An Evaluation of RSVP Using Real Applications

Tony DeWitt, Jun Gao, and Qifa Ke
{adewitt, jungao, ke+}@cs.cmu.edu

December 14, 1998

1 Introduction

Since the day it was published, RSVP [1] has drawn a lot of attention from the networking research community, and lots of follow-up effort was put in as evidenced by a number of RFCs. [3],[4], [5], [6] As a signaling protocol, RSVP fits into the IETF IntServ model well, and it has incorporated some rather innovative ideas such as receiver initiated reservations, multiple reservation styles, the concept of soft state, etc. However, the IntServ model is still in the research stage, so few real applications are using RSVP to make reservations. Even less attention has been paid to the evaluation of RSVP.

The issue of Best-Effort versus Reservations is a constant debate between the IntServ advocates and opponents.[2] The basic question is: Are reservations and the IntServ model necessary at all, or should we just keep the Internet's current best-effort architecture? Advances in networking hardware are making bandwidth more and more abundant, so many believe that there will be enough bandwidth for everyone. It is not at all clear at this point because there are so many uncertain factors involved and there is not a known way to compare the network utilizations under these two schemes.

Like all signaling protocols, RSVP has a certain amount of overhead both at reservation setup time and during the life of the reservation. There are additional problems in certain cases. For example, unicast applications that want to use RSVP may have extra overhead in setting up a reservation because the sender must know the receiver's IP address so it can set up the appropriate PATH state. To our knowledge, no one has quantified how much performance is degraded for a given application. There are other issues with RSVP, like the token bucket parameters that RSVP requires from an application to describe different traffic properties. It is unclear how useful they are and how some applications should set these parameters to make a reservation be as useful as possible.

We believe that it is important to have a strong understanding of RSVP's behavior to determine whether an application should use a reservation or just use best-effort service. Therefore, in this project we evaluate RSVP on three different benchmarks to gain a better understanding of the tradeoffs between best-effort service and reservations. It is not our aim to determine whether reservations or best-effort traffic is better. Instead, we want to know when it makes sense to use reservations and when it is sufficient to rely on best-effort service.

In the next section we state the objectives of this project. In Section 3 we give an overview of RSVP and our interface to it. In Section 4 we present each of the applications that we are evaluating and discuss issues associated with each of them. A discussion of our experimental results are in Section 5. Finally, we conclude in Section 6.

2 Objectives

There are two main objectives for this project:

1. Modify three applications to use RSVP to make reservations instead of using best-effort service. These applications should have different traffic characteristics and should require varying qualities of service. The applications are discussed in detail in Section 4.

2. Evaluate the performance of RSVP for each application. The evaluation can be broken down into the following parts:

   (a) First, we evaluate the usability of RSVP. Usability of any system is subjective at best. This part of the evaluation is more of a description of our implementation and the problems that we have encountered. For example, we had to determine what the "correct" traffic parameters for an application are. Also, we had to decide what to do when an application has a reservation rejected?

   (b) Next we evaluate the performance of each application with and without RSVP support. For this part, we ignore the impact of the application using RSVP on other network traffic. We examine several questions about RSVP such as how much overhead is involved in using RSVP (e.g., setting up path state, making reservations, refreshing soft state) and when an application should use reservations and when it should use best-effort service.

   (c) Third, we evaluate how an application using RSVP interacts with other applications that may or may not be using RSVP. We look at how much protection a flow has from other flows when a reservation succeeds and whether this varies if the other flows are using RSVP. We also examine how best-effort flows are affected by flows that have made reservations.

3 An Overview of RSVP

ISI's implementation of RSVP is implemented as daemons that run both on the host machines and on the routers (see Figure 1). The daemons communicate with each other to set up PATH state and to make reservations. Each daemon communicates with the local traffic controller (i.e., packet classifier and scheduler) to install reservation state on that machine. Applications running on an endpoint node communicate directly with the RSVP daemon running on that machine. This is currently done via a user-level library called RAPI [7]. There is another library called SCRAPI [8], which is layered on top of RAPI, that provides a cleaner, simpler interface to applications. Both of these interfaces are briefly described below.

3.1 RAPI

RAPI contains three main procedures that sender and receiver applications use to set up reservations. These are:

1. **rapi_session**(). This is called by both senders and receivers to register with the RAPI library and create a session id. The session id is used by the RSVP daemon to differentiate between multiple applications or multiple sessions within a single application.

2. **rapi_sender**(). Each sender calls to this procedure. This specifies the traffic characteristics of the sender to the RSVP daemon. It is also used to set up PATH state between this sender and the receiver(s).

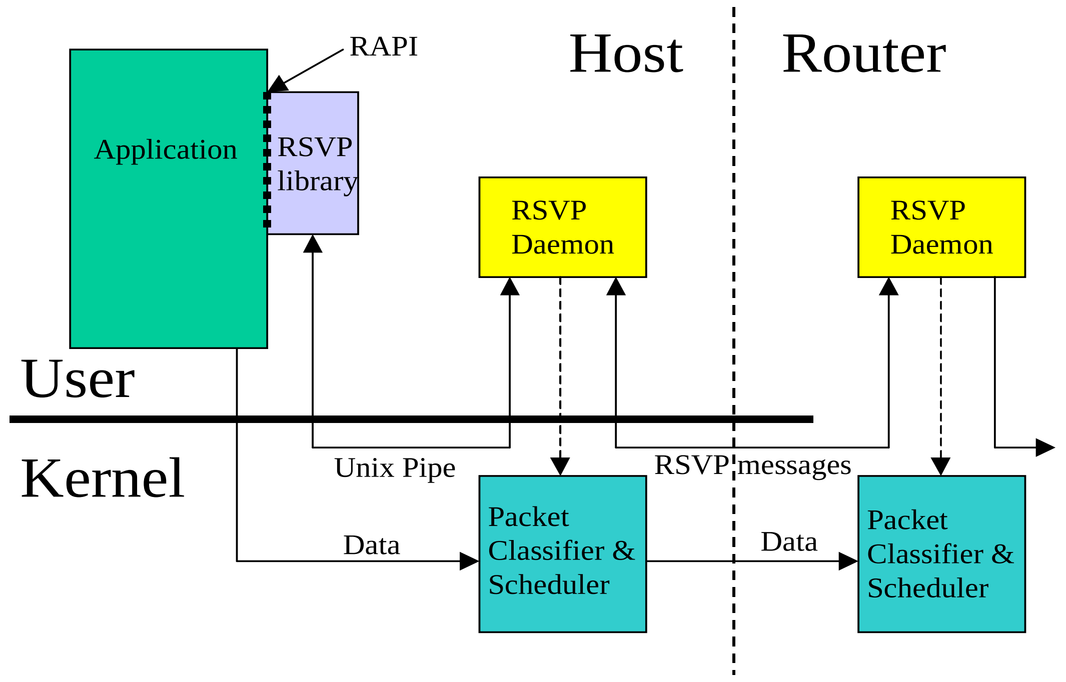

Figure 1: The RSVP implementation model.

3. **rapi_reserve**(). Each receiver makes this call to specify which sender(s) it wants to receive and what quality of service is desired.

There are various other procedures including ones that remove reservations, poll for RSVP events, and format RAPI data structures.

3.2 SCRAPI

SCRAPI [8] was designed as an interface to applications that simplifies the process of adding RSVP support to them. While RAPI only has three main procedures, they take complex parameters (e.g., flow specs, filter specs, ad specs). SCRAPI abstracts these complexities away, requiring only simple parameters (e.g., bandwidth, reservation style) from the programmer. On the other hand, it is less flexible than RAPI (e.g., it does not allow applications to register upcalls).

SCRAPI is layered on top of RAPI. The simple parameters passed to it are translated (making a number of simplifying assumptions) into the more complex parameters required by RAPI. This translation is detailed in Section 3.3. Like RAPI, there are three main procedures:

1. **scrapi_sender**(). This is called by a sender to set up PATH state between this sender and the receiver(s). The sender needs to specify only a few parameters like the amount of bandwidth required, and an optional time-to-live parameter.

2. **scrapi_receiver**(). This is called by a receiver to make a reservation. Only a few simple parameters are specified, such as the IP address of the sender (left as NULL for all senders), the reservation style, the type of service, and whether this call should create, update or destroy a reservation. This call assumes that the receiver's FlowSpec is the same as the sender's TSpec.

3. **scrapi_close**(). Both receivers and senders call this when they decide to finish an RSVP session and to release the reservation.

3.3 SCRAPI Translation

In this section we describe how the SCRAPI parameters are translated into HFSC service curve parameters, which are used by the packet scheduler.

scrapi_sender() has one parameter that describes the source traffic requirements: *bandwidth*. The *bandwidth* parameter is translated by SCRAPI into the token bucket parameters used by RAPI: *token bucket size*, *token bucket rate*, *peak rate*, *minimum policed unit*, and *maximum packet size*. The translation is performed using the following formulas:

```
token bucket rate    = bandwidth
token bucket size    = 2 * bandwidth
peak rate            = 2 * bandwidth
minimum policed unit = MIN_PACKET_SIZE (64)
maximum packet size  = MTU
```

scrapi_receiver() has two parameters we are interested in. One is the *service type*, which specifies the service is either *guaranteed* or *controlled load*. The other is the *reservation style*. The current version of SCRAPI defines two styles: *scrapi_style_shared* and *scrapi_style_distinct*. The former is translated by RAPI as *share_explicited* style while the later is *fixed filter*.

The QoS kernel that we are using implements an HFSC scheduler. It contains three parameters: *m1 = peak rate*, *m2 = long term rate*, and *d = delay*. The RSVP daemon must convert the token bucket parameters to the service curve parameters. It uses the following formulas:

```
m1 = min(Rcv.RSpec.R, Rcv.TSpec.p, Snd.TSpec.p)

m2 = min(Rcv.RSpec.R, Rcv.TSpec.r, Snd.TSpec.r)

d = min(sender bucket size, receiver bucket size) / m1
```

where *Rcv.RSpec.R* is the receiver required service rate. p is the peak rate and r is the token bucket rate, or long-term rate. It is required that $R \geq r$. If the *RSpec* rate is bigger than *TSpec* rate, then the queueing delay will be reduced. The slack term is ommitted by this version of the RSVP daemon.

When the application calls SCRAPI or RAPI to make a reservation, the traffic controller will get the reservation parameters from the RSVP daemon and create a class in the resource tree of each appropriate output interface using the derived service curve parameters.

4 Applications

In this project, we study the use of RSVP with three applications: the Apache web server, Vic (a video conferencing tool), and a distributed FFT computation. Each application uses the network in a different way and has very different traffic patterns. These applications are detailed in the following subsections.

4.1 Apache

The Apache web server is public domain software,[9] freely available with the complete source code ($\sim$ 25,000 lines of C). Despite this fact, it is the most widely used web servers on the Internet today. Because of its importance to the Internet community and the fact that it (and other servers like it) are the source of much of the Internet's traffic, we feel that it is a worthwhile application to study.

Since connections to a web server are short-lived and transfer a small amount data, it is not clear when it makes sense to make reservations for these flows, or if it makes sense to do so at all. This may change in the future because recent versions of the HTTP protocol allow multiple objects to be fetched per connection, instead of one object per connection. Also, connections between a web client and a proxy cache could remain open for an extended period of time. Reservations on these connections could improve performance for objects that are cached at the proxy.

Another interesting characteristic of web traffic is that hard performance guarantees are not required. A user may get annoyed if bandwidth is scarce, but most web data is not highly time sensitive (except maybe stock quotes). This should be factored into the decision on whether or not to use reservations.

We could possibly use a commercial web client, such as Netscape or IE to retrieve data from the Apache server. Since our focus is on the data throughput between the server and the client and we do not need to display the data retrieved, we decided to write our own version of web client. This way we have full control over the client, so we are able to insert instrumentation code that records the desired performance measurements.

Apache runs as an HTTP daemon when started, and it blocks until it receives a connection request from a client. It will then spawn a child process, and the main process will go back to the wait state. The newly spawned child process will first accept the request, set up a TCP connection to the client, then process the request. After the data transmission is complete, it closes the TCP connection.

To provide RSVP support, we modified the child process in Apache such that immediately after it sets up a TCP connection with the client, it will call **scrapi_sender**() and then the RSVP daemon will send out a PATH message towards the client. Apache can not claim itself to be the RSVP sender any time earlier than this, because it does not know who the receiver is until it gets a request.

We also modified the client to call **scrapi_receiver()** immediately after it connects to the web server. This causes the RSVP daemon on the client machine to send out a RESV message. The reservation will be set up when the server receives RESV message and the client receives CONFIRM message. The reservation is torn down by a **scrapi_close**() call before the TCP connection is closed at the end of the data transfer.

This means that every client request will result in one reservation being set up. This occurs even if a single client makes multiple requests to the same server. A simple optimization would be for the client to make a single reservation for all of the requests it makes to a single server.

While this is possible, the authors decided not to implement this way for several reasons:

1. The reservation state should not persist, because the client may not connect to the server again after one retrieval. Therefore if we keep the state around, certain amount of resource is wasted.

2. When a new connection is set up between the same client and server, the server will choose a new TCP port for this TCP connection. Therefore unless we hack the traffic controller code in the RSVP implementation to install a filter for the classifier and scheduler that treats the

port fields as do-not-care, the new connection will not be able to utilize the reservation that was previously made.

3. We understand that the future design of the HTTP protocol will allow multiple retrievals per request. Then our implementation will take advantage of this nicely without the above two complications. The multiple retrievals of one request will share the reservation which will be made only once at the beginning of this connection.

4.2 Vic

Vic is a video conferencing application developed at Berkeley. Video is captured by a camera at each sender and transmitted in real-time to each receiver. This requires hard bandwidth and timing guarantees and, depending on the quality of the video, can require a fair amount of bandwidth. Connections are long-lived, so the RSVP signaling overhead is negligible once the reservation is set up. Vic uses UDP to transmit real-time video frames. Because of its traffic requirements and the characteristics of its data flows, vic is an ideal application for using RSVP.

The source code is freely available, so it is fairly simple to add RSVP support to vic. While we were implementing this support, we found a version that uses the SCRAPI interface that had already been written at ISI. We used this version for all of our experiments since it was already done and because it probably has a reasonable user base. However, their implementation has some limitations. For example, they assume that each machine has a single network interface. This caused us some confusion since the testbed machines have at least two network interfaces. It took us two days to track down this problem and modify their implementation to work on the testbed.

When a sender begins to transmit its video stream, vic calls **scrapi_sender**() to set up PATH state between this sender and the receiver(s) (which can be a multicast group). A set of token bucket parameters are included in the message. The numbers depend on the sender's video format, the resolution of the video, and the frame rate. Once the PATH state has been installed, receivers can click on a RESERVE button on their UIs. This calls **scrapi_receiver**(), reserving the maximum amount of bandwidth that the sender could possibly send. Each receiver can do this for any number of senders. Whenever a sender changes the maximum bandwidth it will send, each reservation is updated to this new level. If a reservation fails, an error message is displayed to the user.

4.3 FFT

FFT is a distributed computation application that performs a sequence of NxN 2D FFTs. We compiled it to run on 3 nodes, with $N = 1024$, and to run for 32 iterations. FFT has an interesting communication pattern in that each node executes code for a period of time, then it distributes the results to the other nodes. For this reason, traffic is very bursty. The FFT communication uses TCP. There are also synchronization issues in that each node waits at a barrier at the end of each round until all nodes have communicated their results from the current round. So, none of the nodes can continue until the slowest one has completed its computation and communication stages.

The version of FFT that we have is written completely in FORTRAN and uses MPI to perform all inter-node communication and it only runs on Digital Unix. Because of this we are not able to use RAPI or SCRAPI (which are C interfaces) to add RSVP support. Furthermore, the RSVP daemons that we have run only on FreeBSD, and as a result we can not run RSVP daemon on the FFT end nodes.

We used one quick solution to work around these problems. We run RSVP daemons in interactive mode on all the routers between the FFT nodes. We then have the capability of installing reservations manually on all the links between routers that may be affected by cross traffic. This way the endhosts do not need to run the RSVP daemon but there will also be no reservations on the edge links. We make sure that during our experiments, there is no interfering traffic on the edge links between each FFT node and its attached router. The reservation installation overhead from RSVP will not be observed due to this limitaion.

5 Evaluation

5.1 Methodology

All of our experiments are run on the Darwin testbed. The part that we used for our experiments is shown in Figure 2. The three routers are yosemite, timberline and maui. Yosemite and maui are Pentium II 300MHz PCs running FreeBSD 2.2.6 and Timberline is a Pentium II 266MHz PC running FreeBSD 2.2.6. Among the end hosts, m3, m4 and m6 are Digital Alapha 21064A 300MHz workstations running Digital Unix 4.0. The end systems swiss and glen are Pentium II 400 MHz PCs running FreeBSD 2.2.6. All the links are full-duplex point-to-point Ethernet links configured as 100Mbps except the two links between yosemite and maui are configured as 10 Mbps.

We used the latest version of RSVP (rel4.2a4) implemented at USC ISI. The traffic controller code that interacts between the RSVP daemon and the packet classifier and scheduler was written by Yang-hua Chu here at CMU.

The cross traffic we used in the experiments is a CBR UDP stream which is generated by a modified version of ttcp (courtesy of Prashant Chandra). The rate of the CBR stream can be specified when the program is started.

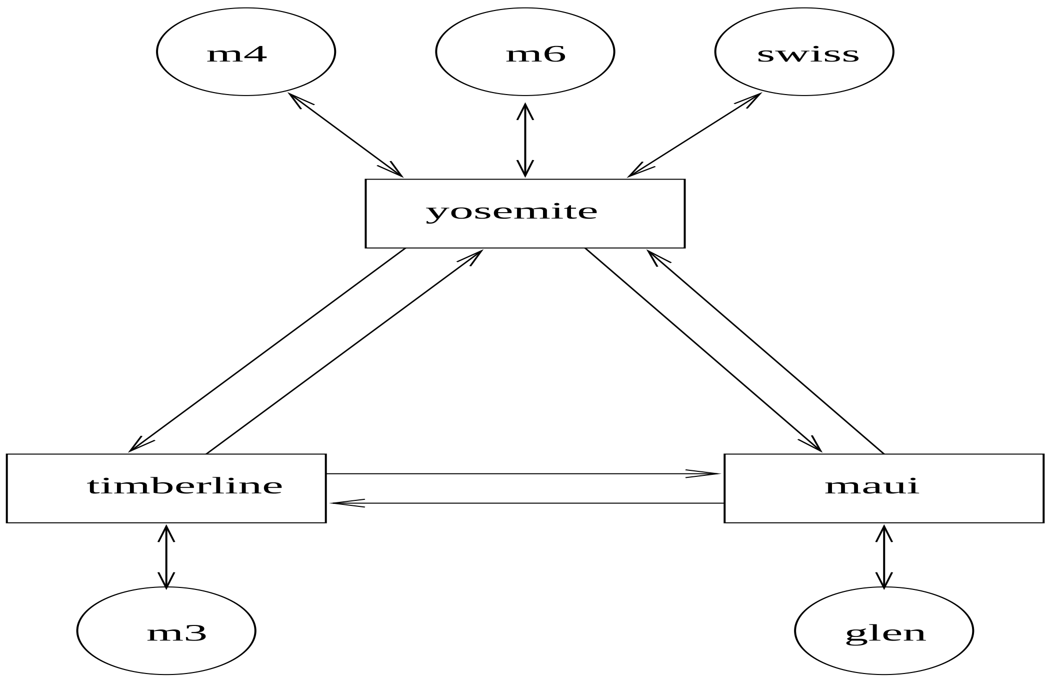

Figure 2: Testbed topology

5.2 Overhead of RSVP

The overhead, in terms of performance, not the amount of protocol state, introduced by RSVP can be classified into two categories:

1. The time to set up RSVP state. When an RSVP session is established between a sender and a (set of) receiver(s), the sender must first send a PATH message toward all possible receivers. The receiver(s) can not make a reservation until the intermediate routers receive these PATH messages and install PATH state. Once this has been done, receivers can send a RESV message back toward the sender to install a reservation. The time overhead of this process is mainly dependent on the round trip time between the sender and the receivers. This can be significant in a WAN, but is not as noticeable in a LAN environment.

2. The refreshing of that state. PATH and RESV state in the routers is soft state. If the sender and receivers do not periodically refresh that state with additional PATH and RESV messages, the state will expire and be removed. These messages are small, so their bandwidth requirements are negligible.

5.3 Apache

We did a series of LAN experiments for Apache using the Darwin testbed. We ran the Apache server on Yosemite and ran the web client on Timberline. Ideally, we would like to run the server and client as far apart from each other as possible. However, due to some technical difficulties, the TCP performance between the FreeBSD end hosts are extremely low ($<$ 1Mbps measured with ttcp using proper parameters). That made them effectively unusable. We use m4-fast and m3-fast as our cross traffic sender and receiver.

For each experiment below, the client makes 200 consecutive HTTP GET requests to the server and tries to retrieve a 1MB file each time. The reason we chose a 1MB file to transfer is to make sure each HTTP file transmission can use a significant portion of the 100Mbps link. [1]

Figure 3 shows Apache's performance without reservation. As we can see, the average bandwidth of Apache transfer is about 41 Mbps when there is no background traffic. When the sum of the amount of cross traffic and the Apache traffic is smaller than the available link capacity ($\sim$ 75 - 80 Mbps), no packets are dropped and Apache performance stays the same. When the cross traffic is increased to about 35Mbps, congestion occurs on the bottleneck link, packets get dropped, and the Apache server backs off immediately (TCP). The Apache throughput decreases to about 20 Mbps when the cross traffic increases from 40 Mbps to about 60 Mbps. Since the cross traffic is not large enough, i.e. there are more than 20 Mbps available, the Apache throughput stays at above 20 Mbps. One thing seems to be a little puzzling is that when the cross traffic is around 40 Mbps, we expect that Apache to get a throughput of about 30 Mbps. The measured numbers (23 Mbps) seem to be a little low. When the cross traffic is further increased, the Apache performance degrades to almost zero. Note that the throughput of the cross traffic keeps increasing linearly until it hits the available link capacity.

Figure 4 compares the behavior of Apache traffic with and without reservation. Apache reserves 45Mbps average bandwidth for each request, before any data is sent. As seen in the figure, the Apache throughput is not affected when the cross traffic is increased. However when the CBR rate is higher than 35Mbps, the cross traffic throughput is really low ($\sim$ 1Mbps, not shown), because under a congestion condition, the scheduler drops all the cross traffic's packets.

[1]When the file is too small, the TCP connection finishes before it can get to a stable state. Also when the file is small, the TCP sending buffer is small, and therefore the bandwidth utilization is small.

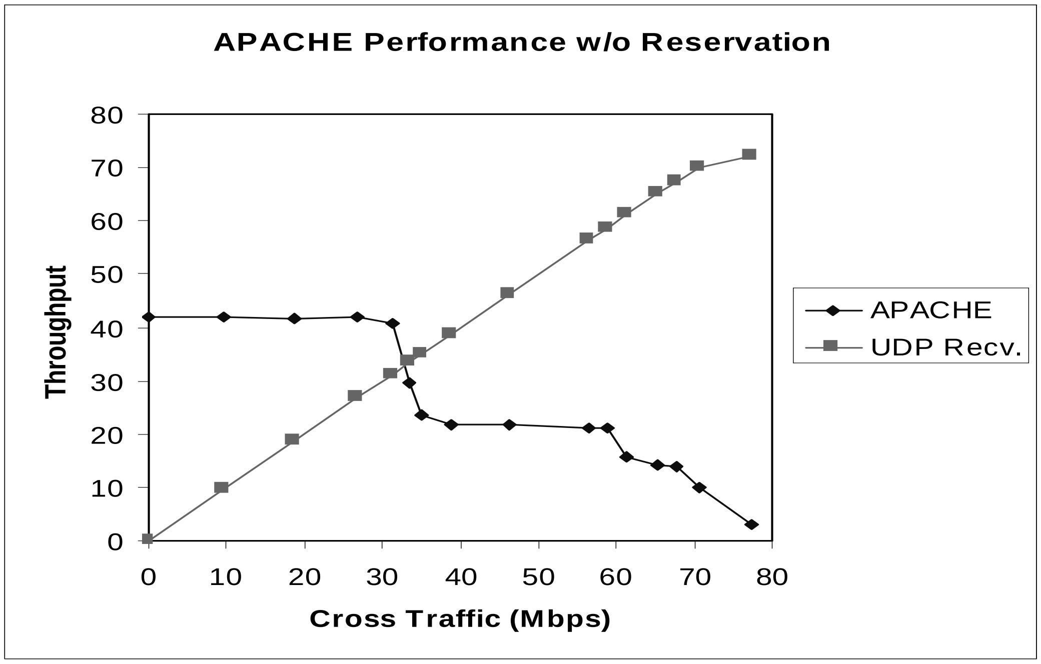

Figure 3: APACHE performance without reservation.

The degradation of the average Apache throughput because of the reservation overhead is marginal (see the first two points in this figure, where cross traffic is zero). This is not totally surprising when considering the short round trip time between the server and client ($\sim$ 400 us). In this set up we measured the average time to set up one reservation to be about 470 us. Compared to the time needed to transfer a 1MB file ($\sim$ 200ms with no cross traffic), the reservation installation overhead is about 0.24%. The overhead caused by the RSVP refresh messages is even smaller because it only happens once every 30 seconds.

We were hoping to conduct some WAN experiments to further investigate the RSVP overhead. We have successfully set up a CAIRN connection. Our experiments were hindered for a rather disappointing reason of not having the root password on the remote machine.

5.4 Vic

Our main experiment for vic was to have a single sender (swissvale) and a single receiver (glenwood) which are separated by two routers on the testbed. The sender attempts to transmit a video stream to the receiver at the frame rate of about 12 fps and at the maximum bandwidth (3.1 Mbps). UDP cross traffic is added to the link between the two routers. We examine the transmission of the video stream with varying amounts of cross traffic, with and without reservations.

Figure 5 shows the throughput of the video stream versus the amount of cross traffic. As can be seen, when a reservation is made for the full amount of bandwidth needed by the stream, the throughput is unaffected by the cross traffic. But when no reservation is made, the video stream throughput drops once the link becomes saturated.

Since there is a linear decrease in the amount of throughput observed as the cross traffic is increased, one might expect a linear decrease in frame rate. Instead, Figure 6 shows that the frame rate drops by nearly half as soon as the link becomes saturated, then drops to only 2 fps as the cross traffic increases. This is because whenever part of a frame is lost, vic discards the entire

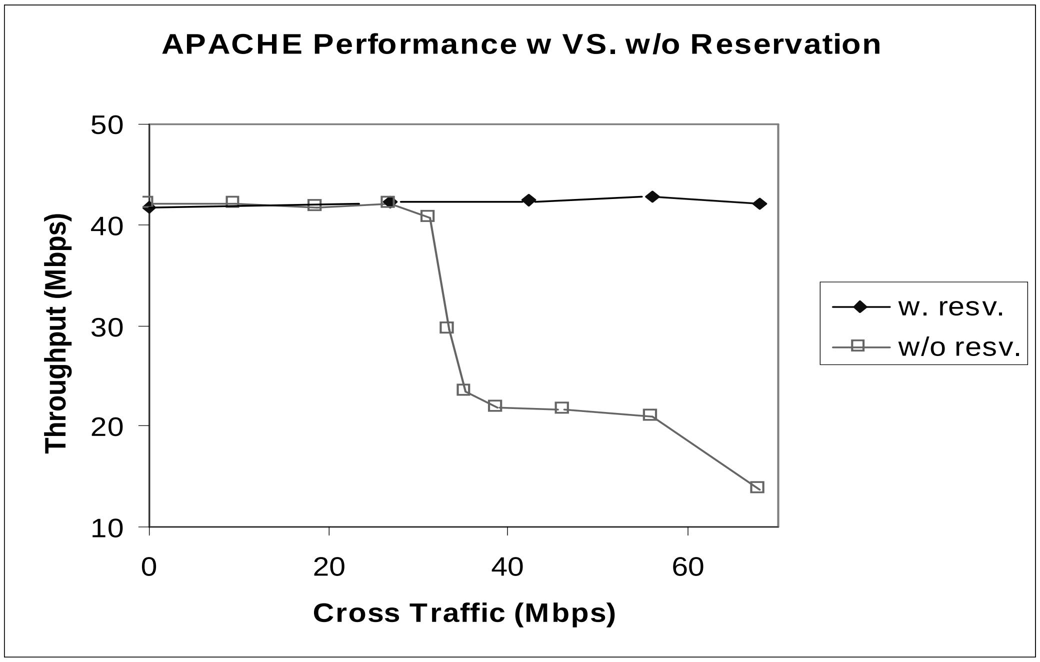

Figure 4: APACHE performance with vs. without reservation.

frame. Figure 7 shows that the frame error rate increases dramatically when bandwidth becomes scarce.

At a cross traffic of 7Mbps, the error rate is less than 10%, and the frame rate drops by nearly one half. This means that the errors are spread across many frames. So we see that vic is highly sensitive to errors, and the lack of reservations when traffic is high can be very costly.

It is also interesting to look at how much cross traffic throughput is achieved compared to the amount that is sent. This is shown in Figure 8. When vic has made no reservation, the cross traffic stream continues to get more bandwidth after the link has been saturated. But, if vic has made a reservation, the stream loses bandwidth when it attempts to send more after the saturation point. This is because only the cross traffic stream suffers packet loss from congestion instead of both streams losing packets.

5.5 FFT

FFT is run on three machines, m3-fast, m4-fast, and m6-fast. Since m4-fast and m6-fast are attached to the same router, we run cross traffic on one of the links between that router (yosemite) and m3-fast. This is the 100Mbps link between yosemite and timberline. Cross traffic is transmitted between swiss and glen (the routing table of yosemite is updated to make this flow pass through timberline). The cross traffic interferes with the transmission of data from m4-fast and m6-fast to m3-fast. This represents one third of the data communicated between nodes during the FFT computation. The experimental setup is depicted in Figure 9.

Experiments are run varying both the amount of cross traffic and the size of the reservation. When run without any cross traffic, FFT has approximately a 1:1 ratio of computation to communication time. Therefore, we set the peak rate for the reservation to be 2X the average rate. Also, the token bucket size is set to be the peak rate (* 1 second).

We ran experiments for three different peak rates: 0 Mbps (no reservation), 1Mbps, and 10Mbps.

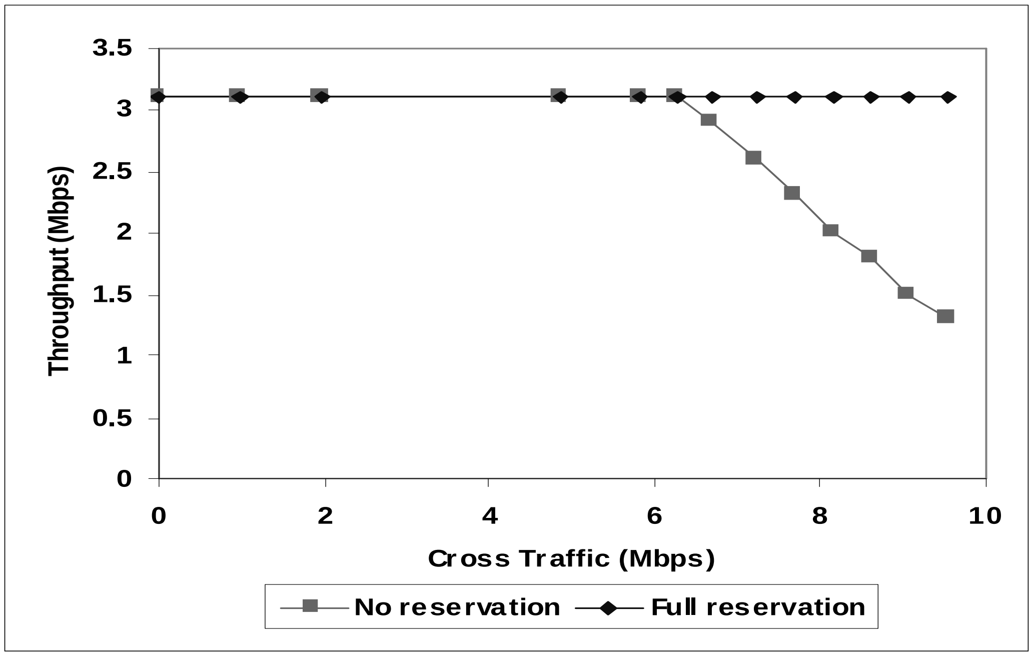

Figure 5: Vic throughput vs. cross traffic.

The results are shown in Figure 10. As can be seen, each of the cases performs the same when the cross traffic is smaller than 70 Mbps. This indicates that the bandwidth required by FFT during burst period is less than 10 Mbps. As the traffic is increased, the cases with less reservation are affected significantly more than those with more reservation. Also, the less reservation cases are affected earlier than those with more reservation. This means that for these types of applications, there is a clear tradeoff decision that needs to be made between how much reservation should be made and the desired execution time. If a large reservation is made, the application will have good performance. But, the application will be occupying lots of network resources. A smaller reservation could be made, resulting in (possibly) lower performance, but fewer resources would be used.

6 Conclusions

RSVP is the chosen signaling protocol for the IETF IntServ model. However its usability for real applications and its performance are not widely documented. In this project, we took three real applications, namely Vic, Apache and Distributed FFT and modified them so that they can invoke the RSVP daemon to make reservations according to their own traffic characteristics before the real data is sent. We used SCRAPI, an RSVP API package to bridge the applications and the RSVP daemon. We then carried out a series of evaluation experiments on a local area test network. The results are pretty much as expected: all applications benefit from their reservations (i.e., when facing congestion, their performance does not degrade, and other non-reserved traffic suffers). The overhead involved in making reservations on a LAN is negligible.

Reservations are particularly useful for Vic-type applications, because they have moderate bandwidth requirements but need hard guarantees. The network can accept a fairly large number of such connections. However, for applications like Apache, it is hard to make use of RSVP. The user

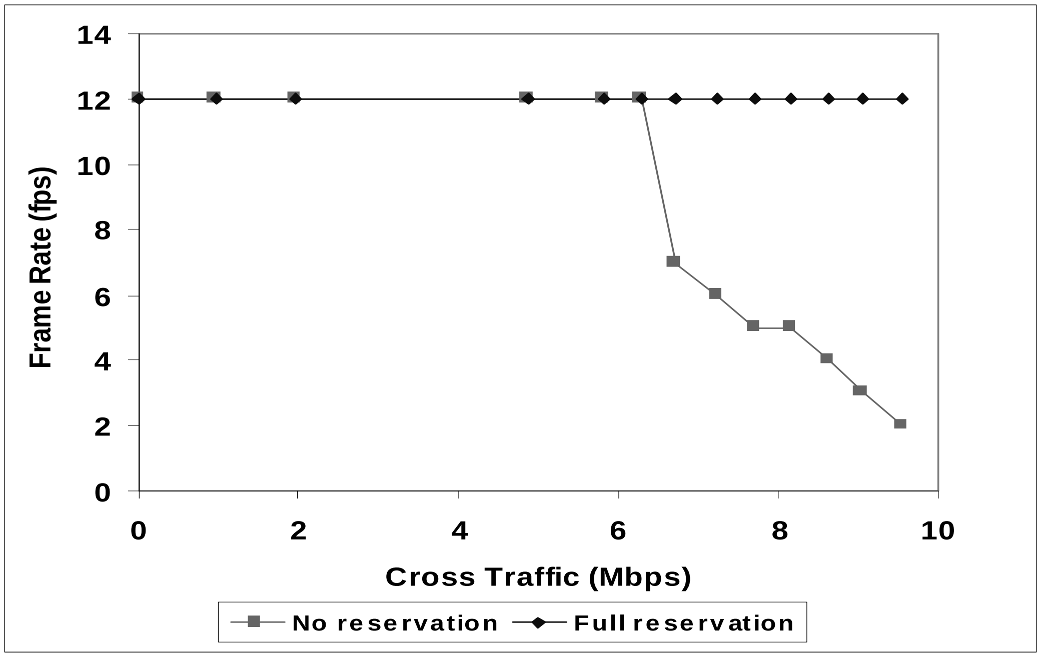

Figure 6: Vic frame rate vs. cross traffic.

of such an application has some tolerance of degraded performance, but would prefer higher performance. The difficulty lies in that the application can not make a good estimation of the bandwidth it needs. If it tries to reserve too much, it is very possible that it will be rejected by the admission control. If it reserves too little, the reservation may be useless. For the FFT computation, when each node has to get data from other nodes no later than a deadline, hard guarantees are required. However, this type of traffic is highly bursty, and the application does not use the network for a significant portion of its running time. If the application tries to reserve its peak rate, it could be rejected or, if it is accepted, lots of bandwidth would be wasted because of the nature of the burstiness. Therefore the application needs to use a compromised amount to make a reservation.

The token bucket parameters that RSVP uses to describe traffic characteristics do not seem to be all that useful, especially to the HFSC scheduler that we are using. The one parameter that matters most is the average rate.

6.1 Who did what

6.1.1 Experiments

1. Apache: Gao, Qifa
2. Vic: Gao, Qifa, Tony
3. FFT: Tony

6.1.2 Writing

We all worked on the each of the reports and presentations.

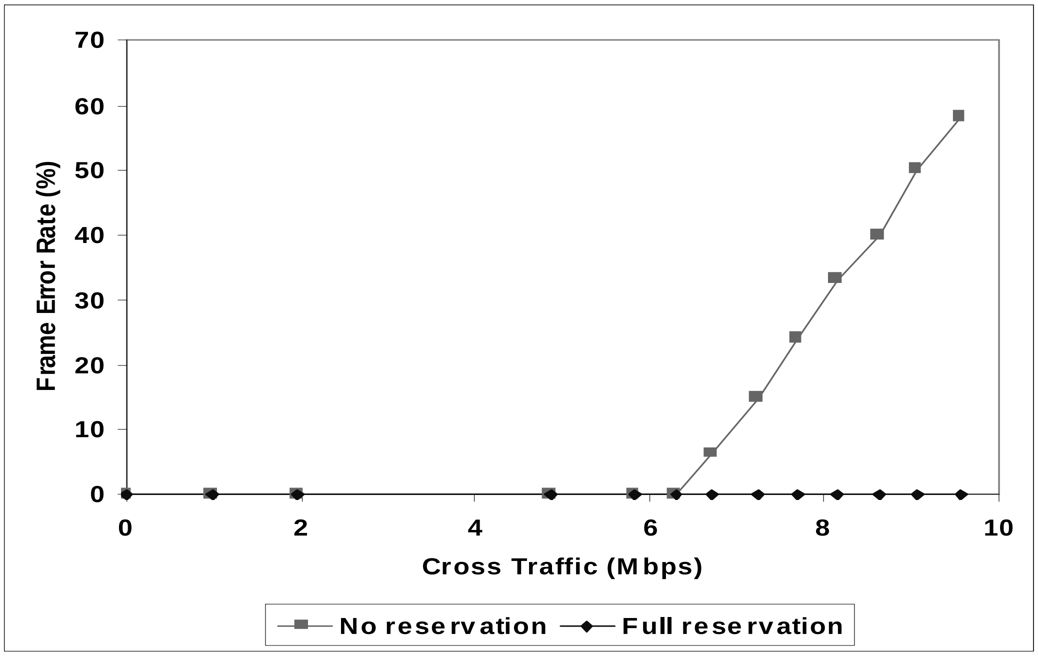

Figure 7: Vic frame error rate vs. cross traffic.

References

[1] L. Zhang and S. Deering and D. Estrin and S. Shenker and D. Zappala, "RSVP: A New Resource Reservation Protocol", IEEE Communications Magazine, 31(9):8-18, Sept 1993.

[2] L. Breslau and S. Shenker, "Best-Effort versus Reservations: A Simple Comparative Analysis", Sigcomm 98.

[3] R. Braden, et al, "Resource ReServation Protocol (RSVP) – Version 1 Functional Specification" , RFC 2205, September 1997

[4] J. Wroclawski, "The Use of RSVP with IETF Integrated Services", RFC 2210, September 1997

[5] J. Wroclawski, "Specifications of the Controlled-Load Network Element Service", RFC 2211, September 1997

[6] S. Shenker, et al, "Specifications of Guaranteed Quality of Service", RFC 2212, September 1997

[7] R. Braden, et al, "RAPI – An RSVP Application Programming Interface, Version 5", Internet Draft, August 1998

[8] B. Lindell, "SCRAPI – A Simple "Bare Bones" API for RSVP", Internet Draft, November 1998

[9] http://www.apache.org

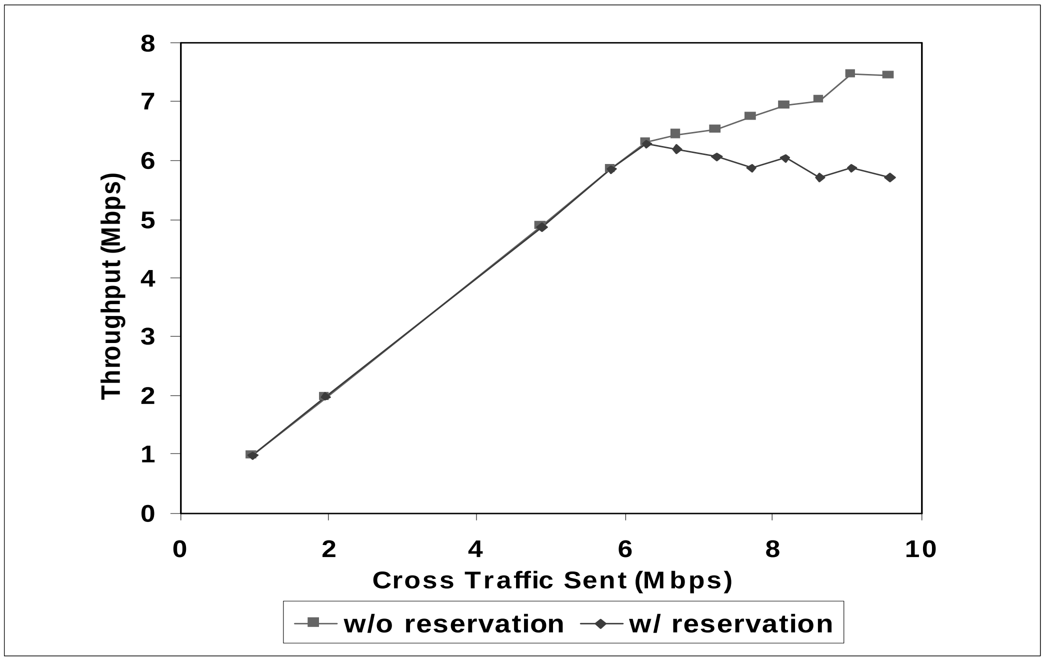

Figure 8: Vic cross traffic throughput vs. cross traffic sent.

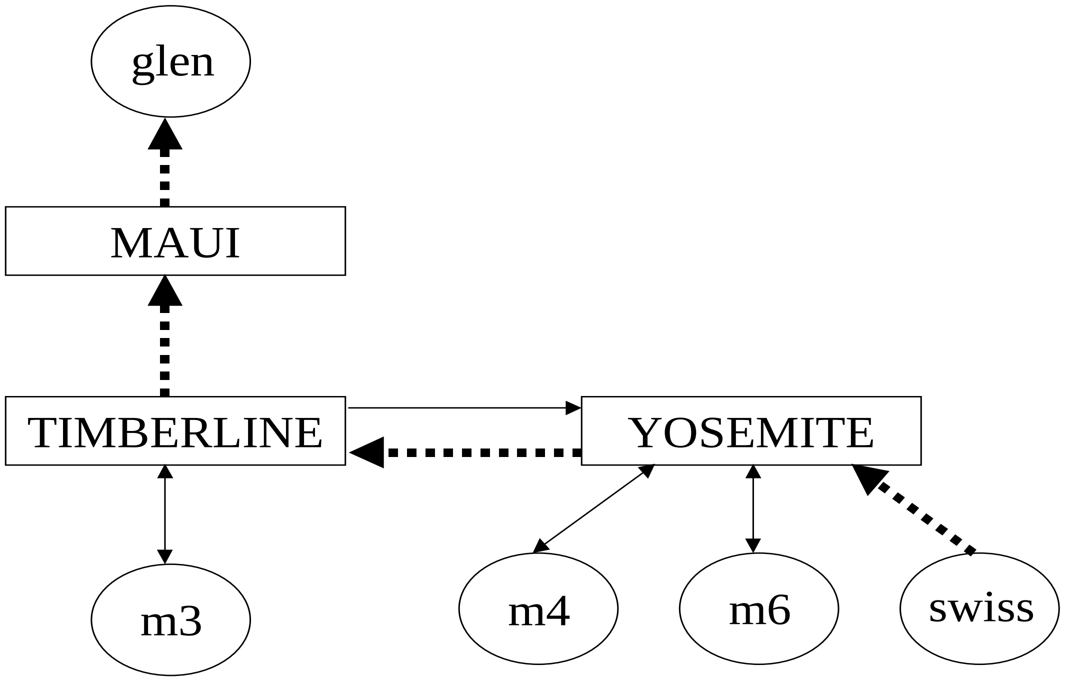

Figure 9: FFT experimental setup. The bold, dashed line indicates the path of the cross traffic.

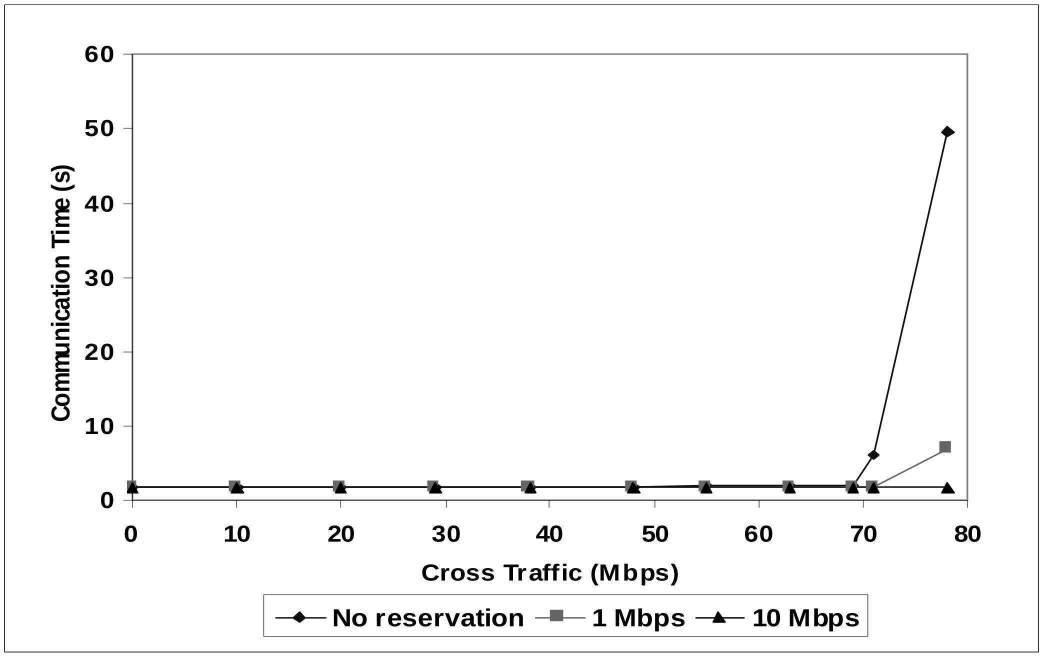

Figure 10: FFT communication time vs. Cross traffic for various reservation levels.