Implementation and Evaluation of the KOM RSVP Engine

Martin Karsten, Jens Schmitt
Darmstadt University of Technology, Darmstadt, Germany
http://www.kom.e-technik.tu-darmstadt.de/

Ralf Steinmetz
Darmstadt University of Technology, Darmstadt, Germany
GMD IPSI, Darmstadt, Germany
http://wwwp.ipsi.gmd.de

Email: {Martin.Karsten,Jens.Schmitt,Ralf.Steinmetz}@KOM.tu-darmstadt.de

***Abstract* - In this paper, we describe implementation aspects and performance results of an innovative and publicly available RSVP implementation. Much debate exists about the applicability of RSVP as a signalling protocol in the Internet, particularly for a large number of unicast flows. While there has been a significant amount of work published on the theoretical concepts of RSVP signalling and conjectures about its presumed shortcomings, rather little attention has been paid to the implementation details of the core protocol engine. With our work, in spite of being still far from a final judgement, we try to shed light on this issue by presenting certain design details of a new implementation and a study about its performance. One particular result is given by the observation that a relatively cheap router based on PC hardware can sustain the signalling for more than 50,000 unicast flows.** [*]

I. Introduction

Much debate exists about the applicability and performance capabilities of RSVP [1] as a QoS signalling protocol for the Internet. However, the discussion of this issue usually lacks solid performance figures from real experiments using a real implementation. Furthermore, the issues of signalling complexity (control plane) in general and packet forwarding complexity (data plane) of per-flow and per-hop reserved flows are often confused. The goals of this paper are twofold. First, we aim to provide additional insight about RSVP's performance capabilities by presenting central design characteristics of our protocol implementation. Second, we present detailed experimental data describing load measurements we have carried out using this implementation. These data cannot be used to draw a single major conclusion, but represent detailed hard facts for others to extrapolate them into their respective implementation context. We do however observe that, despite its undebatable complexity, RSVP can perform better than often assumed. Additionally, we conclude that there is potential for further optimization, both through protocol extensions as well as internal optimization of the protocol engine.

In the past, various proposals have been published, which describe useful extensions to the basic version of RSVP (see Section II.B for details). The goals of these extensions are mainly to complete RSVP's specification in the areas of security and reliability and furthermore, to improve the state refreshing mechanism, which is already identified as currently limiting overall performance. On the other hand, little attention has been paid to the implementation of the core protocol engine itself. As a result, RSVP is often assessed as having a poor performance, however, those judgments are rarely based on solid data. Therefore, the internal design structure and algorithms, as well as the overall protocol performance, have been subject to careful investigation in this work. In this evaluation, we focus on large numbers of unicast flows. One reason is given by the prohibitively extensive infrastructure necessary to carry out large-scale experiments with multicast communication. The second and more important reason is that the suitability of RSVP to handle large multicast groups, for which it was intentionally designed, is commonly undisputed. Rather, the handling of a large number of unicast flows is considered as the dominant scalability problem of RSVP.

The paper is structured as follows. In Section II, we review previous work related to RSVP performance and its evaluation. Section III presents our RSVP implementation, particularly certain central design concepts. In Section IV, we describe the general setup for the performance experiments, while the results are reported in Section V. In Section VI, we present profiling information to further back up certain findings from the performance experiments. The paper is wrapped up by presenting a summary, conclusion and outlook to future work in Section VII.

II. Related Work

A. RSVP Performance

Little work has been reported to assess the performance of commercial RSVP implementations. A notable exception is given by [2], in which a technical framework for carrying out such tests is presented. From the performance figures for a "commercial midrange router" given in [2], it can be deduced that RSVP flow setup scales significantly worse than linear. These results indicate that the particular version of the RSVP implementation under consideration may have been in a rather early development stage and cannot serve as a basis for judgements about its signalling performance.

Another investigation of RSVP's performance is reported in [3]. However, this work mainly considers an existing implementation, the *ISI rsvpd* [4], which we do not regard as the optimal choice for performance measurements. This is further

[*]. This work is partially funded by the European Commission under the 5th Framework Programme IST, Project M3I (11429).

documented in the rest of this paper. Some performance numbers are listed in [3] for another commercial RSVP implementation. However, this implementation does not sustain more than 600 sessions and thus, can be assessed similarly to the implementation studied in [2].

Other published work describes the implementation of an RSVP-capable switch-router in [5], but the reported performance figures are targeted towards the fundamental capability of the system to deliver QoS objectives in the first place, rather than performance of signalling at a large scale.

In [6], interesting performance figures are reported for RSVP message processing on a commercial router platform. However, these performance figures are somewhat without context, because it is not mentioned under which load conditions they were measured. Additionally, because these numbers are not the central focus of the work in [6], not many details about the experiments are given. Consequently, these numbers can serve as a basic indication about RSVP's processing overhead, but they cannot be considered as the final judgement in the discussion about RSVP.

In summary, although earlier work and published results have already indicated some of the conclusions shown in this paper, we present the first thorough study of RSVP's performance. This study is based on a publicly available implementation and thus, verifiable by others.

B. *RSVP Extensions*

A number of protocol improvements have been suggested to increase the performance characteristics of RSVP operations. An initial proposal to speed up the service establishment time in the presence of occasional packet loss and to reduce steady-state refresh signalling overhead has been made in [7]. One of the drawbacks of this approach is the requirement to change the protocol specification and to introduce an additional confirmation message into RSVP. An improved approach has been described in [8], which also deals with the general issue of reliability of RSVP messages, e.g., in case a service invocation is torn down. Instead of refreshing all the state information, neighbouring RSVP nodes only need to exchange 'heartbeats' denoting their liveliness. A slightly different suggestion addressing the same issue even more stringently is currently developed within the IETF RSVP working group [9]. This mechanism addresses further details, such as how to discover a very short-termed node failure.

It is beyond the scope of this work to rate these different techniques. However, they clearly bear the potential to drastically reduce RSVP's processing requirements for steady-state refresh signalling. This eliminates one of the major performance limitations of the current RSVP specification. Other RSVP extensions, which are in the process of being standardized, encompass diagnostic messages [10], inter-operation with IP tunnels [11], cryptographic authentication [12] and user identity representation [13].

III. KOM RSVP Engine

For an overview and explanation of RSVP, see [14]. Unfortunately, the only publicly available router implementation of RSVP, the ISI rsvpd, turns out to be of questionably design and coding quality and contains bugs. Therefore, we do not consider it as the optimal choice for experimenting with protocol extensions and performance assessments and we developed a new protocol engine from scratch, termed *KOM RSVP engine*, or *KOM rsvpd* for short. The main design goals of this implementation are clarity of code, flexibility and extensibility. Additionally, we aim at providing an experimental software platform for other researchers. Such an implementation on a regular workstation using a common UNIX operating system can only serve as a proof of concept and research platform for future investigations. Therefore, although we have tried to keep the design prepared for efficient operation, we do not believe that it is currently necessary to implement for outmost efficiency at the coding level. We have employed an object-oriented design and the implementation is done in C++. It is publicly available at
`http://www.kom.e-technik.tu-darmstadt.de/rsvp/`

A. *General Design*

A detailed description about the design of this implementation can be found in [15]. State information of RSVP is stored as objects containing relationships to other objects. The contents of a PATH message are stored in a Path State Block (PSB) whereas contents of a RESV message are stored in a *Reservation State Block* (RSB). As an example for relationships, each PSB has a relationship to a *Previous Hop State Block* (PHopSB) representing the hop from which this PATH message has been received. Information concerning a reservation at an outgoing interface is stored in an *Outgoing Interface State Block* (OutISB) and the relationship between reservations and PSBs is modelled as separate object *Outgoing Interface at PSB* (OIatPSB). It turns out that this object can serve as a crystalisation point to easily distinguish the operation context when calling the traffic control module, which is useful to collocate the respective code. Additionally, it is used to internally represent an N:M relationship by two 1:N relationships (which simplifies implementation). Figure 1 shows the entity-relationship diagram for the design of RSVP state information. Modelling RSVP state by an entity-relationship diagram is deemed useful both for documentation as well as efficient implementation through object-relationships [15].

Certain details regarding the generic design of the traffic and policy control interface are presented and discussed in [16]. A description of our overall vision of employing RSVP as a general service signalling protocol can be found in [17].

B. *Fuzzy Timers*

By far the largest container in an RSVP implementation is necessary for timer handling. In this implementation, a regular

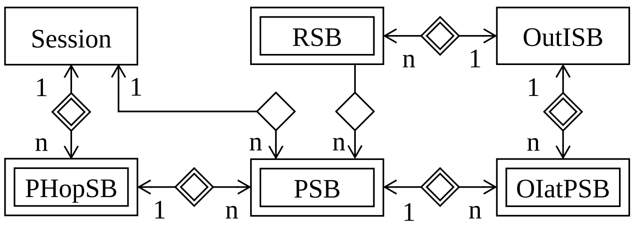

Fig. 1. Entity-relationship diagram for state blocks.

hash-based data structure is used as follows. All timers of the RSVP engine are stored in a hierarchical container. The upper layer is implemented as an array representing time slots and accessed through a hash. Individual time slots in the lower layer are implemented as sorted lists. The amount of time covered by each slot is configurable. Such a container is only capable to foresee a limited amount of time in the future, which should be sufficient for RSVP. In order to accommodate the rare event that timers exceed this time horizon, an additional sorted list is kept and timers from this list are moved into the respective slot when it becomes available. This concept is known as a *timer wheel* [18]. The access complexity of such an implementation is *O(log(n))*, with *n* being the (varying) number of timers in a slot. Consequently, performance of this container can be traded off against memory requirements by choosing the size and number of slots. This data structure design is shown in Figure 2.

For RSVP messages, this scheme can be optimized even further. RSVP is designed to be robust against varying message transmission times and in fact, a large number of all timers are calculated as random numbers within a certain interval. As a consequence, there is no demand for outmost precision in the scale of a few milliseconds. If the duration of a time slot in the hierarchy becomes small compared to the basic refresh time (e.g. smaller than 100 microseconds when the basic refresh interval is set to 30 seconds), an option to employ *fuzzy timers* is implemented. When enabling it, the timers within each time slot are stored in a simple FIFO list instead

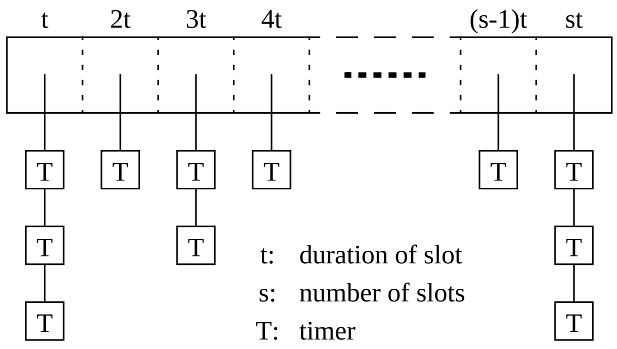

Fig. 2. Design of timer container.

of being sorted according to their precise expiration. During each time slot, timers are fired arbitrarily according to their location in the simple list. The result is a slight inaccuracy of timers compared to their expiration time. The inaccuracy is bounded by the length of a time slot and can be considered a very reasonable trade-off. In principle, this scheme promises a performance gain over the plain timer wheel, because the access complexity is reduced to *O(1)*. However, because of the generally small number of timers stored in one time slot, such performance gains are hardly visible in reality. On the other hand, as discussed in Section V.C, this design can be used to improve interaction with the operating system.

C. *Multi-threaded Message Processing*

Employing the innovative design of the RSVP engine, it was possible to quite easily replace the initial sequential message processing by a multi-threaded protocol engine with an incremental implementation effort of about 6 weeks. A fixed number of worker threads can be used to concurrently process RSVP messages. Because of a current lack of system support, certain interactions with the operating system, e.g. the reception of raw IP packets, cannot be performed truly multi-threaded. Therefore, those operations are currently carried out sequentially. As a consequence, in addition to the worker threads, there is a dedicated thread to initially receive and dispatch protocol messages. Furthermore, a separate thread is created to handle timer events. Synchronisation points are set at

- access to the central state repository (synchronisation point per session),
- interfaces to traffic control (synchronisation point per interface),
- access to the central timer management (global synchronisation point), and
- access to certain system services (global synchronisation point, see above).

The design of multi-threaded message processing is sketched in Figure 3. There are two options to employ this design, which can be chosen at compile time. The first option allows for an arbitrary number of worker threads, simulating the situation of a router possessing multiple CPUs as control engine. The other alternative tries to mimick operation in a potential high-end router, which has a dedicated CPU at each network interface.

Of course, using multiple threads on a single-CPU workstation cannot be expected to significantly increase performance other than potentially providing improved interaction with external I/O operations.

The design could be further improved. For example, the global lock for the timer system could be replaced by more fine-grained locking for each slot of the timer container. On the other hand, with the fuzzy timer scheme, access to the timer container is not as time-consuming and critical as with a

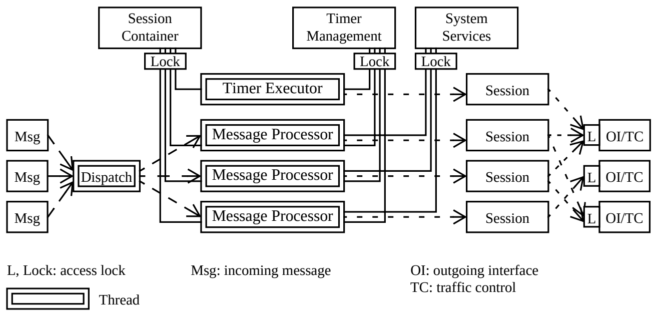

Fig. 3. Multi-threaded message processing.

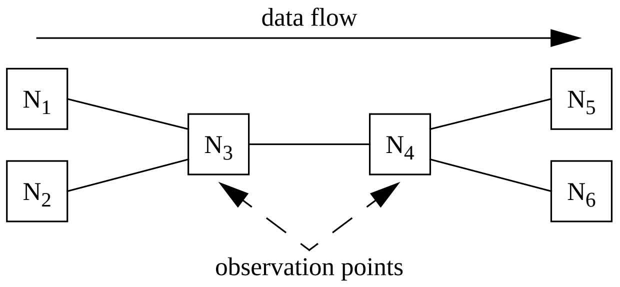

Fig. 4. Setup for performance measurements.

sorted container. To this end, the purpose of this multi-threading extension is to demonstrate the simplicity and feasibility of parallelizing RSVP operations as a proof of concept. Indicative performance tests have been carried out and are described in Section V.D.

It becomes very obvious that the object-relationship design alleviates the task of parallelizing message processing a lot. The reasons are given by the natural encapsulation of data and procedures in an object-oriented design. This allows for easy identification of synchronisation points. Because all state objects are stored and accessed through the session object, no additional locking is necessary for them, besides acquiring a single lock for the session.

IV. Experiment Setup

The performance experiments were carried out on standard PC-based workstations, which serve as a router platform running FreeBSD 3.4. These workstations are equipped as follows:

- single Pentium III processor, 450 MHz, 512KB cache
- point-to-point 100 Mbit/sec Ethernet links, 3Com 3c905C-TX interface cards
- Gigabyte GA-6BXU mainboard, standard hard disk
- 128 MB RAM main memory

The total cost of this equipment as of December 1999 is approximately 600 Euros plus 50 Euros per network interface. For the tests, 6 nodes are connected with each other as depicted in Figure 4. N_5 is used as destination host and N_1 as source host. Multiple unicast sessions are created by specifying multiple port numbers. Since handling of API sessions creates additional overhead at the respective end node, N_2 and N_6 are used as additional source and destination hosts, if a large number of sessions is created. The RSVP refresh interval is set to 30 seconds, as suggested in [1]. The RSVP daemons run in single-threaded mode (except for Section V.D) and exchange basic RSVP messages only, without policy data and integrity objects. However, all experiments encompass the generation and transmission of confirmation messages.

The load generator at N_1 (N_2) creates sessions and path advertisements with a randomized time interval in between, until a certain number of sessions n is reached. The upper bound of this time interval can be chosen for each experiment. When the target number of sessions is reached, the load generator creates and deletes sessions with the same randomized time interval respectively, in a way that the number of sessions is kept in the interval $[n-10, n]$. The receiver at N_5 (N_6) responds to each path advertisement by immediately generating reservation requests, which establish the end-to-end flow reservation.

The observations are made at Node N_3 and N_4. Measurements are done by periodically executing `top` and recording the highest numbers for current total memory consumption and percentage of raw CPU time that is reported for execution of the RSVP daemon on either node. Note that this kind of measurement introduces some inaccuracies and inherent randomness, which however should not mask the principle message of the results.

V. Experimental Results

In order to assess the performance of an RSVP implementation and to address the usual concerns against its processing overhead, a number of performance experiments have been carried out. It is important to mention at this point that the KOM RSVP engine has not been subject to careful and detailed tuning at the coding level. No specific optimization has been carried out, other than the general design decisions and algorithmic improvements described earlier.

The first series of tests compares the performance of the KOM rsvpd with the ISI rsvpd. The second series investigates the current performance limits of the KOM rsvpd and the following experiments analyse the effect of algorithmic improvements that have been implemented. Additionally, an experiment is reported, which investigates the influence of the average flow lifetime on the processing effort. Finally, some experiments have been carried out to obtain additional interesting performance figures, e.g., about the end-to-end setup latency.

A. Comparison with Existing Work

For this first series of tests, no specific optimizations have been turned on in the KOM RSVP engine. The timer container has been configured to consist of 20,000 slots covering 50 milliseconds each. Both implementations have been compiled with the same optimization and debugging flags. The hash-based session container does not provide any performance gain for either alternative, because all flows are targeted to the same host.

The ISI rsvpd contains bugs, which basically prohibit testing scenarios that involve the deletion of multiple sessions. An investigation of this problem revealed at least one non-trivial error in the memory management to be responsible for this situation. It is quite easily possible to fix the most prevalent problem, such that the software does not crash too often, but as a result, memory leaks prohibit reasonable operation.

Because of these problems, the performance figures for the ISI rsvpd can be considered as valid only to a limited extent. We have chosen not to fix the above bug to avoid memory leaks, which otherwise result in an infinite increase of processing effort and memory consumption and thus, preclude to obtain realistic performance figures. As a result, the numbers for the ISI rsvpd can only be considered as a lower bound for CPU consumption, because it always crashes before a stable situation with creation and removal of sessions can be reached. The listed results consequently show the situation just before the crash. With the KOM rsvpd, each test has run for several minutes. The listed percentage of CPU time is the highest number that has been observed during that time. The memory consumption has always stabilized at the reported amount. The results are depicted in Table I.

The load generator used in these experiments first establishes the configured number of flows and then circulates through the flows and removes and re-establishes them. In order to vary the lifetime of each flow, the interval between creation respectively removal of subsequent flows is adjusted. Due to space limitations, this is not shown directly in the tables. Instead, the average lifetime of a single flow is shown, which is calculated according to the fact that the creation/removal interval is evenly distributed between zero and the maximum interval. The maximum creation/removal interval is set to 25 milliseconds for the tests with 2000 flows and more. Therefore, the average lifetime increases with an increasing number of flows. Note that the interval is appropriately adapted for the tests with smaller numbers of flows, such that the average lifetime of flows is not much smaller than the RSVP refresh interval. The influence of the average flow lifetime is further studied in Section V.E.

It can be derived from these performance figures, that the KOM RSVP engine performs significantly more efficiently than the ISI rsvpd. While it is unclear how much of this efficiency gain has to be attributed to a better coding style in general, it can obviously be concluded that the innovative object-relationship design at least does not prohibit performant implementation, however, at the expense of additional memory consumption. The KOM rsvpd consumes almost twice the amount of memory per flow when compared to the ISI rsvpd

TABLE I
PERFORMANCE OF ISI RSVPD VS. KOM RSVPD

Experiment settings		ISI rsvpd		KOM rsvpd	
Flows	Avg. lifetime	% CPU	Memory	% CPU	Memory
0	--	0.00	1920K	0.00	2724K
500	25.00 sec	2.05	2372K	1.13	3620K
1000	25.00 sec	6.18	2856K	3.56	4544K
1500	28.50 sec	10.01	3296K	5.32	5472K
2000	25.00 sec	14.89	3768K	7.37	6388K
2500	31.25 sec	20.51	4244K	9.91	7308K
3000	37.50 sec	25.93	4728K	13.38	8236K
3500	43.75 sec	33.74	5208K	16.60	9160K
4000	50.00 sec	42.53	5692K	20.26	10084K
4500	56.25 sec	51.37	6168K	23.73	11008K
5000	62.50 sec	60.45	6656K	27.83	11928K
5500	68.75 sec	79.69*	7140K	32.96	12848K

* number of successful reservations: ~ 5400

numbers. This can be attributed to the fine-grained implementation of state relationships, but also to the fact that memory consumption was not the primary goal for optimization of our implementation.

B. *Performance Limits*

The goal of this set of tests is to find the upper limits on the number of reservation requests for a tuned version of the RSVP implementation. The experiment setup and measurements have been done as described above. In the tuned version, the timer container consists of 100,000 slots covering 10 milliseconds each and the code for API processing is disabled at intermediate nodes. Assertion checking and debug output is turned off. Since these tests are carried out in a limited infrastructure with at most two destinations hosts, port numbers are included into the hash calculation for the session container in the tuned version. Because doing so establishes a perfect hash distribution for the test scenario, the session hash index has been restricted to 4096 to simulate a realistic situation. Furthermore, the load generation is distributed between all four end nodes as depicted in Figure 4. The results are listed in Table II.

The following observations can be made in this experiment. Tuning the protocol implementation reveals a significant potential for increasing the performance. A router platform based on standard PC hardware can handle the full signalling for 50,000 unicast flows. The larger amount of initially allocated memory for the tuned version can be attributed to the additional memory requirements for the more fine-grained timer container. The memory requirements per flow remain unaffected. Two additional tests are listed, in which the crea-

TABLE II
PERFORMANCE LIMITS OF KOM RSVPD

Experiment settings		basic KOM rsvpd (load gen. by 2 nodes)		tuned KOM rsvpd (load gen. by 4 nodes)	
Flows	Avg. lifetime	% CPU	Memory	% CPU	Memory
0	--	0.00	2724K	0.00	4724K
2500	31.25 sec	9.91	7308K	4.39	9324K
5000	62.50 sec	27.83	11928K	8.50	13940K
7500	93.75 sec	58.11	16548K	11.38	18560K
9800	122.50 sec	93.12	20788K	--	--
10000	125.00 sec	65.00*	21156K	14.75	23168K
15000	187.50 sec	--	--	20.95	32396K
20000	250.00 sec	--	--	27.73	41632K
30000	375.00 sec	--	--	40.67	60096K
40000	500.00 sec	--	--	55.17	78556K
50000	625.00 sec	--	--	67.99	97012K
40000	240.00 sec	--	--	56.69	78556K
50000	250.00 sec	--	--	70.56	97012K

* load generated by 4 nodes, see main text

tion/removal interval is set in a way that the average lifetime of a flow is approximately 4 minutes. The resulting CPU load numbers demonstrate that the RSVP engine is indeed able to handle such a large number of sessions, even when assuming a realistic average lifetime of calls. In fact, the impact of the lifetime of flows seems to be quite low. Further details are discussed in Section V.E.

One particular detail can be observed when comparing the CPU load numbers for the basic version in Table II, depending on how many nodes participate in load generation. If four end nodes are used, the resulting load is substantially smaller and the performance limit is increased. The explanation of this behaviour is related to the implementation of the timer wheel in combination with the `select` system call, which is used to query for incoming packets. If four end nodes participate in load generation, messages arrive at intermediate nodes at three network interfaces, instead of two. Each switch between timer management and message reception incurs a call to `select`, which is expensive. It takes at least 10 milliseconds on a regular Linux, Solaris and FreeBSD operating system to perform this system call when a timeout is given. After `select` returns, exactly one message is read from each eligible interface. Now, if messages arrive at more interfaces, more messages are potentially received, before the next invocation of timer management. This leads to less context-switching between message reception and timer management and thus, reduces the total number of system calls, which in turn decreases the system load.

Figure 5 shows an overall picture of the experimental results from this and the previous section. The graph depicts

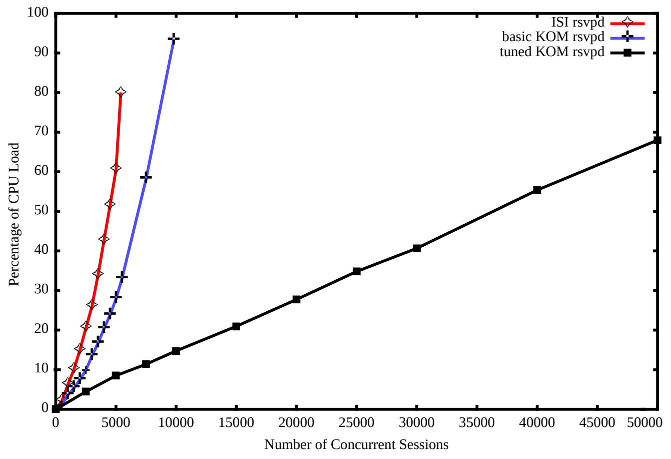

Fig. 5. Performance curve for ISI and KOM rsvpd.

the fraction of CPU load as a function of the number of sessions.

C. Fuzzy Timer Handling

While in theory only fuzzy timer handling can guarantee the property of overall linear complexity by simplifying access to the timer container, the previous experiment shows that, by enabling a fine-grained timer wheel, this linearity is already observed. In fact, a further modification of implementing fuzzy timers is needed to achieve any visible improvement at all. Because of the effects of switching between timer management and interface service, which is described in the previous section, all timers from the current slot are fired whenever the system enters the timer management. This further reduces the number of context switches and calls to `select` and consequently, the overall processing load. A comparison with regular operation, which indicates the additional performance gain, mainly at a high session load, is shown in Table III. At a load of about 58,000 flows, the system exceeds the maximum amount of main memory that is available and starts swapping to disk. This prohibits any further performant execution under this high load.

It turns out that there is a triangular relationship between the kernel clock granularity, the minimum timeout needed for the `select` call and the size of the time slots. It might actu-

TABLE III
PERFORMANCE OF FUZZY TIMER OPTIMIZATION

Experiment settings		tuned KOM rsvpd		fuzzy KOM rsvpd	
Flows	Avg. lifetime	% CPU	Memory	% CPU	Memory
0	--	0.00	4724K	0.00	4724K
20000	250.00 sec	27.73	41632K	26.12	41632K
40000	240.00 sec	56.69	78556K	53.37	78556K
50000	250.00 sec	70.56	97012K	63.96	97012K
58000	232.00 sec	--	--	~70.00	>108M

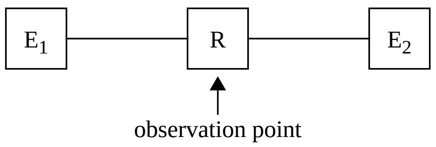

Fig. 6. Experiment setup for parallel processing.

ally be possible to increase the performance limits by increasing the range of time slots or increasing the clock granularity. However, we have not done any such experiments, so far.

D. Parallel Message Processing

This experiment is carried out to investigate the scalability of multi-threaded message processing on a multi-processor platform. This experiment uses the first alternative to employ multi-threaded message processing as presented in Section III.C, in that each message processing thread is bound to a specific network interface. The experiment setup is very simple and shown in Figure 6. The end-systems E_1 and E_2 are the same PCs as in the other experiments and are connected to a router R. Both end-systems act as sender and receiver and create a large number of flows. A SparcServer 1000 with four 60Mhz CPUs running Solaris 2.6 serves as router. Note that this router hardware provides significantly less absolute processing power compared to the other tests. Because a separate thread is needed in the RSVP daemon to receive raw IP packets and dispatch them to the worker threads and another thread is used for timer handling, at least four CPUs are needed to carry out a reasonable experiment for this scenario.

In order to test the capabilities of this system, tests have been run in single-threaded mode and in multi-threaded mode with enabling an increasing numbers of CPUs. The goal of each test is to find the highest number of flows that can be handled reliably. Therefore, the RSVP daemon has been slightly modified to regularly check the difference between the number of PSB and RSB objects. If this difference exceeds a certain threshold, the daemon stops and reports the number of successfully established reservations. Because the total number of flows that can be sustained by this router is rather small, the RSVP refresh time is set to 3 seconds in order to increase the effect of established sessions compared to the creation of new ones. As well, to decrease the high influence of system code, which cannot be executed truly multi-threaded, the software is compiled without compiler optimization. The results are listed in Table IV. Each test is executed ten times and both the highest and lowest result are not taken into account for calculating the average result.

It becomes clear from the resulting performance figures, that the potential for parallelization gains is indeed given, but certainly limited, at least on the tested platform. Furthermore, when comparing the results for single-threaded execution with those of multi-threaded execution on a single CPU, a significant overhead for synchronization mechanisms can be observed. These limitations must be partially accounted to the insufficient support of the operating system to support multi-threaded reception of raw IP packets and other low-level services, but also to inherent parallelization limitations of RSVP processing and the improvable implementation design of the parallel code in the KOM protocol engine. To this end, the result of implementing multi-threaded message processing is somewhat unsatisfactory. On the other hand, the design and implementation of multi-threaded message processing should be considered as a proof of concept, rather than the final design of a production-level implementation. Especially, with proper operating system support, the need for a separate dispatcher thread (which might very well form the bottleneck of the current system) and its synchronisation would be eliminated. As discussed in Section V.B and Section V.C, the overall performance of the RSVP daemon is to a great extent determined by the system-level task of receiving packets from the network and the particular interaction with the user-level daemon. This assumption is further verified in Section VI.

TABLE IV
PERFORMANCE OF PARALLEL MESSAGE PROCESSING

Number of CPUs	Number of flows (individual tests)	Number of flows (average)
single-threaded	451, 425, 464, 473, 466, 450, 450, 494, 520, 489	467
1	345, 389, 386, 380, 373, 373, 393, 350, 357, 366	371
2	552, 478, 571, 605, 571, 532, 563, 556, 518, 572	554
3	707, 723, 693, 756, 731, 718, 711, 702, 729, 727	719
4	592, 621, 711, 662, 652, 655, 648, 666, 655, 663	653

Another conclusion can be drawn from these tests, which backs up the above considerations. Testing the efficiency gain of a multi-threaded RSVP implementation on a simple and small multi-processor workstation as in these tests, is probably not sufficient to fully reveal parallel processing efficiency. For example, the performance drop when observing execution on 4 CPUs can be explained as follows. On this platform, up to 3 CPUs can be bound to a process (process group) exclusively. Effectively, in the tests with 4 CPUs no exclusive binding of CPUs can been done and therefore, the RSVP daemon competes with other processes. Consequently, the overall scheduling effort increases for the operating system. This is reflected by the lower performance and indicates, that these tests cannot be regarded as real tests with 4 CPUs.

As discussed in Section III.C, there is a broad field for further work on tuning the design and implementation of multi-threaded RSVP operations. Additionally, it would be very desirable to compare the results obtained during these tests with performance figures from different hardware and operating system platforms.

TABLE V
INFLUENCE OF AVERAGE FLOW LIFETIME

Experiment settings		fuzzy KOM rsvpd	
Flows	Average lifetime	% CPU	Memory
10000	150.00 sec	13.96	23168K
10000	125.00 sec	14.75	23168K
10000	100.00 sec	14.99	23168K
10000	50.00 sec	15.77	23168K
10000	25.00 sec	16.65	23168K
10000	15.00 sec	21.48	23168K
10000	5.00 sec	77.10*	23168K

* number of successful reservations: ~ 9700

E. *Lifetime of Flows*

The experiments in Section V.B and Section V.C indicate that the average lifetime of flows has only limited influence on the computational effort. In order to further investigate this issue, a dedicated set of tests has been done to examine this effect. The results are listed in Table V.

The figures clearly show, that the resulting CPU load for a certain number of flows is largely unaffected by the average lifetime of flows, as long as it is above the RSVP refresh interval (again set to 30 seconds here). Thereby, these numbers back up the conjecture that the average lifetime of flows has only limited influence on the overall processing effort. Since RSVP state is refreshed periodically, the average flow lifetime basically determines the ratio of setup messages compared to refresh messages, whereas the overall number of messages is approximately the same. It can thus be concluded that there is not much difference between the individual processing effort for setup messages and for refresh messages. Consequently, when flow lifetimes are multiples of the refresh interval, a large fraction of processing effort is due to refresh messages.

Indirectly, this result demonstrates the large potential for performance gains by extending RSVP with mechanisms to reduce the amount of state refresh messages, like those referred to in Section II. However, this particular behaviour could also be an artefact of this specific implementation. Therefore, further work covering different implementations would be needed to investigate the details. Unfortunately, at this time, no such implementation is available. The ISI rsvpd cannot reliably handle the deletion of sessions, hence, this kind of experiment is currently not possible.

If the lifetime of flows becomes significantly shorter than the refresh interval, this generates an absolute increase in the number of RSVP messages and results in a much higher processing load. In fact, for these cases, it can be noticed in Table V that the increase of CPU load is approximately inverse proportional to the lifetime of flows.

F. *Other Experiments*

Some other experiments have been carried out to assess this implementation under a variety of aspects. Because their results are highly bound to the specific scenario, they are somewhat illustrative, but they may not be regarded as relevant as the above experiments. Hence, they are not documented here in the same level of detail.

1) RSVP & Packet Classification

We have done the following experiment in order to measure the combined throughput of signalling and packet classification and scheduling. Along two adjacent FreeBSD-based routers reservations for 10,000 flows are established, similar to the earlier experiments. Each of these flows requests a small amount of bandwidth. Then, a traffic source emits a constant packet stream, which belongs to one of the reserved flows. Although this packet stream exceeds the reservation by far, an intermediate node must still classify and schedule all packets. We observed that in combination with HFSC scheduling from the ALTQ package [19], two adjacent routers running KOM rsvpd can both sustain the signalling and traffic control configuration for 10,000 flows and at the same time, classify and schedule 25,000 packets per second. Note that this result does not make any statement about the aspect whether all flows actually receive their QoS objective. Evaluating the ALTQ package is beyond the scope of this paper.

2) End-to-End Setup Latency

Using the setup shown in Figure 7, tests have been carried out to measure the setup latency of RSVP requests. R_0 and R_3 are not handling any background RSVP session. R_1 and R_2 are loaded with up to 20,000 flows. The total end-to-end setup latency usually varied between 22 and 26 milliseconds, independent of the load of intermediate routers. Consequently, the latency of bidirectional session setup can be estimated to be at most 5-6 milliseconds per intermediate hop, which shows that even along a path with a large number of hops, the end-to-end setup latency will very likely be acceptable.

VI. PROFILING DETAILS

In order to further investigate the performance of our RSVP implementation, we generated profiling information from an experiment equivalent to those described in Section V. The protocol engine was compiled for optimized execution as in Section V.C and has been loaded with the signalling of 20,000 unicast flows. Table VI shows the execution times for various operations, which represent complete and non- overlapping

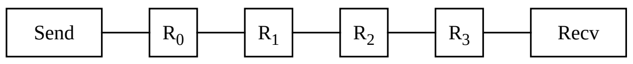

Fig. 7. Experiment setup for end-to-end latency.

TABLE VI
RELATIVE PROCESSING EFFORT OF PROTOCOL OPERATIONS

Operation	% Execution Time
System initialization/cleanup	1.8
Packet awaiting	9.3
Packet reception	6.8
Packet parsing	10.7
Message processing	53.1
Timer handling	18.3
Total	100.0

partitions of the overall message processing. Because of the characteristics of profiling, the execution time here is relative to the daemon's overall accumulated processing effort.

It can be observed from this table, that packet parsing consumes a significant amount of execution time. This can be attributed to the fact that we have not spent effort to optimize the code for parsing RSVP messages. In Table VI, the figures for *packet awaiting* and *packet reception* denote pure system activities, i.e., interaction with the networking stack. The operation of sending out packets is encompassed in *message processing*. The execution time of further system activities, namely sending out packets and looking up routing entries are encompassed in *timer handling*. Details about system operations are illustrated in Table IX below. In Table VII, we describe how the effort for message processing is further subdivided among more fine-grained operations.

Not surprisingly, PATH processing consumes the major amount of execution time, mainly due to looking up the routing information for the destination address (see below). Note that due to their similarity, the operations for processing a RESV and RTEAR message are implemented in the same method, therefore the execution time cannot be subdivided between both.

Another interesting investigation is to analyse the effort necessary for management of the timer system. This number

TABLE VIII
RELATIVE PROCESSING EFFORT OF TIMER MANAGEMENT

Operation	% Execution Time
Timer insertion & removal	7.2
Timer maintenance & firing	4.1
Total	11.3

is somewhat difficult to assess, because of code inlining and optimized compilation. The results are listed in Table VIII. These are the raw numbers for timer management, i.e., excluding the execution times for subsequent actions. They explain the limited effect of fuzzy timer handling on top of an efficient timer container structure. Note that these numbers are not related to the execution time reported for *timer handling* in Table VI. The numbers here denote the raw effort for timer management, excluding for example the operation that is carried out when a timer is fired.

Table IX illustrates how much execution time is spent for carrying out system services. The operations listed in this table are only those contributing significantly. It turns out that the RSVP engine executes system-level code for more than 70% of its time, large parts of this time interacting with the kernel. Considering the restrictions discussed in Section III.C, these figures explain the limited performance gains by parallelizing message processing.

It turns out that looking up routing entries contributes significantly to the overall execution time for system services. This effect can be explained by the rather expensive routing interface on FreeBSD, which requires at least two interactions with the operating system's kernel in order to obtain a routing entry. This interface might bear the potential for optimization, at least in case of unicast routing lookups, which only deliver a single routing entry as result.

Finally, memory management in general can be observed as strongly contributing to the overall execution time. Additional performance gains might be possible by replacing the system's universal memory management algorithms by a mem-

TABLE VII
RELATIVE PROCESSING EFFORT OF MESSAGE PROCESSING OPERATIONS

Operation	% Execution Time
Pre- and postprocessing	5.6
Session location	1.6
PATH processing	28.3
PTEAR processing	0.8
RESV/RTEAR processing	10.6
CONF message forwarding	3.4
Refresh reservations	2.8
Total	53.1

TABLE IX
RELATIVE PROCESSING EFFORT OF SYSTEM SERVICES

Operation	% Execution Time
Routing lookup for PATH messages	16.2
Routing lookup for RCONF messages	2.5
Packet awaiting	9.3
Packet reception	6.8
Packet sending	8.8
System time lookup	8.2
Memory management (total)	20.0
Total	71.8

ory management system, which is specifically optimized for the type of operations needed for RSVP processing.

In general, the data generated from profiling explain the relation between certain results from the performance tests and back up assumptions about the internals of this implementation. They might serve as a basis for future detailed code optimization of this or the design of other code.

VII. Conclusions and Future Work

The assessment of RSVP's technical feasibility started with collecting and analysing the available material. Very soon it became obvious that the publicly available code as well as previously published work were not sufficient to study the aspects that were deemed interesting for this work. Therefore, a new implementation of RSVP has been developed from scratch. It employs the notion of objects and relationships between them to efficiently store and access protocol state. It is innovative in its design and for example, allows easy inclusion of multi-threaded message processing. Furthermore, certain design and algorithmic extensions for the implementation of an RSVP engine have been proposed in Section III. A high potential for performance gains has been demonstrated by tuning the implementation appropriately.

In the performance experiments of Section V, RSVP has been evaluated with respect to its basic mode of operation. The main goal of this work is to show the performance potential, even without further changes to the protocol. From the performance figures, it can be deduced that the suitability of RSVP as a general purpose signalling interface and protocol is much better than generally assumed. A standard PC router, at equipment cost of about 600 Euros (plus 50 Euros per network interface, as of December 1999), can handle the signalling for more than 50,000 sessions in a realistic scenario.

Essentially, the user-level RSVP implementation presented in this paper is not the bottleneck for operation on a standard UNIX platform. Instead, the execution of system services largely determines the overall performance. This can be concluded from the experimental results, including those measuring the capabilities of multi-threaded message processing and is further backed up through profiling experiments. Consequently, further work, especially on different hardware and operating system platforms, is needed to better understand the ultimate limits of an RSVP engine. As discussed in Section V.C, further experiments can be carried out, which investigate the effect of the clock granularity and size of time slots on a FreeBSD platform.

Implementation of a software platform for general end-to-end service signalling remains an ongoing effort for us. We are planning to investigate the effects of RSVP extensions as referred to in Section II.B on its feasibility for the purpose of serving as a general signalling protocol. Furthermore, we will focus our future work on inter-operation between RSVP signalling and data-forwarding technologies, for example DiffServ, MPLS or ECN. Last but not least, we plan to study the impact of mobility and to design general solutions for the inclusion of mobile routing protocols into an overall QoS signalling architecture.

References

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

[2] A. Neogi, T. Chiueh, and P. Stirpe. Performance analysis of an RSVP-capable router. *IEEE Network Magazine*, 13(5):56–63, September 1999.

[3] I. Cselenyi, G. Feher, and K. Nemeth. Benchmarking of signaling based resource reservation in the Internet. In *Proceedings of Networking 2000*, pp. 643–654. Springer LNCS 1815, May 2000.

[4] USC Information Sciences Institute. RSVP Software. http://www.isi.edu/div7/rsvp/release.html.

[5] E. Basturk, A. Birman, G. Delp, R. Guerin, R. Haas, S. Kamat, D. Kandlur, P. Pan, D. Pendarakis, V. Peris, R. Rajan, D. Saha, and D. Williams. Design and implementation of a QoS capable switch-router. *Computer Networks*, 11(1-2):19–32, January 1999.

[6] P. Pan and H. Schulzrinne. Yessir: A simple reservation mechanism for the Internet. *ACM Computer Communication Review*, 29(2):89–101, April 1999.

[7] P. Pan and H. Schulzrinne. Staged refresh timers for RSVP. In *Proceedings of Global Internet'97, Phoenix, Arizona, USA*, November 1997. also IBM Research Technical Report TC20966.

[8] L. Mathy, D. Hutchison, S. Schmid, and S. Simpson. REDO RSVP: Efficient signalling for multimedia in the Internet. In *Interactive Distributed Multimedia Systems and Telecommunication Services*. Springer LNCS 1718, October 1999.

[9] L. Berger, D.-H. Gan, G. Swallow, P. Pan, F. Tommasi, and S. Molendini. RSVP Refresh overhead reduction extensions. Internet Draft draft-ietf-rsvp-refresh-reduct-05.txt, June 2000. Work in Progress.

[10] A. Terzis, R. Braden, S. Vincent, and L. Zhang. RFC 2745 - RSVP Diagnostic messages. Standards Track RFC, January 2000.

[11] A. Terzis, J. Krawczyk, J. Wroclawski, and L. Zhang. RFC 2746 - RSVP Operation over IP tunnels. Standards Track RFC, January 2000.

[12] F. Baker, B. Lindell, and M. Talwar. RFC 2747 - RSVP Cryptographic authentication. Standards Track RFC, January 2000.

[13] S. Yadav, R. Yavatkar, R. Pabbati, P. Ford, T. Moore, and S. Herzog. RFC 2752 - Identity representation for RSVP. Standards Track RFC, January 2000.

[14] P. White and J. Crowcroft. Integrated services in the Internet: State of the art. *Proceedings of IEEE*, 85(12):1934–1946, December 1997.

[15] M. Karsten. Design and implementation of RSVP based on object-relationships. In *Proceedings of Networking 2000, Paris, France*, pp. 325–336. Springer LNCS 1815, May 2000.

[16] M. Karsten, J. Schmitt, and R. Steinmetz. Generalizing RSVP's traffic and policy control interface. In *Proceedings of the 7th International Conference on Parallel and Distributed Systems Workshops (ICPADS'00), Iwate, Japan*, pp. 249–254. IEEE, Piscatay Way, NJ, USA, July 2000.

[17] M. Karsten, J. Schmitt, N. Berier, and R. Steinmetz. On the feasibility of RSVP as general signalling interface. In *Proceedings of Quality of future Internet Services Workshop (QofIS 2000), Berlin, Germany*, pp. 105–116. Springer LNCS 1922, September 2000.

[18] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. *Operating Systems Review Special Issue: Proceedings of the Eleventh Symposium on Operating Systems Principles, Austin, TX, USA*, 21(5):25–38, November 1987.

[19] K. Cho. A framework for alternate queueing: Towards traffic management by PC-UNIX based routers. In *Proceedings of USENIX 1998 Annual Technical Conference, New Orleans, LA, USA*, June 1998.