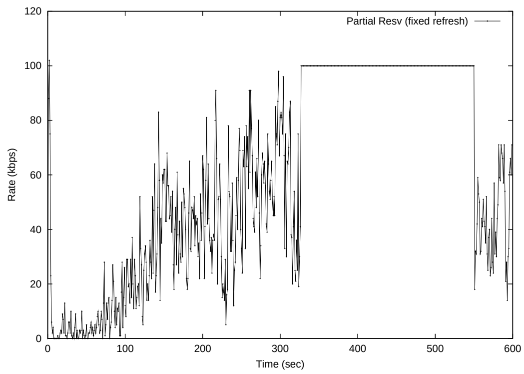


Figure 3. YESSIR (retry only at every refresh cycle): reservation completed after 77 s and 3 tries.

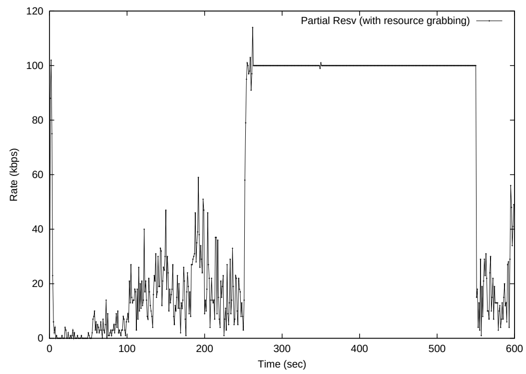


Figure 4. YESSIR (retry whenever resource becomes available): reservation completed after 12 s and 19 tries.

We then ran the same identical testing scenario with YESSIR 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 with YESSIR, where all the nodes, in particular, use the reservation recover mechanism that we have described in Section 2.4 with a very large threshold $\mathcal{T}$. The testing flow reservation takes only 12 seconds to complete, but retries 19 times.

We then collected reservation failure and success data from all the nodes. Figure 3.1 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 more aggressive retry mechanism performed the best. It completed the reservation in 12 seconds, and retried reservations 19 times in total on all 6 nodes during this

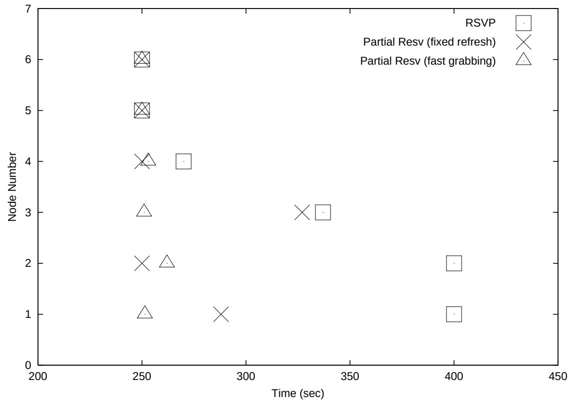


Figure 5. Reservation sequence for RSVP, YESSIR (retry with only refresh), and YESSIR (retry whenever resource becomes available). The ordinate shows the node number in the simulation network of Figure 1.

period.

### 3.2. Reservation Retry Performance

To evaluate the reservation retry performance, we increased the number of background reservation flows to 120, that is to request 240% more resource than what the network can provide. We experiment with both RSVP and YESSIR in this highly congested network. We also apply the threshold algorithm defined in Section 2.4 on all network nodes during YESSIR experiments.

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

| Description | # of retries on all nodes | Test flow between Node-0 and Node-6 |
|---|---:|:---:|
| YESSIR, $\mathcal{T} = \infty$ | 28,314 | no reservation made in 184 tries |
| YESSIR, $\mathcal{T} = 10$ | 8,534 | no reservation made in 46 tries |
| YESSIR,, $\mathcal{T} = 8$ | 7,137 | reservation completed after 232 s and 32 tries. |
| YESSIR, $\mathcal{T} = 3$ | 4,382 | reservation completed after 172 s and 14 tries |
| YESSIR, $\mathcal{T} = 1$ | 2,685 | reservation completed after 19 s and 1 retry |
| YESSIR, $\mathcal{T} = 0$ | 2,588 | reservation completed after 43 s and 3 tries |
| RSVP, fixed refresh | 3,409[2] | no reservation made in 10 tries |

Table 1 collects the total number of reservation retries from all network nodes, and the statistics on setting up an end-to-end reservation from node 1 to 6. We observed that

[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.

RSVP could not make the end-to-end reservation. This is probably due to its ineffectiveness during reservation retry. On the other hand, YESSIR with very aggressive retry process (high $\mathcal{T}$) failed to make end-to-end reservation as well. An explanation is that, with an aggressive retry algorithm, all reservation flows in the network try to take as much resource as possible. As a result, few flows get the full reservations.

With a proper threshold value, the user flow can successfully make the reservation in a highly congested network. In the simulation, the best scenario is the one with $\mathcal{T} = 1$. 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 seems to be supported by tests using different $\mathcal{T}$. We also conclude that in using YESSIR with reservation retry, there is a trade-off between the ability to obtain resource quickly, and the likelihood of causing reservation deadlocks.

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 less resources from the network, and thus perform poorly.

## 4. Protocol Implementation and Performance

In the previous section, we have shown that YESSIR has the properties to establish reservations faster, and recover from partial reservations quicker, comparing with RSVP. In this section, we formally introduce YESSIR protocol operation and evaluate its signaling processing overhead with an implementation on a common computer platform.

YESSIR is designed to be a very simple signaling protocol. It provides resource reservation to real-time flows that use RTP [11]. Figure 6 shows the reservation processing sequence at routers. 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 [12] in the IP header. 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.

We selected FreeBSD as the experimenting platform for YESSIR. Since FreeBSD does not have a clean mechanism to intercept IP option packets, including router alerts, through the socket interface, we have designed and developed a new socket family, `PF_IPOPTION`, on FreeBSD. More information on its implementation can be found in [13].

### 4.1. State Management

One of our objectives is to be able to support thousands of reservation flows efficiently, while keep the design simple. To meet this objective, we used hash tables to manage the reservation states in the implementation. The motivation behind using hashing comes from the observation that any RTP session in the network can be *uniquely* identified by

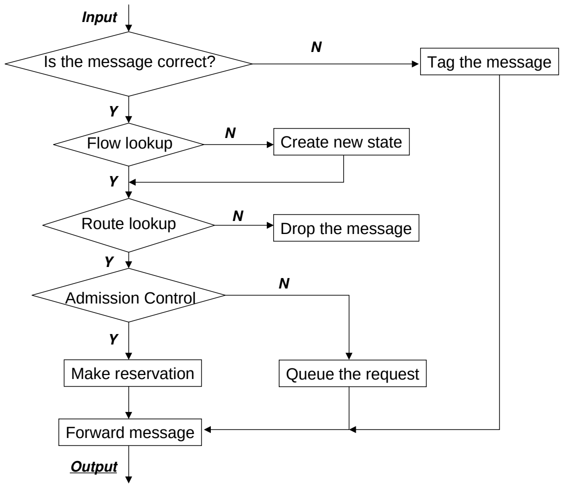


Figure 6. Reservation processing flowchart.

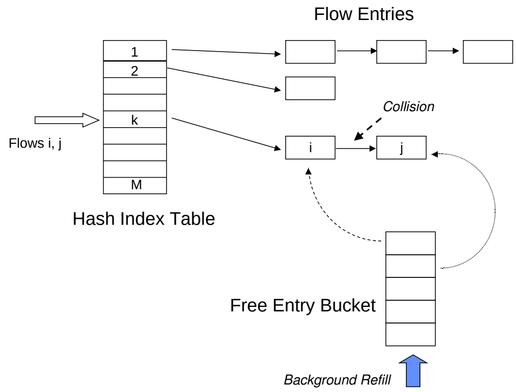


Figure 7. An example of reservation state management in our implementation

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 [14]. 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 simply: $h(k_i) \leftarrow k_i \bmod M$.

Figure 7 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.) The hash table and memory management take about 1,400 lines of C code to implement.

### 4.2. 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

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

Table 2
Hash table performance with collision resolution.

| Number of flows in the router | flow build time ($\mu$s) |
|---|---|
| 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 3
Processing time for YESSIR

| Description | Processing time (in $\mu$s) |
|---|---|
| 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 |

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.

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.

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 $\mu$s. 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 $\mu$s, i.e., approximately half the processing time is spent in the kernel.

## 5. Conclusion

We have investigated two important design issues in a reservation signaling protocol design: reservation retry and one-pass reservation.

Through simulation, we have compared the reservation performance achieved from RSVP and YESSIR. RSVP uses a more conservative hop-by-hop reservation retry mechanism, thus it may take a long time to achieve reservation convergence in resource-constraint networks. On the other hand, a better alternative, such as the one employed in YESSIR, is to retry reservations on multiple links in parallel. We also discovered that routers need to control the reservation retry process of the failed flows. One such mechanism is to limit the number of reservation retries on failed flows.

To gain a better understanding of reservation costs, we have implemented YESSIR on the FreeBSD platform. Since YESSIR is a lightweight one-pass signaling protocol, we are able to accomplish good reservation performance at very low processing cost on routers. With careful implementation and reasonable state management techniques, we can support up to 10,000 reservation flow setups per second on a commodity 700 MHz Pentium PC.

Thus, we believe that with proper protocol design and implementation, routers can support a large number of user flows, while providing admission control. Further, 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.

## 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. 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.
6. 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.
7. 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.
8. R. Guerin, S. Herzog, and S. Blake, "Aggregating RSVP-based QoS requests," Internet Draft, Internet Engineering Task Force, Nov. 1997. Work in progress.
9. G. Gaines and L. Salgarelli, "RSVP implementation survey," tech. rep., Institute for Information Technology of the National Research Council of Canada, July 1997.
10. 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 (NOSSDAV)*, (Cambridge, England), pp. 141–151, July 1998.
11. 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.
12. D. Katz, "IP router alert option," Request for Comments 2113, Internet Engineering Task Force, Feb. 1997.
13. P. Pan and H. Schulzrinne, "PF_IPOPTION: A kernel extension for IP option packet processing," Technical Memorandum 10009669-02TM, Bell Labs, Lucent Technologies, Murray Hill, NJ, June 2000.
14. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.