Collected Experience from Implementing RSVP

Martin Karsten*, Version: Feb 2005
School of Computer Science, University of Waterloo, 200 University Ave W, Waterloo ON N2L 3G1, Canada
mkarsten@uwaterloo.ca

***Abstract* — Internet QoS is still a highly debated topic for more than ten years. Even with the large variety of QoS proposals and the impressive research advances, there is still little deployment of network layer QoS technology. One specific problem domain is QoS signalling, which has recently attracted increasing attention to bring forward new standardization approaches. In this article, an extensive study of RSVP is presented, covering protocol design, software design and performance aspects of the basic version of RSVP and of certain standardized and experimental extensions. The work is based on and presents the experience from implementing RSVP for UNIX systems and the ns-2 simulation environment, including a variety of protocol extensions and incorporating a number of internal improvements. The implementation has been subject to extensive functional and performance evaluations, the results of which are reported here.**

I. Introduction

Quality of Service (QoS) and particularly QoS signalling in the Internet are highly debated topics for more than ten years. Recently, QoS signalling has attracted increasing attention resulting in new standardization efforts [1]. RSVP [2] is a QoS signalling protocol that is proposed as IETF standard. It has been discussed extensively, but most of the discussion often seems to be based on second-hand knowledge rather than proper analysis or first-hand experience. While it is clearly arguable whether RSVP's initial design goals still hold true in the current and future Internet environment, it is generally important and the major goal of this work to thoroughly study RSVP, before precipitously designing successor protocols. The work presented here summarizes, updates and consolidates earlier publications on this topic, which are cited if they provide crucial additional details. It is not at all intended to give any final judgement about the suitability of RSVP for any specific purpose, but rather aims to deliver an unbiased and comprehensive evaluation of its strengths and weaknesses from a practical implementation point of view.

This article is organized as follows. In the next section, basic considerations for QoS signalling are reviewed to set the context for a conceptual discussion of RSVP's design properties in Section III. In Section IV, the design and implementation of a protocol engine with standardized extensions is presented, while the details and results of various experimental studies are reported in Section V. Some experimental protocol extensions are presented and discussed in Section VI, again in terms of implementation and experimental evaluation. A survey of related work can be found in Section VII and the article is concluded in Section VIII with a summary of the overall findings and results, as well as an outlook to future work.

*. Parts of this work were carried out while the author was with the Multimedia Communications Lab at Darmstadt University of Technology, Germany.

II. QoS Signalling Implementation - Fundamentals

Internet routers typically employ a kernel-based operating system similar to UNIX. The packet processing and forwarding path is implemented in kernel space and often executes on dedicated hardware while control-level mechanisms, for example routing protocol engines, are run as user-level daemons on the main processor. Given this type of platform, there is a choice to implement a QoS signalling protocol as part of the operating system kernel or as a user-level daemon. There are several aspects that need to be taken into account when making that decision. In practice, the higher the complexity of a protocol engine, the more likely it is to be implemented as a user-level daemon. This is due to the stronger robustness of user-level software through process and memory management services, increased flexibility with respect to updating core functionality, and easier realization of interactions with external components, for example a policy system [3].

For intra-domain QoS signalling, a protocol's main task is to implement a distributed algorithm to achieve a common goal. In case of inter-domain QoS signalling, however, a signalling protocol mainly serves as a service interface between adjacent network domains. Both cases differ a lot in terms of their requirements. In the intra-domain case, there is a higher level of trust and common interest among multiple participating nodes. Efficiency is then the prime goal for all technologies being applied. In the inter-domain case, there is less trust and common interest. For example, accountability, security, and non-repudiation are major concerns when it comes to negotiating service requests and likely require a more complex QoS signalling protocol. Also, these aspects likely require an implementation that allows for flexible interaction with external components. Existing proposals address a different subset of these aspects. In that spectrum, RSVP is probably best regarded as hybrid proposal for a uniform protocol that can be employed both within a network domain at every node, but also as inter-domain service interface. It is not at all clear whether RSVP is most suitable as either an intra- or inter-domain QoS signalling protocol, or whether alternative proposals will eventually dominate in terms of functionality and efficiency. However, a detailed understanding of these different requirements is necessary to adequately decide between a user- or kernel-level protocol implementation.

The work presented here is focused on plain RSVP signalling only and as such, no assumptions are being made about either of the above cases. However, because of the significant complexity of RSVP as well as the potential interaction with external components, it has been decided to implement the protocol engine as a user-level process.

III. Protocol Design of RSVP

In this section, the design of RSVP is discussed informally, based on available earlier work and the author's implementation experiences. For a detailed introduction into RSVP, the reader is referred to a tutorial article [4].

A. Processing Overview

As with all IETF standard protocols, there is no formal specification of a finite state machine for RSVP. Nevertheless, it can be very useful to introduce the notion of at least an informal processing diagram to properly present and understand the functionality of a protocol implementation. Fig. 1 shows the illustration of such a diagram for the four main messages that potentially change internal signalling state (PATH, PTEAR, RESV, RTEAR). Error and confirmation messages (PERR, RERR, RCONF) are merely forwarded by an RSVP daemon and are thus not captured by the processing diagram. Interactions with the traffic control subsystem are not explicitly shown, as well.

Beginning with *State Update*, each processing step may or may not result in changes of internal state information. The respective next processing step is only executed, if state information is changed during the previous one.

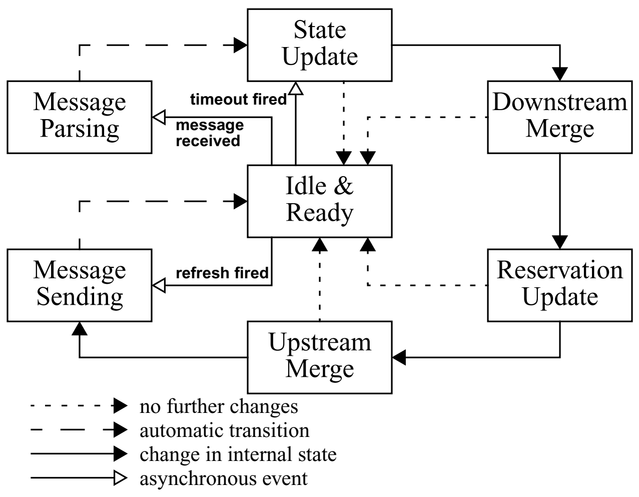

Figure 1: RSVP Processing Diagram.

During *State Update*, it is checked whether an incoming message merely refreshes existing state (PATH, RESV) or does not refer to existing state at all (PTEAR, RTEAR). If yes, state timers are revised and message processing terminates. For all other messages, reservation aggregation for multiple downstream hops behind an outgoing interface (multicast) is handled in *Downstream Merge*. The resulting reservation state is communicated to the packet scheduler during *Reservation Update*. Reservation aggregation across multiple outgoing interfaces (multicast) and for multiple senders sending through the same incoming interface is handled in *Upstream Merge*. If during any of the processing steps no further changes have to be propagated throughout the system, message processing is terminated. Whenever state is created or refreshed during *State Update*, a timeout timer is started. Upon expiration of such a timeout, the corresponding state is deleted and the remaining processing sequence is invoked. If a new outgoing message is created, a refresh timer is started which triggers the periodic retransmission of this message. As a result, the pure processing diagram appears to be relatively simple. However, in combination with the variety of different flow description styles and the resulting flexibility requirement for storing internal state information, the protocol engine becomes relatively complex, particularly in terms of state administration. This topic is further discussed in the next section and in Section IV.B.

B. Particular Design Aspects

RSVP offers a high degree of robustness and flexibility and efficiently supports a variety of group communication patterns. The complexity of any RSVP implementation is largely determined by RSVP's main design goals, but it is also influenced by other decisions, such as the layout of internal state representation. In this section, design principles of RSVP (originally presented in [5]) are reviewed with respect to their impact on execution complexity and performance. The findings are justified later in the article.

Per-Flow Signalling. RSVP requires the exchange of signalling messages between participating nodes per flow. This in turn results in communication overhead that is at least linear in the number of signalling requests. In principle, a flow could also denote the aggregation of multiple application flows, but the basic RSVP flow addressing scheme effectively prohibits such aggregation. There has been work on including CIDR addressing into RSVP [6], but this proposal has not been incorporated into the standard specification. Recent work on protocol extensions for tunnelling [7], request aggregation [8] and MPLS signalling [9] alleviate this restriction, albeit by assuming additional packet encapsulation and/or packet marking.

Per-Flow State. In RSVP, per-flow state information is stored at each participating node. This results in storage overhead that is linear in the number of flows. Despite the need for randomly accessible per-flow state, an appropriate design of an implementation's internal data structures (see Section IV.B for details) can keep the resulting computation overhead almost constant.

Soft State. The concept soft state in RSVP mandates periodic but asynchronous refreshing of state information, such that for any flow between any pair of adjacent nodes, an individual refresh period can be used. The periodic refresh of state information increases the communication overhead by a fixed factor, but does not change its complexity. The administration of timeouts and asynchronous refreshes requires the use of at least two timers for each unit of state information. This in turn results in a timer container, the size of which is linear in the number of flows. Again, with an appropriate layout of the timer container, the resulting overhead can be kept almost constant.

Simplex Flows. RSVP is designed to signal for simplex transmission flows, because one-way multicast communication was deemed an important application scenario at the time. However, even for bidirectional communication scenarios, a network layer signalling protocol must internally use the notion of simplex flows as basic building blocks to gracefully deal with asymmetric unicast routing and multicast routing not using shared trees. Because of this design, establishing QoS for a bidirectional communication path requires at most twice as much communication and state overhead as a duplex signalling protocol (which would require symmetric routing). Details depend on the QoS model and its representation and whether the requested service characteristics are symmetric.

Message Exchange. With RSVP, the actual QoS request is initiated by the receiver of a flow. The communication pattern is termed "one-pass with advertising", but essentially represents a two-pass signalling model. PATH messages establish the path information at intermediate nodes from each sender to each receiver and provide a fallback mechanism to react to routing changes. This information is later used to transmit RESV messages from the receivers back towards the senders along the same paths, subject to request merging. This particular design was chosen for a number of reasons. First, receiver-initiated QoS requests are necessary to support receiver-oriented multicast and request merging. Second, storage of path information enables transparent traversal of non-RSVP nodes along the network path. Finally, storage of sender-specific path and reservation information makes RSVP messages idempotent, which is an important and valuable robustness property. This communication pattern imposes increased communication and computation overhead when compared to a sender-initiated protocol, since each node has to process two messages to establish QoS for a simplex flow. Further, supporting sender-specific state information prohibits certain types of aggregation, as for example proposed for BGRP [10]. On the other hand, global consistency for proposals such as BGRP requires reliable communication, which may be hard to guarantee in the presence of node failures, or reduced resource utilization to accommodate temporary failures.

Multicast. RSVP's support for multicast also requires maintaining separate sender and receiver state, at least for multicast sessions. This case is detectable by inspecting the session address when receiving the first PATH message for a new session. While it seems possible to use separate processing paths and optimize the protocol engine for the unicast case, this would certainly increase the overall complexity of a protocol implementation.

Multiple Senders. Support for multiple senders requires maintaining separate sender and receiver state, at least for such sessions with multiple senders. It might be possible to optimize an implementation for the single-sender case, but besides the resulting increase of the overall complexity of a protocol implementation, this would likely need support in the protocol specification. In contrast to multicast sessions, the eventual existence of multiple senders is not detectable when state is created for a new session, so an appropriate protocol element would be necessary to distinguish the single-sender from the multiple-sender case.

Filter Styles. RSVP allows for the specification of session senders in multiple types of combinations. These so-called filter styles increase the implementation complexity, because multi-sender specifications must be used as keys to index certain types of state elements. Multiple filter styles prohibit the design of a single, simple data structure for this purpose, which in turn increases the complexity of matching state entries. As before, an optimization for certain subsets of the full functionality might be possible at the cost of increased overall complexity.

Flexible Protocol Elements. There is a great amount of flexibility in RSVP to compose protocol messages from individual protocol elements. Further, protocol elements for the actual QoS specification are separated from signalling elements. All protocol elements are specified in a binary format. The possible variations on the order and existence of objects increase the complexity of internal message representation and message parsing, compared to a more simplistic protocol.

Extension: Refresh Reduction. An extension to RSVP can be used to reduce the overhead associated with state

TABLE 1: SOFTWARE PACKAGE OVERVIEW

Software Component	Lines of Code
Protocol Elements (Generic)	4,300
Common Processing Services (Generic)	3,300
Common Services (Platform-Specific)	800
Protocol Daemon (Generic)	7,900
Protocol Daemon (Platform-Specific)	2,400
Interaction with Packet Schedulers	4,800
Interaction with MPLS Forwarding	500
Client API	700
Java Client API	500
Traffic Generator	4,300
Other Client Programs	3,100
Core RSVP ns-2 Extensions (C++,Tcl)	2,000
Other ns-2 Extensions	1,000
Total	35,600

refresh in RSVP [11]. This extension reduces the communication overhead of refresh signalling by assigning integer identifiers to state information and bundling multiple refresh notifications into a single protocol message. The effect of this extension is somewhat limited, because there is a maximum number of state refreshes per refresh message to avoid packet fragmentation (which would nullify the intended performance benefits). Further, certain operations still have to be performed for each flow for each refresh cycle and these operations often dominate the execution cost and keep the overall execution overhead linear in the number of signalling requests per time.

IV. SOFTWARE DESIGN AND IMPLEMENTATION

The major design goals of the protocol implementation presented here are clarity of design and code, as well as the potential to investigate protocol extensions and internal optimization. The resulting code might achieve these goals only to a limited extent, but some lessons have been learned and are reported below. These findings are likely to be useful for the efficient implementation of other signalling protocols, as well. The software is available at:

http://www.kom.e-technik.tu-darmstadt.de/rsvp
http://www.uwaterloo.ca/~mkarsten/rsvp

A. Software Overview

To give a very rough idea of the complexity of the software, all parts of the package are listed in Table 1 together with their approximate size in lines of code. In Table 1, the terms *Protocol Elements* and *Common Services* refer to code that is used for both the client API and the protocol daemon.

B. State Representation

The state representation in this implementation is based on the design suggested in RFC 2209 [12], but is significantly modified and broken down into a more fine-grained layout. All state is stored as objects containing relationships to other objects. The main entry point into the state representation is given by *Session* objects, which are stored in a global, hash-based container. Starting from a Session object, the full state for each session can be traversed to access specific state blocks. Most information from a PATH message is stored in a *Path State Block* (PSB) whereas request-specific 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 about an actual resource reservation at an outgoing interface is stored in an *Outgoing Interface State Block* (OutISB). The relationship between such reservations and PSBs would be an N:M relationship, because of RSVP's support for multiple senders and multiple receivers. This N:M relationship is split into two 1:N relationships by introducing an entity *Outgoing Interface at PSB* (OIatPSB). This class allows to precisely consolidate the context for interactions of the main protocol engine with the traffic control module, for example, to enable automatic cleanup of packet scheduler state. Fig. 2 shows the entity-relationship diagram for the design of RSVP state information. Table 2 shows the most important attributes stored in different entities and illustrates the more fine-grained distribution of state information across entities compared to the basic design in RFC 2209 [12]. Modelling state by an entity-relationship model is deemed useful for understanding and documenting the design of an implementation using object-relationships [13].

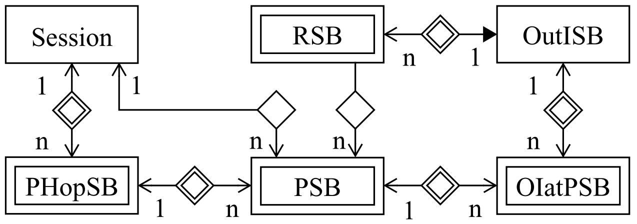

Figure 2: Entity-Relationship Diagram for State Blocks.

C. Fuzzy Timer Management

As discussed in Section III.B, two timers need to be maintained for each path and reservation state block. Consequently, timer handling requires the single largest sorted container in an RSVP engine. The data structure in this implementation is based on a *timer wheel* [14].

TABLE 2: STATE BLOCKS AND ATTRIBUTES

State Block	Attributes
Session	session (IP address, port, protocol)
	filter style
PSB	sender (IP address, port)
	traffic information (TSpec from PATH)
	path information (AdSpec from PATH)
	aggregated upstream reservation to sender
	timeout timer for PATH from previous hop
Refresh Red:	message ID of PATH from previous hop
Refresh Red:	message ID for PATH to next hop
MPLS:	label from previous hop
PHopSB	previous hop (IP address)
	incoming interface
	aggregated/combined upstream reservation
	refresh timer for RESV to previous hop
Refresh Red:	message ID for RESV to previous hop
RSB	next hop (IP address)
	downstream reservation (Flowspec from RESV)
Refresh Red:	message ID of RESV from next hop
OutISB	outgoing interface
	installed reservation
OIatPSB	scheduling filter
	refresh timer for PATH to next hop
	timeout timer for RESV from next hop
MPLS:	label to next hop

The basic timer wheel works as follows. All timers of the protocol engine are stored in a hierarchical container. The upper layer is implemented as an array representing time slots and accessed through a hash-like division function preserving the order of timers. Timers within individual time slots are kept in simple sorted lists. The amount of time covered by each slot is configurable at runtime. The container is only capable of foreseeing a limited amount of time in the future, which is sufficient for RSVP. Its best possible update complexity is *O(log(n))*, with *n* being the (varying) number of timers in a slot. Consequently, performance can be traded off against memory usage by choosing the size and number of slots. This data structure is illustrated in Fig. 3. Since the per-slot containers are fully sorted, timers can be fired precisely according to their individual expiration time.

The basic timer wheel can be significantly improved for RSVP. Since RSVP uses randomized refresh timers, timer precision only needs to be at the scale of tens of milliseconds. If the duration covered by a single time slot is below this time-scale, *fuzzy timers* can be used. While in the basic timer wheel, the per-slot containers are sorted, fuzzy timers are instead stored within each time slot in an unsorted list. When a time slot becomes eligible for expiration, its timers are fired in order of their position in the unsorted list, rather than the precise expiration time. The scheme results in a slight inaccuracy of timers, which is bounded by the length of a single time slot. This can be considered a very reasonable trade-off, particularly in case of RSVP, because it reduces the update complexity to *O(1)* and requires changing only about 20 lines of code. The performance gains of this optimization are studied in detail in Section V.C.

In both cases, the cost for searching the next element in the timer wheel is *O(n)* in the worst case, with *n* being the number of time slots. It is possible to reduce this worst-case bound, for example by using (hierarchical) bitmaps. However, in the context of RSVP signalling, it is reasonable to expect that most timer slots are loaded at any time, since the system needs to periodically fire refresh timers. Therefore, the actual overall cost of the timer data structure is determined by its update complexity.

D. Dedicated Memory Management

As with any other signalling protocol used for this purpose, an RSVP daemon frequently needs to dynamically allocate and de-allocate memory for objects. Using regular heap memory management incurs a significant processing overhead (see Section V.D for details) due the limitations of general purpose heap memory management algorithms. Therefore, this implementation optionally employs a very simple memory management system, which is implemented transparently for the rest of the code by overloading the `new` and `delete` operators of C++ and requires only about 200 lines of additional code. It turns out that such a simple optimization is sufficient to recover a significant portion of the execution overhead of the standard heap memory management.

A dedicated memory management subsystem is created for different types of fixed-size state objects and other frequently allocated data structures, such as basic list nodes.

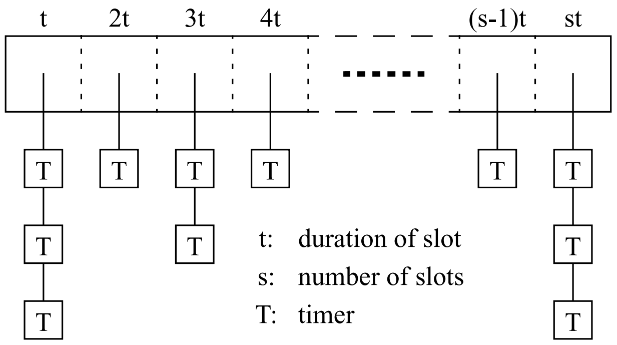

Figure 3: Design of Timer Container.

When an object is destroyed, its memory is inserted into a FIFO free list associated with the type-specific memory management subsystem. Allocation requests are first served from the respective free list, if possible, or otherwise by allocating memory from the heap. Further, the first few bytes of each object's associated memory are used to store the pointers necessary to keep the memory chunks in the free list. As a result, for each data structure larger than a few bytes, dedicated memory management does not increase the memory usage. This is true for all relevant data structures in the protocol engine.

Essentially, this scheme results in an ever-growing allocation of heap memory corresponding to the maximum number of flows that are stored in the protocol engine. Since no memory is wasted, this is not a problem, because often a network node will be designed to sustain a certain maximum number of flows. If there are memory-intensive tasks that are supposed to be executed by a network node in times of low signalling load, an asynchronous memory cleanup procedure can be executed to recover memory. The performance gains of this optimization are studied in detail in Section V.C and Section V.D.

E. *Standardized Protocol Extensions*

The IETF and other institutions have produced standardization proposals for using and extending RSVP. Two of these proposals are partially implemented in our prototype to study their effects in terms of implementation complexity and signalling performance.

1) Refresh Reduction

Due to the soft-state in RSVP, all protocol messages are idempotent and the same message is used as trigger or refresh message. Processing refresh messages is relatively expensive and thus, depending on the length of the refresh interval, refresh messages contribute significantly to the overall processing effort (see Section V.D. for details). This situation has been addressed by research proposals like [15,16] to reduce refresh overhead and led to an IETF standardization proposal [11], which specifies a mechanism for reducing refresh overhead by using so-called summary refresh messages. This mechanism has been implemented for unicast communication to investigate its effect on the overall performance capabilities of the protocol engine. Details of the mechanism are not presented here, but instead the reader is referred to the appropriate research and standardization literature cited above. Note that refresh reduction for multicast communication is not implemented in our prototype. To implement this mechanism, the multicast routing interface is required to supply a list of next hops upon a routing lookup, which is not supported by the RSRR interface [17] used in this implementation. The superiority of RSVP for multicast communication is largely undebated and therefore, is not considered a crucial aspect for this investigation. Message bundling as specified in RFC 2961 [11] is not implemented, but it is possible to assess its performance effects based on available data (see Section V.D).

The implemented mechanism is based on assigning unique identifier values to trigger messages and subsequently refreshing state by transmitting multiple identifiers within a single summary refresh message [11]. No particular sequencing is applied to messages, other than the requirement that trigger messages must be assigned a strictly increasing identifier by the sending hop in order to distinguish new trigger messages from older ones that may have been delayed in the network. To exploit the potential efficiency gain, these identifiers can be used by the downstream node to access incoming state information through hash-based containers, using the message identifier as an access key. For outgoing state information, the available range of identifiers must be administered in a similar way. There is consequently a choice of operating the protocol engine with one global container or a confined container per adjacent hop. According to RFC 2961 [11], there is no requirement for globally unique identifiers, but instead identifiers only need to be unique for the adjacent node. Therefore, the implementation employs unique and increasing identifiers per adjacent RSVP hop. Such a scheme has the advantage that for outgoing state, it is easy to build an almost perfect hash using a limited number of hash buckets, whereas a global container would tend to spread identifiers not as uniformly over the hash function. This is due to the fact that summary refresh messages are targeted to a specific hop and contain refresh state previously announced to this common hop. Initial tests have shown that it is beneficial to evenly distribute the generation and sending of summary refresh across the refresh interval and to keep their size below the link-layer MTU. Otherwise, packet fragmentation largely nullifies the performance benefits of the summary refresh mechanism. The extension requires about 850 lines of code.

2) MPLS Signalling

Support for MPLS label distribution and explicit routing as specified in RFC 3209 [9] is implemented as proof of concept that the existing software structure allows for rapid and easy integration of such extensions. The signalling part is capable of interfacing with two publicly available MPLS data forwarding packages for Linux [18,19]. The overall effort for the protocol extension is very limited and the majority of additional code handles the low-level interaction with the MPLS forwarding mechanisms. This finding is backed up by the fact that the extension requires only about 1000 lines of code, most of which is concerned with interfacing to the kernel-level forwarding software.

F. Network Simulation

The prototype software has been ported to the ns-2 simulation environment [20]. The successful port, which in total required less than 4000 lines of code, including a significant piece of code for reactive admission control (see Section VI.C) serves as a proof of concept that the core RSVP implementation is indeed highly portable. The ns-2 environment is capable of running a realistic version of RSVP and there is only one slight inconsistency between ns-2 and real-world packet forwarding. In ns-2, packet routing is based on each node's identifier, rather than the address of a network interface. At certain points in the RSVP code, this requires an appropriate translation between node and interface identifiers.

Another goal of this effort is to use concurrent simulation and lab experiments to investigate the behaviour of the protocol and its extensions. In this case, the simulated technology can be compared to and calibrated by results from real-world experiments. This in turn should help to validate subsequent results from large-scale simulations. The port is considered a proof of concept for this experimental methodology, although there clearly is a trade-off between the resulting complexity of very realistic simulation code and the possible benefits of such integrated software development.

V. PERFORMANCE AND EXECUTION COST

The results of detailed performance experiments allow the investigation of the execution performance of a basic RSVP implementation, as well as the effects of extensions and optimization described in the previous section. While the detailed numbers are valid only for one specific implementation and one particular platform, it is assumed that in combination with publicly available software, they are useful beyond those particularities. The study focuses on

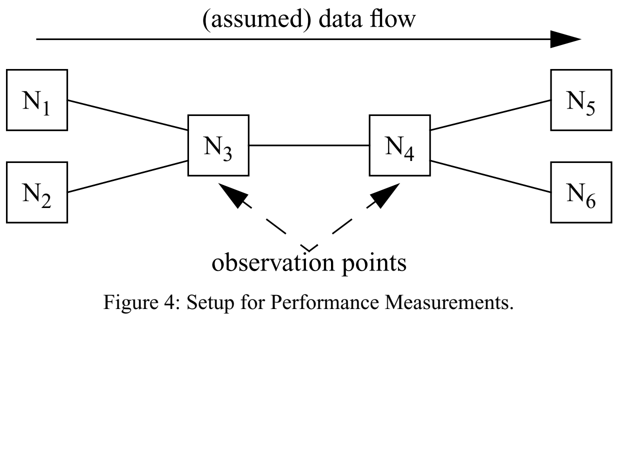

Figure 4: Setup for Performance Measurements.

unicast communication, since the superiority of RSVP for multicast communication is largely undebated. The crucial aspect for assessing RSVP's suitability as QoS signalling protocol is currently rather given by its behaviour for small unicast flows, for example for Voice over IP (VoIP).

The first set of experiments is aimed at investigating the maximum sustainable signalling load, depending on the different implementation variants. The interpretation of "sustainable" in this context is that no more than 0.01% of signalling requests are lost at any point in time. The maximum sustainable load then allows calculating an estimate for the average processing time associated with each session. Note that for a user-level daemon, it does not make sense to attempt to obtain any more fine-grained information, since the system jiffy (granularity of system timer interrupts) and buffering in the network stack results in slight variations of the processing delay. In the second step, the total execution overhead is decomposed to investigate the contribution of various processing blocks, to further illustrate the effects of various implementation options, and to estimate processing latency.

A. Experiment Setup

The experiments have been carried out on standard PCs running FreeBSD 4.5 on 450 MHz PentiumIII processors with 256 MB main memory. Six nodes are connected by full-duplex FastEthernet links as depicted in Fig. 4. Nodes N_5 and N_6 are used as destinations and N_1 and N_2 as sources. Multiple unicast sessions are created by specifying multiple port numbers. The average refresh interval is set to 30 seconds, as suggested in RFC 2205 [2]. The protocol daemons exchange basic RSVP messages without policy data or integrity objects and no confirmation messages. Sessions are always torn down by senders, so there are no reservation tear messages either. The length of timer slots in the timer wheel is auto-configured by the RSVP daemon, depending on the measured system jiffy (time period between system timer interrupts), which is approximately 20 milliseconds on a vanilla FreeBSD system. The time horizon is set to 600 seconds, which is sufficient for all timers in these experiments. Consequently, the timer wheel consists of 30,000 slots. The hash-based session container is restricted to 8192 buckets to limit the positive effects of the perfect hash that results from the session generation in this lab environment. The receivers at N_5 and N_6 react to each path advertisement by immediately generating reservation requests, which establish the end-to-end reservation. No data packets are transmitted.

The signalling load generators at N_1 and N_2 create sessions with a uniformly distributed inter-arrival time and constant duration of 240 seconds. A simple and deterministic session arrival model is used, since the main goal of these experiments is to investigate the average processing overhead of RSVP daemons under high signalling load. A deterministic configuration resulting in a stable signalling load is necessary to relate average profiling results to individual message processing times. In contrast, a realistic traffic arrival and duration pattern would make the interpretation of results much harder. In reality, a burst of arriving signalling messages is buffered in the network stack of an RSVP system. Given the average message processing time, realistic arrival patterns can then be analysed using standard queuing theory.

The observations are made at Node N_3 and N_4. Measurements are done by periodically executing `top` (every 3 seconds) and recording the highest numbers of total memory consumption and percentage of raw CPU usage that is reported for execution of the RSVP daemon on either node. The per-flow memory allocation is measured by substracting the memory consumption when no flow is present from the total memory allocation and then dividing the result through the number of flows. Although this kind of measurement introduces some inaccuracies and inherent randomness, those effects are limited by choosing deterministic experiment configurations. Further, the consistency of the results shows that the principle effects are not obstructed. Note that `top` already averages CPU usage, so it is in line with the overall goal of measuring the average processing effort, and it is quite accurate with respect to memory consumption.

These experiments measure raw signalling processing performance, without taking into account actual traffic or traffic control. Further complexity in the experiment setup would probably only obscure the results. In reality, signalling messages are likely transmitted with a special priority setting and therefore, will be unaffected by high traffic load. It does not seem useful to incorporate the interactions with UNIX-based software packet schedulers into these experiments. Special network-level conditions, such as routing instabilities, occur on a different system scale and as such, warrant a dedicated investigation.

B. Relation to Earlier Work

A subset of similar performance figures is reported in our earlier work [21]. These results were obtained by running a previous version of the protocol implementation on FreeBSD 3.4 using the same hardware. Experiments have shown that running the older version of the software on FreeBSD 4.5 incurs an increase in execution cost of about 10 % compared to FreeBSD 3.4. In both cases the same compiler version is used, so the difference is likely caused by operating system internals and have not been explored in the context of this work. On the other hand, general coding-level improvements of the protocol engine achieve an equivalent reduction in execution cost, such that the fundamental results are almost directly comparable.

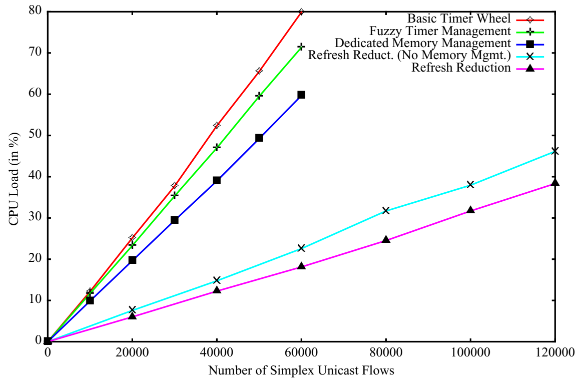

Figure 5: Execution Overhead of RSVP Signalling

C. Signalling Performance

The results of the signalling performance experiments are shown graphically in Fig. 5. The figure shows the relative CPU load depending on the number of sessions signalled. Different software configurations are investigated in an incremental fashion to identify the influence of each option on the overall execution overhead. Each of the configurations is discussed in the subsections below in the order of labels in Fig. 5 from top to bottom. The respective memory usage per flow is shown in Table 3.

Since the session duration is 240 seconds, 120,000 sessions result in 500 session setups and teardowns, or 1000 transactions, per second. The most optimized version then incurs a CPU load of 40% and is essentially constrained by main memory during these experiments. Since normally 80% user-level CPU load can be sustained without losing signalling requests, this amounts to a maximum processing performance of 2,000 transactions per second.

Earlier work [21] has shown that the execution overhead of the ISI reference implementation [22] is clearly superlinear. In raw numbers, the ISI protocol daemon cannot only sustain a maximum signalling load of 5,500 sessions using the same experimental configuration on the same platform, mainly because it does not offer the same level of flexibility in terms of tuning data structures. However, besides using suitable container structures, it has also been found that the design presented in Section IV creates significant performance improvements [21].

1) Basic Timer Wheel

The leftmost curve in Fig. 5 shows the system performance using the basic timer wheel. The execution overhead is almost linear in the number of sessions, but increases to be slightly superlinear when a large number of sessions is

TABLE 3: MEMORY USAGE PER SIMPLEX UNICAST FLOW STATE

Configuration	Bytes per Flow
Basic Timer Wheel	1388
Fuzzy Timer Management	1388
Dedicated Memory Management	1388
Refresh Reduction (No Memory Mgmt.)	1546
Refresh Reduction	1546

signalled. This is the expected behaviour as the access complexity for each time slot is negligible for a small number of sessions and thus timers per time slot. With an increasing number of timers per time slot, the intensified effort of administrating each sorted container results in a visible increase of the overall execution overhead.

2) Fuzzy Timer Management

The second curve from the left in Fig. 5 shows the system behaviour with fuzzy timer management being enabled. The overall processing effort is reduced and remains truly linear in the number of sessions, even at a high load. The corresponding performance gain ranges from 4% under low load to almost 11% under high load. The same amount of memory is used in both cases (see Table 3).

3) Dedicated Memory Management

As illustrated by the third curve from the left in Fig. 5, employing dedicated memory management reduces the execution overhead by 16-18%, depending on the number of sessions, without affecting the total memory usage (see Table 3). For these experiments, almost all data structures that are dynamically allocated on the heap are configured to use dedicated memory management.

4) Refresh Reduction

The two curves at the right of Fig. 5 illustrate that enabling the refresh reduction mechanism reduces the execution overhead on average by 69% when compared to the basic operation mode including fuzzy timer management. Table 3 reveals an increase in memory usage by slightly more than 11%. Nevertheless, the execution cost remains linear in the number of sessions, instead of decreasing to sublinear. This observation is explained in the light of results from the profiling experiments in the next section. Using dedicated memory management results in a further reduction of execution overhead by approximately 18%.

D. Decomposition of Execution Overhead

To understand the contribution of individual processing steps on the overall execution overhead, profiling is used to decompose the overall processing effort. The same experiments as in the previous section are carried out with a session load of 20,000 sessions. A complete breakdown of the overall execution effort is shown in Table 4. The relative cost of system services is shown in Table 5.

Three versions of the software, which all include fuzzy timer management, are studied. The label *Basic Version* refers to the basic protocol version. This version is compared to the implementation including dedicated memory management, labelled as *Memory Mgmt*. Additionally, the basic version is compared to the refresh reduction version without dedicated memory management which is shown as *Refresh Reduction* in the tables. Comparing these three version suffices to draw the most interesting conclusions, even about other configurations.

The execution effort for the versions including dedicated memory management and refresh reduction, respectively, is shown in relative numbers. To be able to directly compare the absolute execution effort, the relative numbers are also shown normalized to the execution effort for the basic version (in columns labelled as *norm*). The normalization factor is taken from the total overhead measured in the experiments presented in Section C for 20,000 sessions and can be found in row *Total* of Table 4.

Since no confirmation is requested and flows are always torn down by senders, neither RCONF nor RTEAR messages are exchanged. Compared to our earlier work [21], the profiling methodology has been improved to increase accuracy and clarity, particularly by eliminating the daemon initialization time from the measurements. Nevertheless, the numbers are naturally not fully precise, but rather a good indication of the relative processing overhead.

1) Processing Steps

The processing step *Message Reception* consists mainly of interaction with the operating system to receive packets that contain RSVP messages. In the basic version, it consumes a large fraction of the overall execution effort, due to the high cost of system calls (discussed below). As discussed in Section III.B, RSVP allows messages to be flexibly composed of individual protocol elements. Therefore, *Message Parsing* requires significant effort. In this implementation no specific measures have been implemented to optimize message parsing. During *Basic Processing*, a number of validity checks, as well as certain lookup operations are performed, for example to locate the sending hop. It only incurs a small fraction of the overall processing cost, similar to the *Session Lookup* step. All the above processing steps are executed for each incoming message.

The execution effort for *PATH Processing* contains the state creation or maintenance for all incoming PATH messages. A trigger PATH message synchronously prompts the creation of an outgoing PATH message during *Trigger*

TABLE 4: DECOMPOSITION OF OVERALL PROCESSING EFFORT

Operation	*Relative* Execution Effort in %				
	Basic Version	Memory Mgmt.		Refresh Reduction	
			norm		norm
Message Reception	19.9	21.1	17.8	12.3	4.0
Message Parsing	11.4	10.6	8.9	7.8	2.5
Basic Message Processing	2.5	3.0	2.5	1.8	0.6
Session Lookup	2.1	2.3	1.9	1.2	0.4
PATH Processing	20.4	21.8	18.4	7.8	2.5
Trigger PATH Refresh	1.9	1.7	1.4	5.7	1.8
RESV Processing	10.2	8.6	7.2	5.3	1.7
Trigger RESV Refresh	1.9	2.1	1.8	5.4	1.8
PTEAR Processing	2.4	2.1	1.8	10.3	3.4
SRefresh Processing	n/a	n/a	n/a	31.6	10.3
Reservation Merging	2.5	1.9	1.6	5.7	1.8
Timer Firing	5.9	5.9	5.0	2.0	0.7
Timer Execution (PATH)	12.6	11.7	9.9	n/a	n/a
Timer Execution (RESV)	6.3	7.2	6.1	n/a	n/a
Timer Execution (Refresh)	n/a	n/a	n/a	3.1	1.0
Total	100	100	84.3	100	32.5

PATH Refresh. Similarly, *RESV Processing* covers the creation or maintenance of reservation state for all RESV messages, while *Trigger RESV Refresh* encompasses the additional overhead associated with trigger messages. During *PTEAR Processing*, the protocol daemon destroys existing state information, including the associated reservation state. The configuration of the session lifetime and refresh period (240 and 30 seconds, respectively) results in an average of 8.5 PATH and RESV messages[†] that are exchanged during each session with only the first of each actually creating internal state. In contrast, only a single PTEAR message is transmitted at the end of each session. When refresh reduction is used, only a single PATH and RESV message are transmitted for each session and state refresh processing is contained in *SRefresh Processing*.

The row labelled *Reservation Merging* shows the overhead incurred by merging reservations that are requested from the previous hop. It is executed during the processing of the first RESV message per session and reception of a PTEAR message. *Timer Firing* covers the raw cost of timer maintenance, while *Timer Execution* denotes the corresponding actions of sending PATH and RESV refresh messages to adjacent nodes.

[†]. Since the actual refresh period is determined randomly, there is a probability of 50% that the refresh message for the last refresh period arrives before the PTEAR message for the session.

TABLE 5: PROCESSING EFFORT FOR SYSTEM SERVICES / SYSTEM CALLS

Operation	*Relative* Execution Effort in %				
	Basic Version	Memory Mgmt.		Refresh Reduction	
	basis		norm		norm
Routing for PATH	11.7	13.4	11.3	3.1	1.0
Routing for SRefresh	n/a	n/a	n/a	21.6	7.0
Multiplexing (select)	6.7	7.4	6.2	5.9	1.9
Receiving (recvmsg)	9.5	10.7	9.0	4.7	1.5
Sending (sendto)	14.2	16.2	13.7	8.0	2.6
Get Time (gettimeofday)	0.7	0.8	0.7	1.4	0.5
Memory Management	22.5	3.6	3.0	26.1	8.5
Total	65.3	52.1	43.9	70.8	23.0

2) Cost of System Services

The relative execution times shown in Table 4 indicate high cost of system services. This conjecture is confirmed by the breakdown in Table 5, which shows the relative execution overhead for requests that interact with operating system services. If a single system call is used for the request, it is listed in the table. A routing lookup requires two system calls to write and read a routing socket. Memory management contains all requests for heap memory.

In the basic version, all tasks related to sending and receiving packets (*Multiplexing*, *Receiving*, and *Sending*) are responsible for more than 30% of the overall processing time. Additionally, there is a significant cost for routing lookups (*Routing for PATH*). Without optimization, *Memory Management* is responsible for more than 22% of the total processing overhead.

3) Processing Cost for Individual Messages

Based on the experimental data, it is possible to obtain reasonable estimates for the processing cost and latency of individual messages in different configurations. The ratio of message types is determined by the session lifetime and refresh interval. As an example, with the given configuration in the basic protocol version, every session on average results in 18 messages (8.5 PATH, 8.5 RESV, and 1 PTEAR) being transmitted. Different messages trigger different processing steps of those listed in Table 4. For example, *Message Reception* is executed for each message, *PATH Processing* is only invoked for PATH messages, while *Trigger PATH Refresh* is only necessary for the first PATH message. Table 6 shows which processing steps correspond to which message types.

An estimation for the relative cost for each message can be calculated as the sum of per-message costs for each processing step. The per-message cost of each processing

TABLE 6: PROCESSING STEPS AND MESSAGE TYPES

Operation	PATH		RESV		PTEAR
	trig.	refr.	trig.	refr.	
Message Reception	1	7.5	1	7.5	1
Message Parsing	1	7.5	1	7.5	1
Basic Message Processing	1	7.5	1	7.5	1
Session Lookup	1	7.5	1	7.5	1
PATH Processing	1	7.5			
Trigger PATH Refresh	1				
RESV Processing			1	7.5	
Trigger RESV Refresh			1		
PTEAR Processing					1
Reservation Merging			1	7.5	1
Timer Firing		7.5		7.5	
Timer Execution (PATH)		7.5			
Timer Execution (RESV)				7.5	

step is obtained by dividing its total relative cost through the number of messages for which it is executed.

For example, basic PATH message processing incurs a relative cost of 4.4%. Trigger PATH messages are responsible for another 1.9% for the synchronous refresh, while the regular generation of PATH refresh messages accounts for 2% relative overhead. It can be concluded that refresh and trigger messages require almost the same processing effort. This finding is reasonable when considering that generating refresh message includes firing timers and further, the total cost is dominated by system calls that are required for both. Carrying out the same calculation for RESV messages results in 5.4% total cost for RESV trigger messages and 4.7% for refresh messages including timer execution. Again, the cost of refresh messages is close to the cost of trigger messages. There is more overhead associated with processing a RESV trigger message, because of dynamic memory allocation and reservation merging. The maximum load of 60,000 active sessions for the basic version results in 4,500 messages/sec, so by combining the ratio of 8.5:8.5:1 with the relative cost figures calculated above, the processing latency for each message can be estimated.

4) Effects of Dedicated Memory Management

With dedicated memory management, the absolute execution cost for 20,000 sessions is reduced by almost 16% (see Section C). Due to its simplicity, dedicated memory management is fully implemented using inline functions. Because the software is compiled with compiler optimization, the cost reduction shown in Table 5 appears slightly too high, since profiling accounts some of the cost to the respective caller functions while Table 5 only shows the actual cost of system heap memory management. Comparing the relative and the normalized relative execution cost in Table 4 and Table 5 reveals that all processing components benefit from this improvement, but to a different extent. For example, the execution cost of all other system-related requests is dominated by the underlying system calls and thus, is not improved much by dedicated memory management. Consequently, the relative execution overhead is increased, while the normalized relative execution overhead is decreased. Those processing steps that exhibit a reduced relative execution overhead, for example *Message Parsing* and *RESV Processing*, benefit most from this optimization.

5) Effects of Refresh Reduction

The primary benefit of the refresh reduction extension is the reduction in the number of messages that are being exchanged between adjacent RSVP nodes. On the other hand, some additional functionality is necessary to maintain message identifiers. This cost is best illustrated by the increase in relative and absolute processing cost for *PTEAR Processing*, since there is still a single PTEAR message transmitted at the end of each session. Further, the creation and processing of summary refresh (SRefresh) messages needs to be considered.

In case of refresh reduction, overall message processing is dominated by processing of summary refresh messages (see Table 4). This overhead in turn is largely caused by expensive routing lookups (see Table 5), which are also necessary for refreshed PATH state to detect routing changes. The rest of summary refresh processing also uses significant processing time. This is no surprise, given the information density of a summary refresh message.

When using refresh reduction, individual state refresh timers are replaced by summary refresh timers. Therefore, the relative processing effort of timer firing and timer execution is reduced and the reduction of overhead is significantly higher than for the rest of the message processing. By comparing the relative and the normalized relative execution effort, it is possible to identify those processing steps that benefit the most from this protocol extension.

6) Message Bundling

Taking into account the available information, it is possible to estimate the effects of other protocol extensions, such as message bundling [11]. Bundled messages require a small overhead in terms of message creation and parsing, but reduce the overhead associated with sending and receiving messages. Depending on the details of signalling and the link-layer MTU, a bundled message may contain up to a certain number of regular RSVP messages. According to Table 5, the execution overhead of sending and receiving messages accounts for approximately 30% of the overall execution overhead in the basic protocol version. If, for example, 10 messages can be replaced by a single bundled message, the overall execution overhead may be reduced by up to 27%. In case of refresh reduction, however, the relative execution overhead of network I/O is lower (20%) and message bundling can only be applied to trigger messages, since summary refresh messages already occupy the full link-layer MTU. Thus, the relative improvement would be much smaller.

E. Communication Overhead

The relative communication overhead of RSVP signalling on the wire largely depends on the particular configuration in terms of refresh interval and refresh reduction. Also, the transmission rates of those flows for which QoS is requested determine the relative communication overhead. A worst-case estimation can be done as follows.

A PATH message including a basic ADSPEC object uses 132 bytes while the size of a RESV message requesting Guaranteed service for a single sender using fixed filter style is 128 bytes. Assuming an estimated overhead of 340 bytes for additional ADSPEC fields, POLICY and INTEGRITY objects, 600 bytes are exchanged for RSVP signalling within one refresh period. Further assuming an average refresh period of 30 seconds and a VoIP-like traffic flow with 64 kbit/sec transmission rate, the data traffic within one refresh period adds up to 240,000 bytes. Then, the ratio of signalling traffic to data traffic in this scenario is 1:400. Note that this ratio is much smaller for larger data flows, for example video streams, and also is significantly reduced when enabling refresh reduction.

F. Summary

The experiments reported here confirm the observations discussed in Section III.B. The overall execution cost of an RSVP daemon can be kept linear in the number of transactions per time. In a user-level implementation, the processing overhead is dominated by interaction with operating system services, such as packet transmission and routing lookups. However, it is possible to significantly improve the performance through simple internal optimization like fuzzy timers and dedicated memory management as well as protocol extensions like refresh reduction. On the other hand, the impact of operating system interaction is so high that even when using refresh reduction, the linear complexity of RSVP is unchanged.

VI. Experimental Protocol Extensions

A number of protocol extensions have been evaluated to assess the extendability of RSVP's scope of operation. They are briefly presented here to support the claim of significant implementation experience with the prototype.

A. One-pass Signalling

As discussed in Section III.B, RSVP uses a two-way message exchange to set up an end-to-end simplex reservation. There are a number of scenarios in which the features of this two-way procedure and simplex reservations are not needed and only result in an unnecessary overhead for both end systems and intermediate nodes. A true one-pass and duplex-capable service establishment mechanism has been designed. It fully interacts with traditional RSVP signalling, such that it is possible to optionally override an initial one-pass reservation with later requests.

A new message type, PATHRESV indicates that reservation(s) based on the transmitted *TSpec* shall be established in one pass from sender to receiver. Other than the message type, the message is exactly the same as a PATH message. A DUPLEX object carrying reverse port information can optionally be added to a PATHRESV message to request a duplex reservation, assuming that the same transport protocol is used in both directions. The duplex extension is only useful, if symmetric routing is employed between the end systems and only for unicast communication. One benefit of this extension is reduced complexity for end systems through a one-pass service interface without any activity by the responder. Also, the communication overhead at all nodes is reduced, because less messages are exchanged. As a result, this mechanism enables lightweight signalling in the framework of RSVP.

The one-pass signalling extension has undergone a performance evaluation similar to the experiments reported in Section V.C (see [23] for details). There is a significant performance improvement of about 40% compared to the basic version, if only one-pass signalling is used. This is no surprise given that only half the number of messages is exchanged and, as presented in Section V.D, the interaction with the system's network services and message parsing use a large fraction of processing resources.

B. Remote Clients

RSVP defines two alternative methods to transmit messages between RSVP-capable nodes: as raw IP packets or using UDP encapsulation [2]. In both cases, multiple clients on a single end system require a central entity (usually the RSVP daemon) to receive and dispatch incoming messages. The remote client extension allows the first RSVP-capable hop to directly communicate with thin clients. Clients only need to implement RSVP stubs instead of a full protocol daemon (see [23] for details).

The remote client extension can be realized through a single new message type, INITAPI, and reusing the LIH field of the RSVP_HOP object. A new flag in the SESSION object is used to distinguish whether a message registers or de-registers a client. Both registration and deregistration messages carry the local IP address of the client system as part of the RSVP_HOP object. The LIH field of this object is used to carry the local UDP port, which can be chosen arbitrarily by the clients. Clients address the remote RSVP daemon at a well-known port. Client registration is done using soft state. Refreshes can be triggered by the RSVP daemon to avoid any timer management and related complexity at the client side.

In order to evaluate the remote client extension, there is not much virtue in running large scale performance experiments, because in reality, a first-hop RSVP node is less likely to be challenged by requests from a lot of clients. Instead, the evaluation is focused on the code and state memory requirements and reveals that using the remote client extension provides significant savings [23].

C. Reactive Admission Control

There are interesting proposals to use load- or congestion-based packet marking for admission control at edge gateways [24,25,26], based on mathematical analysis and simulation. A software prototype realizing these concepts has been built in the framework of RSVP [27].

The abstract system design is shown in Fig. 6. The load signal from internal packet marking nodes is encoded in the packet stream and can be observed at egress nodes. Since the benefits of admission control are moot without proper traffic regulation, respective information has to be transmitted from the egress to the ingress gateway. It does not matter, whether this information conveys a load report or admission control decision, but in case of RSVP, it can easily be piggybacked on reservation messages. Thereby, the receiver-initiated reservation sequence of RSVP is highly suitable to realize this concept. Because admission control in RSVP is usually done per outgoing interface, it is easier to transmit load information and perform admission control at the ingress node of each path. The resulting system prototype includes not only local RSVP signalling extensions, but also modules for packet handling in the data path of edge and internal nodes [27].

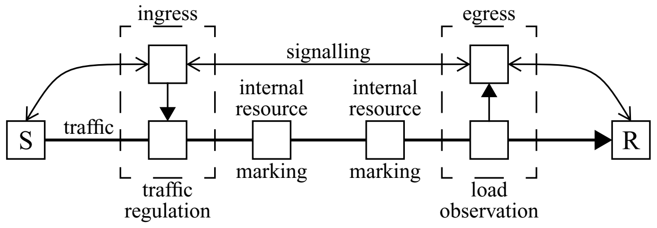

Figure 6: Reactive Admission Control

A detailed discussion and evaluation of the system prototype is reported in [27,28]. The most illustrative finding is given by the experimental verification that the system can deliver the same per-flow rate guarantees and resource utilization as a system employing per-flow scheduling at all internal nodes. In subsequent experiments, the system behaviour has been studied in larger topologies by means of network simulation [29]. These experiments are based on the availability of integrated software for real platforms and the ns-2 simulation environment, and thereby employ and verify the concepts presented in Section IV.F.

D. Firewall Signalling

Traditional firewalls have a static configuration to allow traffic only for certain “secure” transport protocol ports. Multimedia applications, however, dynamically allocate ports during the initial session setup and thus, cannot inter-operate with traditional firewalls. In order to solve this problem, a signalling protocol is needed for communication between end systems and firewalls, such that firewalls can dynamically adapt their filter state to such sessions. The information that needs to be exchanged, is similar to that of QoS signalling, therefore RSVP can be used to carry the necessary information between end systems and firewalls. A prototype has been implemented as a proof of concept of this proposal [30] and the basic idea has been picked up for discussion within the IETF [31].

VII. Related Work

Little work has been reported to assess the performance of commercial RSVP implementations. A notable exception is the work of Neogi et al. [32], in which a technical framework for such tests is presented. From the performance figures for a “commercial midrange router” as given in [32], it can be deduced that RSVP flow setup scales significantly worse than linear. These results indicate that the particular implementation under consideration may have been in a rather early development stage. Performance figures are listed in [33] for a commercial RSVP implementation. It cannot handle more than 600 sessions and thus, can also be regarded as premature. Other published work describes the implementation of RSVP in a switch-router in [34], but the reported performance figures are aiming towards the QoS objectives, rather than performance of signalling at a large scale. The work in [33] also considers the ISI implementation [22], which cannot be regarded as the optimal choice for performance measurements, as shown earlier [21]. In [35], interesting performance figures are reported for RSVP message processing on a commercial router. However, these numbers are not the central focus of the work in [35] and not many details about the experiments are given. Consequently, these numbers can only serve as a basic indication of RSVP's processing overhead. The work in [36] investigates a lightweight signalling protocol and also arrives at the conclusion that the cost of a user-level implementation is dominated by interaction with the operating system kernel.

In summary, although earlier work already indicates some of the conclusions, this work presents the most complete study of RSVP's design and performance from an implementation point of view.

VIII. Summary and Conclusions

In this paper, a comprehensive evaluation of RSVP is presented, which is based on significant implementation experience and a large number of performance-oriented experiments. The study not only focuses on basic RSVP signalling, but also takes into account a variety of standardized and experimental protocol extensions. All results are based on a publicly available implementation. The intended flexibility of RSVP is verified by implementing a number of experimental extensions with limited implementation effort. However, flexibility comes at the price of processing overhead for message parsing.

This article presents detailed insight about the implementation and optimization of a protocol engine. The execution effort is analysed for different configurations and for different processing blocks and message types. While much of the effort is caused by interaction with the operating system, it is still an open question whether a relatively complex protocol like RSVP could realistically be implemented at the kernel level. However, internal design decisions like dedicated memory management would clearly alleviate the task of creating a kernel-based implementation, in this case by reducing the dependency on flexible heap memory management, which is only available to user-level processes.

It is beyond the scope of this work to speculate whether RSVP's execution overhead is justified by its functionality. The experimental results demonstrate that the prototype implementation can handle 2,000 transactions/sec on slow hardware. Comparing this to 10,000 transactions/sec reported for a more lightweight protocol on faster hardware [36] leads to the conclusion that an ultimate decision for a signalling protocol can hardly be based on technical and performance issues only. Instead, a holistic point of view is necessary, which also takes into account all tradeoffs between performance and functionality.

Ongoing work in the context of this implementation is concerned with moving small parts of a user-level daemon into the operating system kernel to reduce the number of context-switches while retaining the flexibility of a user-level implementation as much as possible.

Acknowledgements

This work has been carried out with the help of many individuals and support from various sources. Particularly, I would like to acknowledge the cooperation with Jens Schmitt and the support of Ralf Steinmetz. Feedback from the anonymous reviewers and Tim Brecht helped enormously to improve the presentation.

References

[1] Next Steps in Signaling (nsis). IETF Working Group, see http://www.ietf.org/html.charters/nsis-charter.html.

[2] 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.

[3] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. RFC 2748 - The COPS (Common Open Policy Service) Protocol. Standards Track RFC, January 2000.

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

[5] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network Magazine*, 7(5):8–18, September 1993.

[6] J. Boyle. RSVP Extensions for CIDR Aggregated Data Flows. Internet Draft draft-ietf-rsvp-cidr-ext-01.txt, December 1997. Expired. Available from http://www.watersprings.org/pub/id/draft-ietf-rsvp-cidr-ext-01.txt.

[7] A. Terzis, J. Krawczyk, J. Wroclawski, and L. Zhang. RFC 2746 - RSVP Operation Over IP Tunnels. Standards Track RFC, January 2000.

[8] F. Baker, C. Iturralde, F. L. Faucheur, and B. Davie. RFC 3175 - Aggregation of RSVP for IPv4 and IPv6 Reservations. Standards Track RFC, September 2001.

[9] D. O. Awduche, L. Berger, D.-H. Gan, T. Li, V. Srinivasan, and G. Swallow. RFC 3209 - RSVP-TE: Extensions to RSVP for LSP Tunnels, December 2001.

[10] P. Pan, E. Hahne, and H. Schulzrinne. BGRP: A Tree-Based Aggregation Protocol for Inter-domain Reservations. *Journal of Communications and Networks*, 2(2):157–167, June 2000.

[11] L. Berger, D.-H. Gan, G. Swallow, P. Pan, F. Tommasi, and S. Molendini. RFC 2961 - RSVP Refresh Overhead Reduction Extensions. Standards Track RFC, April 2001.

[12] R. Braden and L. Zhang. RFC 2209 - Resource ReSerVation Protocol (RSVP) – Version 1 Message Processing Rules. Informational RFC, September 1997.

[13] M. Karsten. Design and Implementation of RSVP based on Object-Relationships. In *Proceedings of Networking*

2000, Paris, France, pages 325–336. Springer LNCS 1815, May 2000.

[14] 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.

[15] 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.

[16] 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. Proceedings of IDMS'99.

[17] D. Zappala and J. Kann. RSRR: A Routing Interface for RSVP, 1998. available from http://www.ietf.org/proceedings/98aug/I-D/draft-ietf-rsvp-routing-02.txt.

[18] K. Fraser and P. Quiney. MPLS on Linux, September 2000. Software and Documentation available at http://www.cl.cam.ac.uk/Research/SRG/netos/netx/index.html.

[19] J. R. Leu. MPLS for Linux, November 2001. Software and Documentation available at http://sourceforge.net/projects/mpls-linux.

[20] K. Fall and K. Varadhan, editors. *The ns Manual*. April 2002. Software and Documentation available at http://www.isi.edu/nsnam/ns/.

[21] M. Karsten, J. Schmitt, and R. Steinmetz. Implementation and Evaluation of the KOM RSVP Engine. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01), Anchorage, USA*, pages 1290–1299. IEEE, April 2001.

[22] USC Information Sciences Institute. RSVP Software, 1999. http://www.isi.edu/div7/rsvp/release.html.

[23] M. Karsten. Experimental Extensions to RSVP - Remote Client and One-Pass Signalling. In *Proceedings of the 9th IEEE/IFIP International Workshop on Quality of Service (IWQoS'01), Karlsruhe, Germany*, pages 269–274. Springer LNCS 2092, June 2001.

[24] R. J. Gibbens and F. P. Kelly. Distributed Connection Acceptance Control for a Connectionless Network. In *Proceedings of the 16th International Teletraffic Congress - ITC 16, Edinburgh, UK*, June 1999.

[25] R. Andreassen, editor. *M3I Deliverable 1 - Requirements Specification Reference Model*. EU 5th Framework, Program IST, Project 11429 (M3I), June 2000. Available from http://www.m3i.org/results/m3idel01v7_1.pdf.

[26] T. Kelly. An ECN Probe-Based Connection Acceptance Control. *ACM Computer Communication Review*, 31(3):14–25, July 2001.

[27] M. Karsten and J. Schmitt. Admission Control based on Packet Marking and Feedback Signalling - Mechanisms, Implementation and Experiments. Technical Report TR-KOM-2002-03, Darmstadt University of Technology, May 2002.

[28] M. Karsten and J. Schmitt. Market-Based Resource Allocation for Packet-Switched Networks. In *Proceedings of the 10th International Conference on Telecommunication Systems Modelling and Analysis (ICTSM10), Monterey, USA*, pages 52–62, October 2002.

[29] M. Karsten and J. Schmitt. Packet marking for integrated load control. In *Proceedings of 9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, May 2005.

[30] U. Roedig, M. Goertz, M. Karsten, and R. Steinmetz. RSVP as Firewall Signalling Protocol. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC), Hammamet, Tunisia*, pages 57–62. IEEE, July 2001.

[31] M. Shore. Towards a Network-friendlier Midcom. Expired Internet Draft, June 2002. Expired. Available from http://www.employees.org/ shore/draft-shore-friendly-midcom-01.txt.

[32] A. Neogi, T. Chiueh, and P. Stirpe. Performance Analysis of an RSVP-Capable Router. *IEEE Network Magazine*, 13(5):56–63, September 1999.

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

[34] 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.

[35] P. Pan and H. Schulzrinne. YESSIR: A Simple Reservation Mechanism for the Internet. *ACM Computer Communication Review*, 29(2):89–101, April 1999.

[36] P. Pan and H. Schulzrinne. Processing Overhead Studies in Resource Reservation Protocols. In *17th International Teletraffic Congress (ITC), Salvador, Brazil*, September 2001.