[Ka00] *Martin Karsten:* **Design and Implementation of RSVP based on Object Relationships;** Networking 2000

Design and Implementation of RSVP based on Object-Relationships

Martin Karsten *
Industrial Process and System Communications
Darmstadt University of Technology
Merckstr. 25 • 64283 Darmstadt • Germany
phone: +49-6151-166156 • fax: +49-6151-166152
email: Martin.Karsten@KOM.tu-darmstadt.de
http://www.kom.e-technik.tu-darmstadt.de

Abstract RSVP has been proposed by the IETF as a signalling protocol for reservation-based quality-of-service enabled communication in IP networks. While RSVP's concepts are very sophisticated, further research efforts and potential modifications might be necessary to accomplish additional requirements before general deployment and commercial usage. Currently, only one freely available implementation exists and even some of the commercial implementations are based on it. In this paper, an alternative approach to describe RSVP protocol operations is presented, employing relational specification of state blocks and object-relationships between them. The result appears to be more concise and comprehensible than existing processing rules, yet not giving up efficiency. An implementation design based on this methodology, as well as specific details and optimizations are derived and explained. The implementation is designed to be portable across different operating system platforms and even to simulation environments. The primary purpose is to carry out research on modifications of RSVP, being able to examine those by simulation, emulation and real tests. Applying these considerations, an experimental protocol engine has been implemented, which is publicly available.

1 Introduction

RSVP (Resource ReSerVation Protocol), initially designed and described in [1], has been specified by the IETF [2] to carry reservation requests for communication resources across IP networks. Because RSVP is designed to handle requests for arbitrary service classes, an even more general point of view can be adopted by regarding it as a universal signalling protocol to carry quality of service requests.

For a variety of reasons [3], I believe that further research is needed to determine the best design of such a signalling protocol. This research can be grounded on the existing specification of RSVP, because of both its basic existence and its sophisticated design. The only existing freely available implementation [4] is not considered well-suited for such research (see [3] for details). An attempt to create a more suitable test environment should adhere to the following design objectives:

- structured (object-oriented) design and implementation
- portability for multiple platforms, including simulators
- clear representation of RSVP's concepts in the code

While at a first glance RSVP seems to be straightforward and easy to understand, the details of an implementation are rather complex. The goals of this project are twofold. The first goal is to specify protocol operations more comprehensible than existing doc-

*. This work is sponsored in part by: Volkswagen-Stiftung, Hannover, Germany.

umentation does. The second goal is to create and publish an experimental platform which allows researchers to test and examine modifications with reasonable effort. The context and initial motivation of this implementation project are in the areas of charging for QoS in network communication [5,6,7] and interoperability of heterogeneous QoS architectures [8,9].

The rest of this paper is organized as follows. In the next section, a brief overview of RSVP is given, adopting the terminology of [2]. A specification of RSVP message processing, based on object-relationships, is presented in Section 3. In Section 4, an appropriate software design approach is derived from this specification. The current status of the implementation with respect to the RSVP specification is described in Section 5. Section 6 concludes the paper with a summary and an outlook on further work items. Note that this paper is shortened in order to accomplish the space limitation for its publication here. A more detailed version is available as [3].

2 RSVP Overview

RSVP is designed to carry reservation requests for packet-based, stateless network protocols such as IP (Internet Protocol). In essence, it is aimed at combining the robustness of connectionless network technology with flow-based reservations by following a so-called *soft state* approach. State is created to manage routing information and reservation requests, but it times out automatically, if it is not refreshed periodically. In the RSVP model, senders inform RSVP-capable routers and receivers about the possibility for reservation-based communication by advertising their services via PATH messages. These messages carry the sender's *traffic specification* (TSpec) and follow exactly the same path towards receivers as data packets, establishing soft state in routers. Receivers initiate reservations by replying with RESV messages. They contain a TSpec and a *reservation specification* (RSpec) and also establish soft state representing the reservation. RESV messages are transmitted hop-by-hop and follow exactly the reverse path that is formed by PATH messages.

RSVP treats reservation requests (e.g. TSpec and RSpec) as opaque data and hands them to complementary local modules, which are able to process them appropriately. Being tuned to support large multicast groups, RSVP uses logic from these modules to merge reservation requests that share parts of the transmission path. Merging takes place at outgoing interfaces by merging requests from different next hops that can be satisfied by a single reservation at the same interface. As well, reservation requests that are transmitted towards a common previous hop are candidates for merging. The amount of merging possible is determined by the filter style, which is requested by receivers. For shared filter style, all reservations for the same interface and all reservations towards the same previous hops are merged, respectively. When distinct filter style is requested, only reservations that specify the same sender are being merged. Furthermore, filter styles are classified by whether applicable senders are wildcarded or listed explicitly. The (potentially empty) list of senders is called *FilterSpec*. The following filter styles are currently defined:

- FF (fixed filter): single sender, distinct reservation
- SE (shared explicit): multiple senders, shared reservation
- WF (wildcard filter): all senders, shared reservation

All these filter styles are mutually exclusive and a session's filter style is determined from the first arriving RESV message. The combination of TSpec and RSpec is called *flow specification* (FlowSpec). The combination of FlowSpec and FilterSpec is referred to as *flow descriptor*.

3 Specification of RSVP Message Processing

In this section, a specification of RSVP message processing is presented, based on relational design and object-relationships between state blocks. A rigorous approach for modelling RSVP would begin by representing state information as relations and identifying functional dependencies between them. Then, well-known normalisation algorithms could be applied to create the highest possible normal form and message processing could be expressed using relational algebra. Intuitively, this is often done to some extent by software designers and programmers.

In this work, while not following the strict method, state information is explicitly modelled as relations which in turn are considered as state blocks to create object-relationships between them. The initial relational model is deduced from the relevant standardization documents [2,10,11] and personal reasoning about the protocol. Additionally, experiences made during design and implementation of the software have been a source of insight into protocol operations. We omit the details of relational representation here for reasons of brevity and refer to [3].

A significant part of RSVP message processing consists of finding appropriate state blocks for certain operations. For normal implementation (i.e. without using a relational database), state blocks and object-relationships are considered to be more expressive and efficient than directly implementing the relational model. The relationships between objects are explicitly stored when knowledge is available, instead of recalculating them through relational rules whenever they are needed. The algorithmic description in [10,11] exhibits a relational style, but without being rigorous. Opposite to that approach, the processing rules in this paper are based on object-relationships between state blocks. A subset of state blocks is similar to those described in [10,11], but semantics and lifetime are occasionally modified. Additional relations are designed to express useful state information. Eventually, these are represented as state block objects as well, to efficiently accomplish certain operations.

3.1 State Blocks and Relationships

From the initial relations, corresponding state block objects and state block relationships are deduced. Although this is not done rigorously (i.e. by using normalisation algorithms), certain optimizations are possible to avoid redundancy of informa-

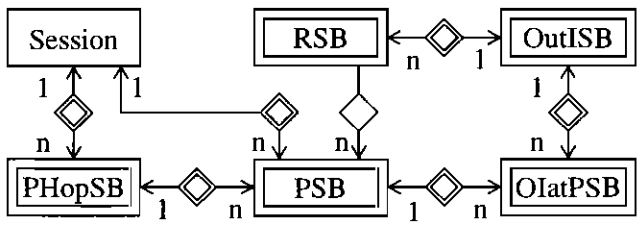

Figure 1: Entity-Relationship Diagram for State Blocks

tion and suit an efficient implementation. The result can be expressed as an Entity-Relationship Model (ER-Model) and is shown as diagram in Figure 1.

3.1.1 State Blocks

Session For each RSVP session, the Session state block bundles all relevant information and the session's destination address and port is saved there. Relationships are kept to those state blocks that are needed to fully access all information. All Session objects are bound to a single RSVP object, representing an RSVP router.

Path State Block (PSB) A PSB holds all relevant information from a PATH message, i.e., the sender's address and traffic specification, routing information, etc.

Reservation State Block (RSB) An RSB represents a reservation requested from a next hop, particularly by holding the reservation specification, i.e., the FlowSpec, which determines the amount of resources that are requested, depending on the service class. It identified by its owning OutISB and the next hop's address.

Outgoing Interface State Block (OutISB) This state block represents the merged reservations from multiple RSBs applying at a certain outgoing interface. It is roughly comparable to the TCSB in [10,11]. However, different from those processing rules, the TCSB is split into a general (OutISB) and a specific part. The nature of the specific part depends on the particular traffic control implementation, which in turn depends on the corresponding link layer medium behind the interface [2,12,8]. An instance of OutISB is constructed immediately upon creation of the first contributing RSB.

Outgoing Interface at PSB (OIatPSB) For each outgoing interface that is part of the routing result for a PATH message, an instance of OIatPSB is created. A relationship to an OutISB object expresses that an actual reservation is active at this interface. The introduction of this state block allows to split the N:M relationship between PSB and OutISB into two 1:N relationships, which simplifies implementation.

Previous Hop State Block (PHopSB) The concept of an explicit PHopSB is new to an RSVP description. It is used to hold information about reservations that are merged at a certain incoming interface towards a previous hop, as well as the resulting reservation request that is sent to this hop. A PHopSB is identified by the previous hop's IP address and the incoming interface, at which traffic from this hop arrives for the destination address of a session. Again, an object is created as soon as the first PATH message arrives from a certain previous hop.

In the rest of the paper, the terms *state block* and *state block object* are used synonymously. In Figure 1, all entities except Session are weak entities, i.e., they cannot uniquely be identified without the respective session key. Furthermore, OutISB is indirectly identified through an OIatPSB at any of the PSBs it applies to, although the cardinality ratio implies the opposite direction. In RSB there is no information about the outgoing interface stored and the list of senders is not used for identification. Thereby, it is a weak entity depending on the key of OutISB. For both RSB and OutISB, instead of storing the set of applicable senders, a relationship to PSB is maintained (indirectly in case of OutISB). The cardinality ratio of each relationship is shown in the diagram in Figure 1.

3.1.2 Relationships

Each relationship is presented, including the necessary key to traverse it, if it is a multi-object relationship. The respective keys applying to these relationships are often smaller than the full key of each state block. This is due to inherent identification through the relationships. However, the implementation of this model is done by directly storing the relationships. Furthermore, all Session objects are bundled into a global container. This could be considered as a special relationship to a unique object representing the RSVP router.

Session $\leftarrow\!\diamond\!\rightarrow_{1\ n}$ **PSB** key for PSB: Sender, Incoming Interface and Previous Hop (1)
In a PSB object, information about incoming interface and previous hop are not stored directly, instead this information can be extracted from the corresponding PHopSB (see Relationship (7) below).

Session $\leftarrow\!\diamond\!\rightarrow_{1\ n}$ **PHopSB** key for PHopSB: Incoming Interface and Previous Hop (2)

OutISB $\leftarrow\!\diamond\!\rightarrow_{1\ n}$ **RSB** key for RSB: Next Hop (3)
Each OutISB is related to those RSBs that are merged together at an outgoing interface. Because an RSB only contributes to one specific OutISB, partitioning the set of RSBs along their OutISBs creates a complete and disjunct decomposition of all RSBs. Therefore, a relation between Session and RSB is not necessary to access RSBs from a Session object.

RSB $\rightarrow\!\diamond\!\rightarrow_{n}$ **PSB** key for PSB: Sender (4)
A reservation applies to a set of senders, either by explicit selection (SE or FF filter style) or implicit association (WF filter style). Instead of storing a list of all sender addresses, a relationship to the respective PSBs is maintained from the RSB.

OIatPSB $\leftarrow\!\diamond\!\rightarrow_{n\ 1}$ **PSB** key for OIatPSB: Outgoing Interface (5)
A merged reservation, installed at an outgoing interface, applies to a set of senders. As with RSBs, this is expressed by storing relationships to the respective OIatPSBs, instead of their address/port pairs.

OIatPSB $\leftarrow\!\diamond\!\rightarrow_{n\ 1}$ **OutISB** key for OIatPSB: Sender (6)
This relationship expresses the reservation at a certain outgoing interface that is applied to traffic from a sender.

PHopSB $\leftarrow\!\diamond\!\rightarrow_{1\ n}$ **PSB** key for PSB: Sender (7)
Each PSB is logically connected to the PHopSB representing its previous hop. This relationship is mainly used when reservation requests are created for previous hops. Information from the PHopSB (hop address and incoming interface) is used to distinguish PSB objects (see Relationship (1) and Path State relation).

3.2 Operations

In this section, the core operations of the RSVP protocol engine are explained with respect to the relationships between state blocks. The presentation is divided into 4 parts, which together form the central RSVP operations:

- State Maintenance

- Outgoing Interface Merging
- Incoming Interface Merging
- Timeout Processing

In general, if a message or timeout triggers a modification of internal state, all relationships are updated immediately during State Maintenance or Timeout Processing, except the relationship between PSB and OutISB. For Outgoing Interface Merging, the "old" state of this relationship has to be available to appropriately modify the filter setting at the underlying traffic control module. Afterwards, this relationship is updated, as well. If the contents of an RSB change, Outgoing Interface Merging is invoked. If during Outgoing Interface Merging, the contents of an OutISB are changed, Incoming Interface Merging is triggered. Only if the resulting reservation request (stored in PHopSB) changes, a new RESV message is created and sent to the previous hop.

The basic claim of this work is that maintaining relationships imposes no significant additional overhead during analysing an incoming message and updating state from it. However, when it comes to merging reservations and timeout processing, existing relationships can be used instead of recomputing them every time, especially under stable conditions. In this section, only the basic operations are described. Whenever the term *interface* is used in the following subsections, it might also denote the API (application programming interface). Again, additional details about message pre- and post-processing can be found in [2,10,11].

3.2.1 State Maintenance

Arriving RSVP messages are decomposed into components and processed depending on the type of message. During processing, appropriate state blocks have to be located, created and/or modified. In the following, a pseudo-algorithmic description of the processing rules are given for each message type. Although it is not mentioned explicitly for most of the message types, usually the appropriate Session object has to be determined first.

PATH Find a Session and check for conflicting destination ports. If no Session exists, create one. Find a PSB for this sender through Relationship (1) and check for conflicting source ports. If none exists, create a new PSB. When creating a PSB object, create the relationship to the corresponding Session object. If the PSB is new and no appropriate PHopSB can be found, create a new PHopSB. Set a relationship between PSB and PHopSB. If the session address is multicast and the incoming interface differs from the routing lookup result, mark this PSB as local to an API session. Update all information in the PSB and in case of relevant changes, trigger an immediate generation of a PATH message and potentially invoke Outgoing Interface Merging (Section 3.2.2).

RESV Process each flow descriptor separately, i.e., each pair of FilterSpec and FlowSpec. Find or create an appropriate OutISB through Relationship (5) and (6) and find or create an RSB using Relationship (3). When creating new objects, set the corresponding relationships. Match (i.e. consider the intersection) the filter specification (in case of FF or SE) or the address list determining the scope (WF) against all existing PSBs that route to the outgoing interface through Relationship (1). Update the RSB and invoke Outgoing Interface Merging, if relevant content has changed, e.g., FilterSpec or FlowSpec.

PTEAR Find a PSB through Relationship (1). If found, forward the message to the PSB's outgoing interfaces, remove the PSB, clear its relationships and invoke Outgoing Interface Merging.

RTEAR Process each flow descriptor separately. Find an RSB through Relationship (5) and (6) and Relationship (3). If found, remove the filters that are listed in the message and invoke Outgoing Interface Merging. If the RSB's filter list is empty, remove the RSB and clear its relationship to OutISB.

PERR Find a PSB through Relationship (1). If found, forward the message through the PSB's incoming interface.

RERR Find a PHopSB for the previous hop address from the message. If found and the error code indicates an admission control failure, set a blockade FlowSpec at those PSBs from the PHopSB that match a filter from the message. Find all OutISBs that match a filter from the message and do not belong to the incoming interface. Forward the message to all RSBs that have a relationship to these OutISBs. In case of admission control failure, forward the message to only those RSBs that do not have a FlowSpec strictly smaller than that of the message.

RCONF Forward the message to the outgoing interface that results from a routing lookup for the message's destination address.

3.2.2 Outgoing Interface Merging

During the merge operation at an outgoing interface, all applicable PSBs and RSBs have to be collected to access their TSpecs and FlowSpecs. Precise operation depends on the nature of the underlying link layer and appropriate algorithmic descriptions can be found for point-to-point or broadcast media in [10,11] and for non-broadcast multiaccess media (e.g. ATM) in [13,12,8]. Outgoing Interface Merging operates on a certain OutISB. Relationships to those PSBs that are relevant and route to this interface as well as RSBs that contribute to the merged reservation state are known and can be traversed directly, instead of recomputing them. Therefore, no special (filter style dependent) rules have to be given on how to find those state blocks, but instead only rules to process them appropriately are necessary. The result is stored in the OutISB and, if the merged FlowSpec or the FilterSpec has changed, the appropriate PSBs and PHopSBs (accessible through Relationship (5), (6) and (7)) are marked for Incoming Interface Merging. Certain policing flags have to be passed to traffic control, which can be derived from accessible information, as well. To determine whether this reservation is merged with any other reservation that is not less or equal, the LUB (least upper bound) of all merged FlowSpecs from all OutISBs (at different interfaces) for all PSBs can be calculated by traversing Relationship (5) and (6). If afterwards the OutISB's filter list is empty (which must coincide with having no relations to RSBs), remove the OutISB and clear its relationship to Session.

3.2.3 Incoming Interface Merging

After a single or multiple (in case of RESV message processing) invocations of Outgoing Interface Merging, all PHopSBs that are marked for update are subject to Incoming Interface Merging. During this sequence, it is again possible to traverse relationships, instead of collecting state blocks. The details of this merging operation depend on the

filter style for the session. In case of distinct reservations (FF), each PSB that relates to the PHopSB is considered separately. All OutISBs accessible through this PSB are merged and a flow descriptor is created, containing the PSB's sender address and the merged FlowSpec. For shared reservations, all OutISBs having a relationship to any of the PSBs are merged and the resulting flow descriptor contains the set of all sender addresses and the single merged FlowSpec.

3.2.4 Timeout Processing

According to the soft state paradigm, each state block is associated with a timer and deleted upon timeout. Periodic refresh messages restart the timer. Timers are directly connected to the object they apply to and the actions resulting from a timeout are similar to those when receiving a PTEAR or RTEAR message. The only difference is that the respective PTEAR/RTEAR message has to be created instead of just forwarding it.

4 Software Design

Given the objectives of the project, these goals have been set for an implementation:

- Message handling (creation/interpretation) should be clear, simple and extensible.
- Message processing should be clear and comprehensible, yet efficient.
- The implementation should be portable, but also nicely integrate with system level interfaces.

The design that has been chosen is a hybrid form of object orientation and procedural design. Object orientation does not seem to be fully appropriate for implementing state machines like network protocol engines, however, many aspects of an implementation can benefit from data encapsulation, inheritance and polymorphism. C++ has been selected as the programming language of choice to implement such a hybrid design under the given objectives. In the following, identifiers stemming from the implementation are printed in italic, when they are introduced. Figure 2 gives an overview of the design. In this picture, the main components, which together form the contents of a global *RSVP* object, are shown. An RSVP object represents an RSVP-capable router and interacts through abstract interfaces with system-dependent services like routing, network I/O, traffic control and others. Multiple *Session* objects exist, representing currently active RSVP sessions. A number of *LogicalInterface* objects encapsulate physical and virtual interfaces of the un-

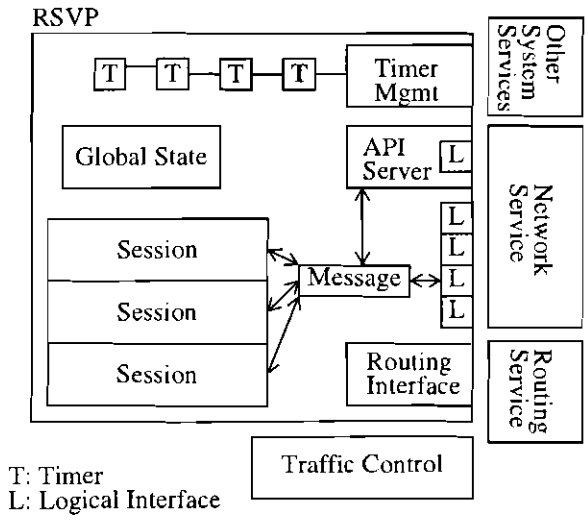

Figure 2: Design of RSVP Implementation (Overview)

derlying system. Logical interfaces are numbered and the number is used as LIH (Logical Interface Handle, see [2] for details). The API is modelled as a dedicated object, called *API_Server*, containing a special instance of class LogicalInterface, and all information about currently active API clients. RSVP messages are encapsulated in a *Message* class and passed between LogicalInterface and Session objects, potentially involving API_Server. Global state is kept in the RSVP object, for example, the current message, a PHopSB refresh list, etc.

4.1 Message Processing

Each incoming message arrives at the main RSVP object. After preprocessing and updating global state, the message is dispatched to the appropriate Session object for further processing. Some of the message processing rules from Section 3.2.1 are carried out in the RSVP object (e.g. finding or creating a Session object), but the majority is implemented in class Session.

The sequence Outgoing Interface Merging is link-layer dependent and consequently, functionality is split up. Common merging logic is implemented in class OutISB. A base class *TrafficControl* provides basic services and a uniform calling interface for the link-layer specific part. This calling interface takes an instance of OutISB and potentially a list of newly arrived filters as input parameters. All state that is needed for admission control and updating of the underlying scheduling system is then accessible through OutISB. Both classes TrafficControl and OutISB are inherited by link-layer specific classes.

Incoming Interface Merging takes place when reservation state has changed, that is, if FlowSpec or FilterSpec of an OutISB has been modified. It is implemented in class RSVP, so that it can directly access all relevant global state information.

4.2 Implementation Details

Relationship Representation Relationships are implemented as dedicated classes, which are used as base classes for those classes they apply to. The relationship classes automatically maintain referential integrity. A single-object relationship is internally represented by a pointer or reference, whereas a multi-object relationship is internally represented by a sorted list of pointers to the respective objects. Relationship (5) is internally stored as an array of pointers, because at most one OutISB exists at each interface and can be accessed directly by using the interface's unique LIH as index.

Timers Timer management is logically separated from the rest of the implementation, such that it can be independently optimized without considering other parts of the code. A base class *BaseTimer* exists, from which refresh and timeout timers are derived. They are controlled by their owners, but handled commonly through BaseTimer. Currently, all timers are kept in a container, ordered by their expiration time. This design completely hides implementation details between timer management and timer clients.

Container Classes A simple container library for lists and sorted lists has been implemented, in a style similar to the C++ STL (Standard Template Library). While it is conceptually very advantageous to use common container classes, it seems not necessary to provide the most efficient implementation for them. It is left to the user of this implementation to decide whether outmost efficiency is required when accessing certain con-

tainers or not. Because of the encapsulated design, testing of different algorithms and data layouts for containers is possible with relatively low effort.

4.3 Lessons Learned

It seems clear that introducing PHopSB and OutISB as important central state blocks representing merging state provides advantages due to their naturally given relations to RSBs and PSBs. The notion of recalculating relationships at every stage of message processing seems sub-optimal compared to maintaining and traversing these relationships. Some additional details are listed in [3].

5 Implementation Status

In this section, the current implementation status is described in comparison to the RSVP specification. This implementation is a full implementation of RSVP operations, except certain limitations given below. It is developed and tested to automatically compile on Solaris 2.6, FreeBSD 3.X and Linux 2.X operating systems, using GNU C++ 2.95 and higher. The complete source package consists of approximately 19,000 lines of code. System-dependent code is cleanly separated and consists of about 2,000 lines of code with at most 150 lines dedicated to each system. The software is publicly available from http://www.kom.e-technik.tu-darmstadt.de/rsvp/.

5.1 Features

The implementation already provides some features that are new to an RSVP implementation and rather rare for experimental signalling protocols in general.

RSVP can be run in an emulation mode, in which multiple daemons execute on the same or differing machines and use a configurable virtual network between them, including shared link media and static multicast routing. Without such a feature, examinations of RSVP protocol behaviour in non-trivial network topologies are only possible by using a simulator or by using real systems. In the second case, it is necessary to start multiple processes on multiple machines needing super-user privileges and a suitable infrastructure. The emulation mode allows to experiment without the need for additional software nor hardware. A test-suite can be created by writing high-level configuration files, from which detailed configuration files are built with a special tool. A preconfigured test-suite consisting of 16 virtual nodes and including test scenarios already exists. Furthermore, the emulation mode can be combined with real operation, for example, to test interoperability with other implementations.

Communication between RSVP daemon and API clients uses soft state. This is deemed useful in cases when RSVP operates on a router on behalf of an API client at a different host. There is no need for complicated connection management and the API can be treated similar to an ordinary RSVP hop. It is configurable at compile time to have asynchronous API upcalls realized by signals or by using threads. Many other options devised for testing purposes are configurable at compile time, as well.

5.2 Limitations

Some of the properties of a full compliant RSVP implementation are currently missing. The main reason for them to be missing is their relative importance with respect to the project goals, compared to the effort necessary to develop and test these features.

- IPv6 is currently not supported. Due to the modular and portable design of the software, this should not create too much effort, yet it has to be tested then.
- UDP encapsulation as described in [2] is not supported. It is not planned to support this in the future, because it does not belong to the core of the specification and it is already discussed in the IETF to drop this requirement [14].

5.3 Traffic Control Interface

.An interface to real packet scheduling is provided for the CBQ package on Solaris and for HFSC and CBQ scheduling using the ALTQ package on FreeBSD (see [3] for appropriate references). In the absence of real scheduling packages, the total amount of available bandwidth can be configured per interface. For each reservation, the necessary resource requirements are calculated in terms of service rate and buffer. The results are checked against the available capacity and logged.

6 Summary and Future Work

In this paper I have presented an implementation of RSVP that is based on different design and implementation paradigms than existing work. The description of RSVP operations becomes more comprehensible when using object-relationships as principle method of describing state blocks and message processing. Furthermore, a high-level description of processing rules can be translated into implementation details and vice versa with less semantic deprivation. A brief specification of RSVP processing rules is presented to demonstrate the capabilities of this approach. Certain optimizations have been carried out and additional tuning seems possible, if an implementation is based on maintaining object-relationships instead of recomputing them when needed. Design objectives for an experimental RSVP platform have been formulated and a design for an RSVP implementation is presented, following these objectives and being based on object-relationships. To a large extent, the design objectives have been met by the prototype. It is shown in this paper, how an experimental research platform can benefit from the application of modern software principles. The implementation described in this paper will be publicly available to the research community.

There is still a lot of research work to be carried out in the area of signalling resource requirements. A formal specification of RSVP in terms of relational algebra could be derived from the results of this work. This in turn could be used for formal verification of protocol implementations and modifications. Many potential protocol refinements remain open for examination. For this project, it is planned to further extend and tune the implementation as well as completing to port it to a simulation environment, which is already under way. Additionally, the research issues and existing proposals that are mentioned in Section 1 are going to be explored based on this implementation. Finally, results from this project might be useful when new proposals for new signalling protocols are being discussed, implemented and tested.

Acknowledgments

Jens Schmitt implemented the initial integration of the CBQ and ALTQ packages. I also acknowledge the help of Jens Schmitt and especially Nicole Beriër during preparation of this paper.

References

[1] 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.

[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] M. Karsten. Design and Implementation of RSVP based on Object-Relationships. Technical Report TR-KOM-2000-01, Darmstadt University of Technology, February 2000. Avalaible at ftp://ftp.kom.e-technik.tu-darmstadt.de/pub/TR/TR-KOM-2000-01.ps.gz.

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

[5] M. Karsten, J. Schmitt, L. Wolf, and R. Steinmetz. An Embedded Charging Approach for RSVP. In *Proceedings of the Sixth International Workshop on Qality of Service (IWQoS'98), Napa, CA, USA*, pages 91–100. IEEE/IFIP, May 1998.

[6] M. Karsten, J. Schmitt, L. Wolf, and R. Steinmetz. Provider-Oriented Linear Price Calculation for Integrated Services. In *Proceedings of the Seventh IEEE/IFIP International Workshop on Quality of Service (IWQoS'99), London, UK*, pages 174–183. IEEE/IFIP, June 1999.

[7] M. Karsten, N. Berier, L. Wolf, and R. Steinmetz. A Policy-Based Service Specification for Resource Reservation in Advance. In *Proceedings of the International Conference on Computer Communications (ICCC'99), Tokyo, Japan*, September 1999.

[8] J. Schmitt, L. Wolf, M. Karsten, and R. Steinmetz. VC Management for Heterogeneous QoS Multicast Transmissions. In *Proceedings of the 7th International Conference on Telecommunications Systems, Analysis and Modelling, Nashville, Tennessee*, March 1999.

[9] J. Schmitt, M. Karsten, L. Wolf, and R. Steinmetz. Aggregation of Guaranteed Service Flows. In *In Proceedings of the Seventh International Workshop on Qality of Service (IWQoS'99), London, UK*, pages 147–155. IEEE/IFIP, June 1999.

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

[11] B. Lindell, R. Braden, and L. Zhang. Resource ReSerVation Protocol (RSVP) – Version 1 Message Processing Rules. Internet Draft, February 1999. Work in Progress.

[12] J. Schmitt. Extended Traffic Control Interface for RSVP. Technical Report TR-KOM-1998-04, Darmstadt University of Technology, July 1998. Avalaible at ftp://ftp.kom.e-technik.tu-darmstadt.de/pub/TR/TR-KOM-1998-04.ps.gz.

[13] E. S. Crawley, L. Berger, S. Berson, F. Baker, M. Borden, and J. J. Krawczyk. RFC 2382 - A Framework for Integrated Services and RSVP over ATM. Informational RFC, August 1998.

[14] R. Braden. RSVP/IntServ MIB issues, June 23rd 1998. Contribution to rsvp mailing list. Available from ftp://ftp.isi.edu/rsvp/rsvp-1998.mail.