Internet Engineering Task Force
Internet-Draft
draft-kim-llrmp-01.ps

P. Kim
HP Laboratories Bristol
Expires: 6/97

Link Level Resource Management Protocol (LLRMP)

Protocol Specification - Version 1

December, 1996

Status of Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Abstract

This memo describes the LLRMP, a link level signalling protocol used to setup resources in shared-medium and bridged/switched LANs. Network layer resource management protocols like RSVP or STII, or a local network manager, may invoke the LLRMP to request a certain quality of service over a bridged LAN.

The LLRMP protocol supports a distributed admission control over bridged LANs. Resources are reserved independently on each segment, on a hop-by-hop basis, as a LLRMP reservation message travels along the data path from the source to the receiver.

Changes from the last Version

1. Added a section describing the LLRMP operation in a full-duplex switched LAN and the consequent simplifications that can be made to the protocol.
2. Added a section describing the RSVP Snooping Module and how this enables full receiver heterogeneity to be supported without changes to LLRMP or RSVP.
3. Added an appendix A.2 containing top-level pseudo-code for the LLRMP host and switch algorithms.
4. Added an appendix A.5 containing detailed pseudo-code for the above algorithms.

Table of Contents

1. Introduction 3
2. Protocol Properties 4
2.1 The LAN Model 4
2.2 The Reservation Model 4
2.3 The Receiver Model 5
2.4 Summary 5
3. Protocol Operation and Mechanisms 7
3.1 The LLRMP Message Types 7
3.2 The Reservation Setup in a Full-Duplex Switched LAN 10
3.3 The Optimistic Setup Strategy - Reservations without local ACK-Messages 11
3.4 The Pessimistic Setup Strategy - Reservations with local ACK-Messages 12
3.5 The Reservation Refresh Mechanism 13
3.6 The Dynamic Segment Arbiter Election 13
3.7 Request Modifications and Tear Down 14
3.8 Control-Message-Forwarding within Bridges/Switches 14
3.9 Dynamic Topology Changes 15
3.10 Heterogeneous bridged LANs with non-LLRMP Clouds 15
3.11 More Receiver Heterogeneity - The RSVP Snooping Module 15
4. LLRMP Relationship to RSVP 16
4.1 LLRMP Invocation 16
4.1 Interface 17
5. Functional Specification 17
6. References 17
A. Appendix 18
A.1 Host and Switch Control Structures 18
A.2 General Operation Overview 19
A.3 Switch Functions 21
A.4 Timer Values and Constants 22
A.5 Example Pseudo-Code 22

1. Introduction

The Link Level Resource Management Protocol (LLRMP) was designed for setting up resources in local, shared-medium and bridged/switched networks. This document describes the protocol operation and the mechanisms used to achieve this. The LLRMP can be used to dynamically reserve resources in heterogeneous bridged/switched networks composed of segments of a different technology e.g. Ethernet, Token Ring, FDDI and 100VG-AnyLAN. The reservation setup for those segments will only differ with regard to technology specific information which might need to be carried by the LLRMP and the actual admission control algorithm applied to control access to the link.

Whenever an application desires quality of service then it negotiates this service with the network layer management protocol e.g. RSVP [1]. RSVP then carries the request through the internetwork and initiates admission control on each of the intermediate links between source and receiver. When an intermediate link in the data path is a bridged LAN then RSVP passes the reservation request down to the LLRMP. In general, the LLRMP performs similar operations for the link layer as carried out by RSVP or STII [2] at the network layer: it carries the request through the bridged network and initiates admission control for each of the segments within the data path between the local source and local receiver. Resources are reserved when necessary and available. Unicast and multicast flows are supported. However the LLRMP is much simpler than RSVP and can be deployed on existing switches.

The LLRMP is a new link level protocol that runs in end-hosts and on bridges. All LLRMP control messages are sent with a new link level protocol identifier (e.g. a new *ethertype*) and are addressed to a multicast MAC address which is to be exclusively allocated for the protocol. The deployment of the LLRMP in bridges/switches can be performed gradually since the protocol can operate transparently through bridges and switches which do not support it. This would allow an administrator to initially control resources at a particular bottleneck within the bridged LAN. Note that if the link technology supports e.g. priorities then only end-hosts which use the higher priority access mechanism need to be updated. Nodes which only send with normal priority are not affected.

The LLRMP is independent of the network layer resource management. This allows the protocol to serve requests from different upper layer management entities e.g. RSVP, STII or a local network management system. Reservations are established and torn-down dynamically. The protocol further dynamically adapts to topology changes. Protocol state is only maintained in bridges along the data path and not in every bridge on the LAN. This relies on MAC address entries within bridges. We assume standard IEEE 802.1 learning bridges [5]. In order to forward control-messages efficiently, the LLRMP entity on a bridge needs access to the local MAC address table. However, the protocol does not perform any routing functionality. This is assumed to be carried out by other link layer mechanisms e.g. the standard learning process.

Throughout this draft, we expect the reader to be familiar with RSVP [1] and the current drafts for the Guaranteed- and the Controlled-Load service [3], [4] proposed in the Integrated Services IETF working group. The remainder of this document is organized as follows. Section 2 discusses the reservation- and the receiver model and lists the protocol properties of the LLRMP. We then describe in section 3 the protocol operation for reservation setup, reservation refresh, request mofification and tear down

within full-duplex switched and shared-medium bridged LANs. Section 4 discusses the relationship to RSVP. Appendix A contains example pseudo-code for the protocol operation and the status information to be held within hosts and bridges.

2. Protocol Properties

2.1 The LAN Model

We assume a LAN architecture which may have a pure shared, pure switched, or shared and switched topology. Heterogeneous technologies are assumed to be used for different segments of the bridged/ switched LAN. An example for a bridged LAN is illustrated in Figure 1. It consists of a root-switch and several workgroup-switches connecting shared-medium workgroup-segments to the switched backbone. A router and special hosts e.g. servers are connected directly to the root-switch. Since a pure switched network topology is a special case of a shared one, we mainly discuss the operation of the LLRMP protocol in a shared environment as the more general case.

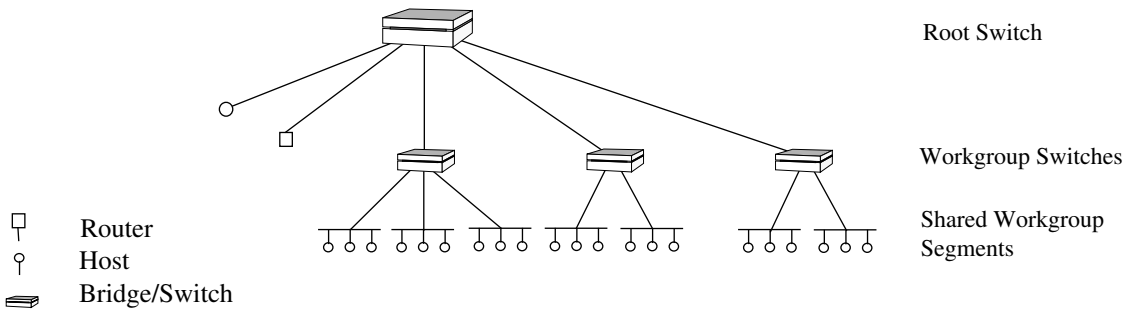

Figure 1:Example bridged LAN Topology.

During the following discussion, we assume that shared-medium segments and bridges/switches support advanced packet-scheduling mechanisms e.g. priorities or have an equivalent mechanism which can enforce the desired quality of service on a per-packet basis. Bridges and hosts are further assumed to be able to classify data packets and process them according to the service assigned to the corresponding session. A simple and efficient classification at the link layer might be based on the IEEE 802.1p traffic-class and the destination address. The actual algorithms used for packet classification and scheduling are independent of the LLRMP protocol and out of the scope of this paper. However the LLRMP needs to control the classification and scheduling process. Interactions occur e.g. when a new flow is admitted - if admission control is successful the LLRMP informs the local classifier and scheduler in order to enable the requested quality of service for the new flow.

2.2 The Reservation Model

The LLRMP performs a sender-based reservation setup. The current reservation model is *one pass*. Resources are reserved independently on each segment in the data path, on a hop-by-hop basis, as the

LLRMP reservation-message travels from the source to the receiver. Resources are requested at the node where the traffic enters the link.

We chose a sender-based reservation model because this ensures simplicity and good scaling. It further supports network-layer independence and preserves the ability of the LLRMP to serve different upper layer management protocols and to operate in different networking environments - not just IP. The protocol maintains a single interface to be used for requesting reservations, regardless of whether the request is for RSVP or non-RSVP flows. Furthermore, the model can more easily support measurement based admission control schemes since the existing reservation-messages travelling from the source to the receiver(s) can also be used to carry measured traffic characteristics or particular link or node-specific parameters used at the source node. No additional signalling is required for this case.

2.3 The Receiver Model

The simplest LLRMP reservation setup implies no receiver heterogeneity at all within a multicast group. This is intended to be the minimum requirement to be supported in hosts and bridges/switches.

The LLRMP protocol does not require a mechanism for receivers to signal a reservation request back to the data source since this is already provided by upper layer management protocols e.g. RSVP, and is therefore not required at link layer. By exploiting such upper layer signalling the LLRMP can support full RSVP receiver heterogeneity for all members of the multicast group. This is performed by an optional RSVP Snooping Module on the bridge. This module observes all RSVP control traffic forwarded through the switch and collects per-flow information which is then used by the LLRMP during the reservation setup. Details are discussed in section 3.11.

The separation between the LLRMP as a simple core protocol and the Snooping Module providing additional receiver heterogeneity allows the design of a wide range of switch solutions which can all operate with each other. Low cost switches would only support the LLRMP itself which would enable them to participate in the reservation setup. High-end switches could also support more sophisticated receiver-models up to full RSVP heterogeneity. All solutions can be incorporated in the same bridged/switched LAN. Note that the core LLRMP can be deployed on existing switches since it is simply a new link level protocol. This assumes that these switches have a processor.

2.4 Summary

The LLRMP uses the soft-state concept, as proposed in RSVP, for managing reservation state in hosts and bridges. Resources are reserved for simplex data streams. There is one important control message in the LLRMP protocol: the reservation-message. This message is used to create, modify, periodically refresh and tear down reservation state. In the absence of refreshes, the state automatically times out.

The LLRMP includes two different strategies for admitting and rejecting reservation requests. The first uses an optimistic approach which does not have an explicit local acknowledgment of reservation requests: all nodes running the protocol keep a local data base with information about existing reservations on the local network segment. After a node has successfully performed admission control for a

flow, it sends an LLRMP reservation-message and immediately enables the requested service for data packets of the admitted flow. Other nodes are informed about the reservation when they receive the reservation message. Possible reservation state inconsistencies on different nodes on the segment are resolved using a distributed reject mechanism whereby a service request can be rejected by any node on the same segment.

The second strategy is pessimistic. It uses an explicit acknowledgement-message sent by a single dedicated node on each segment in order to permit the request. The requested service at the source node is only enabled after the source node has received the acknowledgement-message. The dedicated node is called the *arbiter*. It performs the admission control for the local segment. The arbiter can be a bridge or a host and is automatically elected on each segment using a simple election mechanism. In contrast to the optimistic strategy, the pessimistic approach only requires the arbiter to perform the admission control. Other nodes do not have to manage a reservation data base.

Both reservation setup strategies represent a compromise between control packet overhead and probability of service violation. For shared-medium segments, we propose using the Optimistic Setup Strategy for services which do not provide hard guarantees e.g. the Controlled Load service, and the Pessimistic Strategy for a Guaranteed service. On full-duplex switched segments, the optimistic approach can be used for all types of service since the switch has complete control over the outgoing link and reservation state is not shared. A local acknowledgment-mechanism is not required in this case.

Note that the end-to-end reservation setup is optimistic. After resources are admitted on the first segment, the LLRMP will inform the upper layer e.g. RSVP that resources are available on the first hop. The admission control carried out on downstream segments within the data-path could still fail and the source might later receive a notification that resources on a particular segment or branch of the multicast distribution tree could not be allocated. There is no end-to-end acknowledgment mechanism.

The LLRMP protocol mechanisms are identical for hosts and bridges/switches. However since bridges are connected to multiple network segments they also have to make forwarding decisions for reservation-messages. Within bridges, the LLRMP could be implemented in a similar way to other link level protocols e.g. the 802.1D Spanning Tree protocol. Within hosts, we expect the LLRMP to be implemented e.g. as part of the device driver of a LAN adapter card.

Summarizing, the protocol properties of the LLRMP are:

a) The allocation concept is distributed, there is no central allocation-manager for the entire bridged LAN. The reservation setup can follow two different setup strategies.

b) The LLRMP protocol is simple. Resources are reserved in *one pass*. Soft-state is installed in hosts and bridges. Reservation state is created, modified, torn-down and refreshed by the source using the LLRMP reservation-message. Synchronization mechanisms between senders and receivers are assumed to be at a higher level than the link level (e.g. performed by RSVP or ST-II).

c) The LLRMP protocol is independent of the network layer resource management mechanisms.

d) Resource management is supported for unicast and multicast data. Distinct or shared reservations can be created depending on the reservation request and the ability of bridges to do policing and rate regulation.

e) The LLRMP protocol on its own does not support heterogeneous reservation-requests for receivers of the same multicast group. This keeps the core protocol simple and ensures that the LLRMP can be implemented on existing bridges. However, if the LLRMP is used in conjunction with the RSVP Snooping Module then full RSVP receiver heterogeneity can be supported.

f) LLRMP protocol state is only maintained along the data path between source and receiver and not in every bridge in the bridged network. This relies on MAC address entries in bridges/switches and assumes standard learning bridges.

g) Reservation requests may be gathered into large control packets in bridges to reduce the overall control traffic load.

h) The LLRMP protocol mechanisms are identical for end-nodes and bridges. Bridges additionally have to make forwarding decisions since they are connected to multiple segments.

i) The LLRMP can operate in a heterogeneous bridged environment, where only a few bridges support the new protocol. Parts of the network which do not support the LLRMP are treated as a single logical segment by the LLRMP.

3. Protocol Operation and Mechanisms

3.1 The LLRMP Message Types

The LLRMP contains five control messages which are used to manage and maintain a consistent reservation state on the segment. Note that some of the message types listed below are only used for reservation state notification when admission control has failed.

Resources at the link layer are reserved, refreshed, modified and torn-down using the *reservation*-message. If the setup uses the pessimistic strategy then any reservation request is acknowledged by the segment-arbiter. This uses an explicit *acknowledgment*-message. If the optimistic setup strategy (reservations without local acknowledgements) is used then the setup is only based on the reservation-message. No acknowledgement is returned in response to a new reservation-request. Any reservation-request that would exceed the resource limit on a *local* segement is rejected with a *reject*-message. Admission control failures from further down the data path (i.e. *not* on the local segement) are passed back to the source using *notify*-messages. *Query*-messages are periodically multicasted by the segment-arbiter to periodically force all nodes on a shared-medium segment to refresh the resources they have allocated.

3.1.1 The Reservation-Message

LLRMP reservation-messages always travel downstream along the data path from the source to the receiver. A single message may contain several reservation-requests (request). Each reservation-request contains the following elements:

request.*ack_flag*

If this flag is set then the source of the request still expects an acknowledgement from the segment-arbiter for this request. This flag is only used when the reservation setup uses the Pessimistic Setup strategy (reservations with local acknowledgements).

request.*flow_id*

The flow-identifier contains the session-identifier *session_id* and the flow's source-identifier *flow_src_id*. The *session_id* uniquely identifies the session, the *flow_src_id* identifies the source node and contains e.g. the MAC address of the source.

request.*QoS_request*

The quality of service request *QoS_request* contains the reservation-request for this flow. A reservation-request is described by a set of QoS parameters e.g. bandwidth, burst-size, reservation style, minimum and maximum packet size used, etc.

request.*filter_spec*

The filter specification *filter_spec* contains a filter e.g. a multicast MAC destination address. It is passed to the classifier at all switches along the data path. After installing the filter, switches will apply e.g. a particular output priority level to any packet that matches this filter.

3.1.2 The Acknowledgement-Message

Acknowledgements are only sent when the reservation setup uses the Pessimistic Setup strategy. They are always sent by the segment-arbiter. A single message may carry several acknowledgements (ack)'s. Each ack acknowledges a reservation-request made for a flow on the local segment. An ack consists of the flow-originator identifier *phop*, the flow-identifier *flow_id*, the quality-of-service request *QoS_request*, and the filter specification *filter_spec*.

ack.*phop*

The *phop* parameter identifies the flow-originator (the previous hop) on the local segment. It contains e.g. a MAC address of a bridge, host or router. The flow-originator is the node on the segment where the traffic enters the link.

ack.*flow_id*

The flow-identifier *flow_id* is identical to the parameter carried within the reservation-request: request.*flow_id*.

ack.*QoS_request*

The quality of service request *QoS_request* is identical to the parameter carried within the reservation-request: request.*QoS_request.*

ack.*filter_spec*

The filter specification *filter_spec* is identical to the parameter carried within the reservation-request: request.*filter_spec.*

3.1.3 The Reject-Message

A LLRMP reject-message may carry several rejects (reject). Each reject rejects a reservation-request made for a flow on the *local* segment and causes QoS parameters to be reset at hosts and switches (in contrast to the *notify* message below). A reject contains the flow-originator identifier *phop,* the flow-identifier *flow_id*, a set of quality-of-service parameters *QoS*, and an information field *info* carrying information to be returned to the user. Note that the reject message is only required on shared segments (it is not required in full-duplex switched environments).

reject.*phop*

The flow-originator (previous hop) *phop* is identical to the parameter carried within the acknowledgement-message: ack.*phop*.

reject.*flow_id*

The flow-identifier *flow_id* is identical to the parameter carried within the reservation-request: request.*flow_id*.

reject.*QoS*

The *QoS* field contains a set of quality-of-service parameters e.g. bandwidth, burst size, etc. It describes the resources which may be used by the rejected flow on the local segment.

reject.*info*

The *info* field carries information to be returned to the user. It contains e.g. the reject-code, the address of the rejecting segment-arbiter, etc.

3.1.4 The Notify-Message

Notify-messages inform the data source that sufficient resources could not be reserved on a particular segment further down the data path (i.e. *not* on the local segment). This is particularly for the case where admission control for a branch of a multicast tree failed, but the rest of the tree passed. Therefore the QoS parameters set for the rest of the tree can be retained. A single message may carry several notify-entries (notify) e.g. if a switch could not reserve resources on several outgoing segments. Each notify consists of the flow-originator identifier *phop,* the flow-identifier *flow_id*, and an information field *info* containing information to be returned to the user.

notify.*phop*

The flow-originator (previous hop) *phop* is identical to the parameter carried within the acknowledgement-message: ack.*phop*.

notify.*flow_id*

The flow-identifier *flow_id* is identical to the parameter carried within the reservation-request: request.*flow_id*.

notify.*info*

The *info* field carries information to be returned to the user. It contains e.g. the reject-code, the address of the rejecting segment-arbiter, etc.

3.1.5 The Query Message

The query-message has two functions. It first allows a new node (e.g. a bridge) to quickly learn the reservation state on a local segment because all nodes receiving a query will report their reservation state within a defined time interval *TD*. Secondly, query-messages are used to elect the segment-arbiter. Query-messages only carry the arbiter flag *arbiter_flag*.

query.*arbiter_flag*

If set, the *arbiter_flag* indicates that the source of the query-message requests to become the arbiter on the local segment. For details about the arbiter-election process, see section 3.6.

3.2 The Reservation Setup in a Full-Duplex Switched LAN

In order to clarify the fundamental LLRMP operation, we first consider the reservation setup in a full-duplex switched LAN. An implementation that only runs in a full-duplex switched environment can be greatly simplified. It does not have to support the segment-arbiter election process and the acknowledgement mechanism because the links between hosts and switches or between switches themselves are not shared. Each host/switch has complete control over the resources of the outgoing link. Note that this does not apply to half-duplex links.

Figure 2 illustrates the reservation setup from router *RT* to receiver *R1*. Router *RT* sends data packets to the multicast group *g1* which was joined by receiver *R1*. The setup starts when the LLRMP on *RT* receives a reservation-request from the upper layer e.g. RSVP. After receiving the request, the LLRMP first initiates admission control for the outgoing link to switch *SW1*. This is illustrated in step (1) within Figure 2. If the test fails then the reservation request is immediately rejected without any signalling on the network. If the admission control was successful, then the LLRMP sends a reservation-message to switch *SW1*as shown in step (2). The LLRMP on *RT* also informs its local classifier about the existence of the new flow. After the filter for the new flow has been installed, all data packets matching this filter are transmitted using the specified medium access mechanism e.g. a certain priority level.

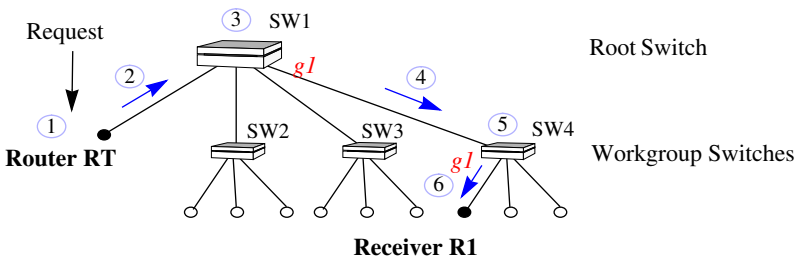

Figure 2: Example: Reservation Setup in a full-duplex switched LAN.

When switch *SW1* receives the reservation-message from *RT*, it first creates the reservation state for the new flow. All output-ports are then checked to see whether a reservation for the corresponding link must be made for this flow. This decision is based on a MAC address entry within the forwarding table of the switch. Since group address *g1* is only registered for the link to switch *SW4*, switch *SW1* only reserves resources for that link. It initiates admission control and if this test was successful, *SW1* for-

wards the reservation-message to switch *SW4*. This is shown in step (4). If however the admission control failed then *SW1* only returns a notify-message to router *RT* reporting the admission control failure on the link to *SW4*. Note that if group *g1* were registered for several output ports then *SW1* would try to reserve resources for these ports despite the fact that the request for the link to *SW4* was rejected.

If switch *SW4* receives the reservation-message then it carries out the same actions as described for switch *SW1*. The setup finishes when receiver *R1* receives the reservation-message reporting the QoS parameters e.g. the bandwidth, bridge-hop-count, etc. reserved along the data path: *RT - SW1 - SW4 - R1*. Note that the reservation setup in full-duplex switched LANs only uses reservation- and notify-messages. Note further that, joining an active multicast group does not necessarily trigger a new reservation-message at the source node. Instead, the reservation setup for the new mulicast receiver may be initiated at the nearest branch point (switch) of the reservation tree.

3.3 The Optimistic Setup Strategy - Reservations without local ACK-Messages

This approach allows a reservation setup across shared medium segments. It is based on the same fundamental mechanisms as discussed for the setup in a full-duplex switched LAN. However, additional mechanisms are required because of the shared character of the link. These are the segment-arbiter election process and mechanisms for performing a distributed admission control for each shared-medium segment. The latter are specific for the Optimistic Setup Strategy.

The reservation setup on a single shared-medium segment is illustrated in Figure 3. The Optimistic Setup Strategy implies that all nodes running the protocol keep a local data base with information about existing reservations on their local network segment. When the LLRMP on router *RT* receives the reservation request, it first performs admission control using its local reservation data base. If the test fails then the request is instantly rejected without any signalling on the network. If the admission control was successful, then the LLRMP on router *RT* updates the local data base and sends a reservation-message onto the local segment. It also informs the local classifier to enable the quality of service for the new flow. This is illustrated in steps (1), (2) and (3) within Figure 3.

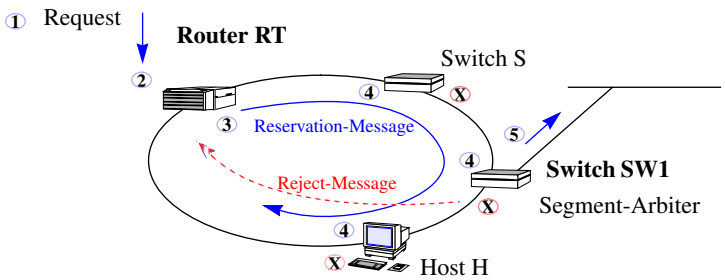

Figure 3: Reservation Setup without Acknowledgement-Messages.

The reservation request message is multicasted to all other LLRMP entities on the local segment using a high priority medium access mechanism, if available. The message is addressed to a well-known,

LLRMP specific multicast address. After receiving the request, each node (e.g. the switches *SW1*, *S* and host *H*) itself does admission control for the request using its own view of the total reserved resources and, if successful, updates its local data base. This is shown in step (4).

If the admission control fails then a node assembles a reject-message and schedules it for transmission within a random time interval between > 0 and *TD* seconds. This process happens on all nodes and is illustrated in step (X). However there is one node on the segment which is allowed to send its reject immediately. This is the elected *arbiter* of the shared-medium segment.

Whenever other nodes receive a reject from the segment-arbiter, they cancel the reject-message they have scheduled, if they have one. This mechanism ensures that requests on a segment are usually rejected by just one node: the arbiter. However other nodes set up a reject-message in case the arbiter fails or has not received the request. After receiving a reject from the network, router *RT* updates its data base and reports the error to the upper layer e.g. RSVP.

This finishes the reservation setup on the local segment. The setup process is repeated if a bridge decides to forwards the request onto an adjacent segment, as shown in step (5) within Figure 3. Note that bridges do not flood reservation-messages. They only forward request messages along the data path using the forwarding rules provided in section 3.8.

3.3.1 The Robustness of the Optimistic Setup Strategy

The optimistic approach can tolerate the loss of a single control message without any impact. Two or even more reservation-messages can go missing at several nodes (e.g. due to congestion) as long as there is at least one node on the segment which has received all the messages. This doesn't have to be the arbiter since all nodes are able to reject reservation requests. However, the probability that reservation-messages go missing is very small since control messages are transmitted over a single segment using a high priority medium access mechanism, if available. In bridges and switches, control messages are passed to the processor queue and not to a (possibly congested) link output queue. It should be emphasized that the quality of the link level service is only affected if reservation-messages go missing and over-link-limit reservations are made during inconsistent reservation state conditions. Further, it is very unlikely that network administrators will allow reservations up to the link capacity.

Inconsistencies of the reservation state are resolved by the refresh mechanism. Reservation-messages periodically update the status information on all nodes in the network. Any crash of the arbiter is resolved by the arbiter election process discussed in section 3.6.

3.4 The Pessimistic Setup Strategy - Reservations with local ACK-Messages

The Pessimistic Setup Strategy uses an explicit acknowledgement of reservation requests to prevent any possible service violation due to inconsistent reservation state on different nodes on the shared-medium segment. After invocation, the LLRMP multicasts a reservation-message onto the local segment as illustrated in step (1) in Figure 4. Unlike the optimistic approach, the local classifier at router

RT is not informed until the request was acknowledged by the segment-arbiter. The admission control for all reservation requests on the segment is carried out by the arbiter as shown in step (2). If the test fails then a reject message is returned to the originator of the request. After receiving this message, router *RT* informs the upper layer e.g. RSVP about the event.

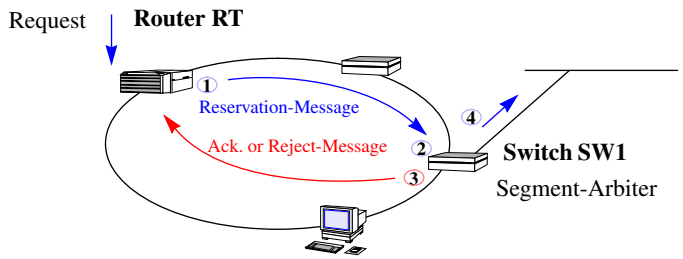

Figure 4: Reservation Setup using local Acknowledgement-Messages.

The arbiter returns an acknowledgement message to router *RT* if the request passed the admission control test. Upon receipt, *RT* updates its local classifier which enables the quality of service requested for the new flow. As in the previous approach, all bridges on the segment check whether the reservation request needs to be forwarded or not. Note that this test is always carried out after the request has been acknowledged on the input-segment. A bridge that is not the segment-arbiter performs this test after it has received the acknowledgement-message (or a subsequent acknowledged reservation-message).

Note that control messages are multicasted. By observing control messages, other nodes can automatically learn the existing reservation state on the local segment.

3.5 The Reservation Refresh Mechanism

The reservation state on all nodes is aged out if not refreshed. Each LLRMP node periodically refreshes the reservation information held about it at all other nodes on the segment by sending a reservation-message. All refreshes are synchronized using a simple query-mechanism. The arbiter periodically sends a query-message onto the local segment. Whenever a node receives a query, it replies with a reservation-message reporting the reservations made by itself. The message is scheduled within an arbitrary time interval of length *TD* to avoid congestion. Each node only reports its own flows.

3.6 The Dynamic Segment Arbiter Election

The query-mechanism allows a node to quickly learn the total reservation state on the local segment. It is also used to elect the arbiter. By default, the arbiter is the node with the lowest MAC address on the segment.

The election is carried out independently on each segment of the bridged LAN. If an LLRMP node comes up, then it first assumes that it is the segment-arbiter and thus queries the network by multicasting query-messages. After receiving a query with a lower MAC source address, a node will reschedule its own query-timer to be longer than the current value. If the new node is itself the new arbiter then its

query-messages will update the query timer on all other nodes, if not, then it will itself receive a query with a lower address. If the arbiter fails, the query-timer on other nodes runs out and finally another node becomes the arbiter.

Optionally a node can set the *arbiter flag* in its query-messages. This forces all nodes which do not have the *arbiter flag* set themselves to update their query-timer, even if they have a lower MAC address than the originator of the query-message. The *arbiter flag* allows a dedicated node e.g. a bridge or a network management station to become the arbiter of the segment regardless of its MAC address.

3.7 Request Modifications and Tear Down

LLRMP allows applications to dynamically change their reservation requests. As with new requests the availability of new resources is first checked by admission control. The new state is distributed using the normal reservation mechanism. Applications always keep their reservations while the LLRMP attempts to increase them. Reservation-messages are further used to release resources. In this case, the LLRMP simply sends a reservation-message with the new parameter set. If all resources for a flow are to be released then *NULL* values for all QoS-parameters are distributed with the reservation-message. This allows resources to be torn down quickly.

3.8 Control-Message-Forwarding within Bridges/Switches

The LLRMP protocol mechanisms are identical for end-nodes and bridges/switches. However bridges are connected to several network segments and therefore have to manage reservation state for each of the segments connected. The forwarding decision for control-messages is based on matching address information found in the reservation-message and in the MAC address table of bridges. For unicast, the rules are similar to the data-packet forwarding rules used within bridges:

1. A request does not need to be forwarded to any other segment if the MAC destination address is registered for the same port from which the reservation-message was received (Leaf Rule).
2. If the bridge has a table entry registered on a port which is different to the one from which the reservation-message was received, then it only forwards the request through that port (Direct Path Rule).
3. If there is no address entry in the MAC address table then the request is forwarded to all segments, except the one from which the reservation-message was received (Flood Rule).

Similar rules apply for multicast. However, optional input can be provided by the RSVP Snooping Module, if such a module exists on the switch. We assume a multicast routing mechanism as currently being standardized in IEEE 802.1p:

1. A reservation request does not need to be forwarded through a particular output-port if a group entry was found in the address table but the group is not registered for that output-port.
2. If the group is registered for an output-port then the reservation request is forwarded. However, this decision can be overruled by information from the Snooping Module on the bridge reporting that no quality of service was requested for the flow on this output-port. Note that the reservation request forwarded through any output-port will always be equal or lower than the request received from the input-port.

3. If there is no address entry in the address-table then the request is forwarded to all segments, except the one from which the reservation-message was received. As in the previous rule, the decision for each output-port can be overruled by information received from an existing Snooping Module.

The forwarding process ensures that reservation-messages are only forwarded along the data path between the sender and the receiver(s). This relies on MAC address entries within the address table of bridges/switches. For reservation requests from RSVP and STII, these entries are likely to be there because: (1) network layer control messages are periodically exchanged. These are *PATH* and *RESV* messages in RSVP and *HELLO* messages in ST-II. (2): applications with stringent service requirements are likely to be interactive, and will thus have a return control-channel and/or a return data-path.

The LLRMP still works when these address table entries are missing. In this case, reservation messages become forwarded to all segments and will reserve the requested resources on the entire LAN. This is equivalent to a network layer based allocation policy, where the link level structure of the network remains hidden. Eventually however, data packets will flow causing entries to be made in MAC address tables of bridges and old reservations to time out on unused segments.

3.9 Dynamic Topology Changes

The LLRMP automatically recovers from dynamic topology changes using the reservation refresh mechanism. Whenever the bridged topology changes, reservation refreshes are forwarded along the new data path and will immediately reserve any missing resources. Old reservation state in bridges will time out since it is not refreshed.

3.10 Heterogeneous bridged LANs with non LLRMP Clouds

The LLRMP can operate in a heterogeneous environment where only parts of the bridged LAN support the new protocol. Existing bridges and switches will flood LLRMP reservation messages since control messages are sent with a new link level protocol identifier (e.g. a new *ethertype*) and are addressed to an exclusively allocated multicast address. Non LLRMP clouds between two LLRMP nodes or bridges/switches are treated as one single logical segment for allocating resources.

3.11 More Receiver Heterogeneity - The RSVP Snooping Module

The RSVP Snooping Module is an optional optimization for supporting RSVP receiver heterogeneity. The module sits on the switch and observes all RSVP control traffic forwarded through the switch. It does not modify any RSVP messages during forwarding (although it might drop duplicate messages), and it also does not require any modifications to RSVP e.g. new frame formats, objects etc. While observing RSVP control traffic, the Snooping Module collects per-session, per-flow, or even per-receiver information. This information is used by the LLRMP when it determines the quality of service to be reserved for a particular session, flow or receiver on a particular output-port.

The level of RSVP receiver heterogeneity supported depends on the amount of information collected. For each flow using e.g. a Controlled Load service, the module maintains a data base entry similar to:

| *flow_id* | *service_id = Controlled_Load* | *list_of_input_ports* | *timer* |

where *list_of_input_ports* is a per-port bit-mask containing a flag for each switch-port. Each bit indicates whether an RSVP *RESV* message for flow *flow_id* was received from this port during the last timer interval. The parameter *timer* contains a simple timer which is used to let the entry time out if not refreshed. It is also a bit mask. More heterogeneity can be supported if the module collects per receiver information. A data base entry for a guaranteed service supporting full receiver heterogeneity could look like:

| *flow_id* | *service_id = Guaranteed* | *receiver_id* | *QoS request* | *input_port* | *timer* |

It is clear that the amount of information to be maintained represents a compromise between costs and level of receiver heterogeneity supported.

4. LLRMP Relationship to RSVP

RSVP is the reservation setup protocol designed for an Integrated Services Internet. Applications use the protocol to negotiate a specific quality of service with the network. After receiving a reservation request, RSVP carries the request through the internetwork and initiates resource-reservation on each of the intermediate links along the data path. During the RSVP reservation setup, a *PATH* message travels downstream from the source towards the receiver(s). It contains the traffic characteristic (*TSpec*) describing the traffic generated at the data source. RSVP *RESV* messages travel upstream from the receiver(s) towards the source. They are used by receiver(s) for signalling their service request (*FlowSpec*).

4.1 LLRMP Invocation

LLRMP reservation requests are made at the node where the traffic enters the bridged LAN. This node could be a host, a router or a gateway. RSVP makes the reservation request for a flow to the LLRMP after is has received a *PATH* and a *RESV* message for this flow, because this provides the required information (*TSpec, FlowSpec*) required for the request. This is illustrated in Figure 5.

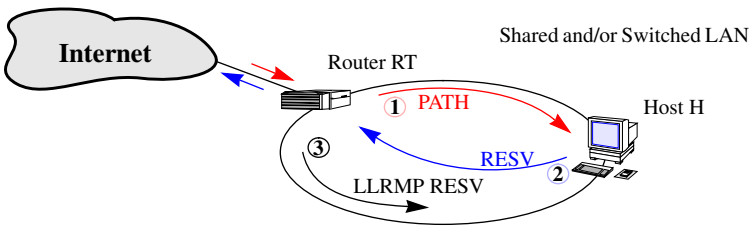

Figure 5: LLRMP Invocation from RSVP.

4.2 Interface

The interface between RSVP and LLRMP is for further study.

5. Functional Specification

The exact message formats are defined in a future draft.

6. References

[1] R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin, *Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification*, Internet-Draft draft-ietf-rsvp-spec-13.ps, August 1996.

[2] L. Delgrossi, L. Berger, *Internet Stream Protocol Version 2 (ST2), Protocol Specification - Version ST2+*, RFC 1819, August 1995.

[3] S. Shenker, C. Partridge, R. Guerin, *Specification of Guaranteed Quality of Service*, Internet Draft draft-ietf-intserv-guaranteed-svc-06.txt, August 1996.

[4] J. Wroclawski, Editor, *Specification of the Controlled-Load Network Element Service*, Internet Draft draft-ietf-intserv-ctrl-load-svc-03.txt, August 1996.

[5] *IEEE 802.1D, International Standard, Information technology - Telecommunications and information exchange between systems - Local area networks - Media access control (MAC) bridges*, ISO/IEC 10038:1993.

Authors Address

Peter Kim
Hewlett Packard Laboratories Bristol
Filton Road, Stoke Gifford
Bristol BS12 6QZ. U. K.
pk@hplb.hpl.hp.com
+44 117 922 8357

A. Appendix

This section contains example pseudo-code for the LLRMP protocol operation and the status information to be held within hosts and bridges. It is intended to provide a detailed overview of the algorithm. However real implementations may use additional or different data structures in order to e.g. optimize status-lookups and management. The pseudo-code uses the packet-formats described in section 3.1. Protocol details like the format verification, some error-cases and the details of the hashing mechanisms were omitted. Several possible optimizations and optional mechanisms were also left out. For the sake of clarity, we further included a summarizing overview for each algorithm.

A.1 Host and Switch Control Structures

A.1.1 Host Control Structures

Note that for the sake of brevity, this section only lists the parameters used within the example pseudo-code.

For each QoS flow, the host maintains a *flow-control-block f* containing the following information:

f.flow_id	-	flow-identifier containing the *session_id* and the *flow_src_id* (see request.*flow_id*).
f.ack_flag	-	acknowledgement-flag - e.g. one bit. If set it indicates that a reservation request is pending and has not yet been acknowledged by the arbiter on the segment.
f.phop	-	flow-originator on the local segment (see ack.*phop* in section 2.1.2).
f.timer	-	reservation-presence-timer. This timer is used to time out the state for the flow.
f.QoS	-	QoS parameters for this flow e.g. bandwidth, burst-size, reservation style, minimum- and maximum packet-size, etc.
f.dest	-	MAC destination address of the flow.

For each local interface, a host maintains an *interface-control-block i* which contains the following information (per-device-driver-state):

i.query_timer	-	query-timer.
i.query_count	-	query-retransmission-count.
i.refresh_timer	-	refresh-timer. This timer is used for responding to query-messages.
i.refresh_count	-	refresh-retransmission-count.
i.arbiter_flag	-	arbiter-flag (requires one bit). If set it indicates that this node requests to become the arbiter on the segment connected through interface *i*. The arbiter-flag is set by the network administrator. Bridges however might have the flag set by default.
i.arbiter_status	-	local segment arbiter-status-flag (requires one bit). If set, it indicates that the node is the arbiter for the segment connected to interface *i*.
i.my_id	-	identifier of the local interface *i* e.g. the MAC address of the interface.

A.1.2 Switch Control Structures

For each QoS flow *f,* the switch maintains a *flow-control-block f*. It contains the same parameters as used for the host-implementation. Note that the acknowledgment-flag *f*.ack_flag is maintained for each switch-port e.g. in a bit-mask (one bit per switch-port). Additional switch-specific parameters are:

f.input_port	-	input-port from which the switch received the reservation-message.
f.resv_flag	-	reservation-flag (requires one bit per port) indicating whether a reservation (and a corresponding *reservation-control-block r* is present for a particular output-port.
f.status_flag	-	status-flag (requires one bit per port) indicating whether a reservation status change

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

occurred on a particular port. It is e.g. set for *port* if a reject-message arrived on *port*.

f.ack_flag - acknowledgement-flag (requires one bit per port) indicating whether the reservation request has been acknowledged by the arbiter on a particular output-port or not.

f.fwd_mask - forwarding mask (requires one bit per port) indicating whether the reservation-message needs to be forwarded through that port or not.

Flow-control-blocks are only allocated on the input-port (the port from which the switch received the reservation-message). For each reservation made for a flow *f* on port *port* != *f*.input_port, a *reservation-control-block r* is maintained by the switch on *port*. In contains the following parameters:

r.f - back pointer to the corresponding flow-control-block *f*.

r.QoS - resources reserved for flow *f* on output-port *port*.

Note that *reservation-control-blocks* are only really needed for supporting a Guaranteed Service where different receivers request different resources e.g. a different delay bound. If a switch only supported resource reservation for a Controlled Load service then all reservation state for a flow could be maintained within its *flow-control-block*. The allocation and management of *reservation-control-blocks* for output-ports is not required in this case.

For each switch-port, a switch maintains a *port-control-block*. A *port-control-block* contains exactly the same parameters as an *interface-control-block* used within host implementations.

A.2 General Operation Overview

This section contains a general overview about the actions carried out by a host or switch when any of the events listed below occurs. Detailed pseudo-code for the actions can be found in section A.5.

A.2.1 The Pessimistic Setup Strategy - Reservations with local ACK-messages

The Host Algorithm

1. On startup of host-interface *i* (e.g. boot up time):
   1.1 Start the query-timer for *i*.
2. On expiration of the query-timer for interface *i*:
   2.1 Send a query-message through interface *i*. The host is the arbiter for segment *i*.
3. On receipt of a query-message from interface *i*:
   3.1 Determine if the sender of the query-message is the arbiter on segment *i*.
   - If this is the case then update the local arbiter-status and query-timer for interface *i*.
   3.2 Set the refresh-timer for interface *i* to trigger a reservation-message.
4. On expiration of the refresh-timer for interface *i*:
   4.1 Send a reservation-message to refresh our reservation state held e.g. at the arbiter or at bridges.
5. On receiving a reservation request (e.g. from RSVP) to reserve or modify the quality of service for a flow:
   5.1 If resources need to be reserved and the host is the arbiter for the outgoing segment (interface *i)* then perform admission control. If the request is rejected then inform the upper layer e.g. RSVP and return. If the request passed the admission-control then inform the local scheduler to enable the QoS admitted and send a reservation-message through interface *i*. Return from the function.
   5.2 If the switch is not the arbiter on the segment (interface *i*) but resources are required then send a reservation-message through interface *i* but do not enable the QoS. The segment-arbiter will

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

acknowledge the request.
5.3 If resources can be released then release them and send a reservation-message through interface *i*.

6. On receipt of a reservation-message from interface *i*:
   6.1 Drop the reservation-request if the host is not the arbiter on interface *i*. Return.
   6.2 If resources need to be reserved then perform admission control.
   - Depending on the result send either an acknowledgment-message or a reject-message back.
7. On receipt of a reject-message from interface *i:*
   7.1 If the reject is addressed to the host then inform the upper layer e.g. RSVP that the reservation request failed. Update the QoS and inform the local scheduler if required.
8. On receipt of an acknowledgement-message from interface *i*:
   8.1 If the acknowledgement is addressed to the host then inform the local scheduler to enable the QoS admitted for the flow on *i*.
9. On receipt of a notify-message from interface *i*:
   9.1 If the notify-message is addressed to the host then inform the upper layer e.g. RSVP that a switch could not reserve the requested resources on a particular output-port.
10. On expiration of the reservation-presence-timer for a flow:
    10.1 Delete any flow state maintained for this flow at the host.
11. On upper layer request (e.g. from RSVP) to tear down the resources for a flow:
    11.1 Send a reservation-message requesting QoS = 0 in order to tear down any downstream resources allocated for this flow. Inform the scheduler to disable the QoS for this flow. Delete any flow state.

The Switch Algorithm

1. On startup of switch-port *port* (e.g. switch boot up time, or if Spanning Tree enables a new port):
   1.1 Start the query-timer for *port*.
2. On expiration of the query-timer for switch-port *port*:
   2.1 Send a query-message through port *port*. The switch is the arbiter for segment *port*.
3. On receipt of a query-message from switch-port *port*:
   3.1 Determine if the sender of the query-message is the arbiter on segment *port*.
   - If this is the case then update the local arbiter-status and query-timer for port *port*.
   3.2 Set the refresh-timer for port *port* to trigger a reservation-message.
4. On expiration of the refresh-timer for switch-port *port*:
   4.1 Send a reservation-message to refresh our reservation state held e.g. at the arbiter or at other bridges.
5. On receipt of a reservation-message from switch-port *input_port*:
   5.1 Drop the reservation-request if the switch is not the arbiter on *input_port* and the reservation-request has the *ack_flag* set (requesting an acknowledgement from the segment arbiter).
   5.2 If the switch is the arbiter on segment *input_port* and if required then perform admission control.
   - Depending on the result send either an acknowledgment-message or a reject-message back.
   5.3 Check whether the reservation state for this flow has changed. Create/update the state maintained.
   - Do not instantly forward a reservation message if previous state was only refreshed.
   5.4 If state has changed then check what resources need to be reserved for each output port *output_port*.
   - If resources need to be reserved on *output_port* and the switch is the arbiter for *output_port* then perform admission control. If the request is rejected then send a notify-message back to the source.

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

If the request passed the admission-control then inform the local scheduler to enable the QoS admitted and send a reservation-message through *output_port*.
- If the switch is not the arbiter on segment *output_port* but resources are required then send a reservation-message through port *output_port*. The segment-arbiter will acknowledge the request.
- If resources can be released then release them and send a reservation-message through *output_port*.

6. On receipt of a reject-message from switch-port *input_port:*
   6.1 If the reject is addressed to the switch then send a notify-message to the previous hop (a bridge or the source-host) on the way back to the data source. The notify-message will inform the source that resources on segment *input_port* could not be reserved by the switch.
7. On receipt of an acknowledgement-message from switch-port *input_port*:
   7.1 If the acknowledgement is addressed to the switch and the switch has a reservation entry (a reservation-control-block) for this flow on *input_port* then inform the local scheduler to enable the QoS admitted on *input_port*. This finishes the processing of this acknowledgment.
   7.2 If the acknowledgment is not addressed to the switch then carry out actions 5.3 and 5.4.
8. On receipt of a notify-message from switch-port *input_port*:
   8.1 If the notify-message is addressed to the switch then forward the notify-message to the previous hop (a bridge or a host) on the way back to the data source.
9. When the GARP entity on the switch reports a membership change for multicast address *maddr*:
   9.1 Check all flows which have *maddr* as MAC destination address.
   - Perform action 5.4 for all output-ports.
10. When the RSVP Snooping Module reports a change of the QoS requested for a particular flow or session on a particular port *output_port*:
    10.1 Increase, decrease or tear down the reservation for this flow or session on port *output_port*.
    - Perform action 5.4 however only for port *output_port*.
11. On expiration of the reservation-presence-timer for a flow:
    11.1 Release any resources reserved for this flow on any output port. Tear down downstream resources by sending a reservation-message with the request QoS = 0. Delete any flow state maintained for this flow at the switch.
12. When a switch port goes down and e.g the Port Manager or the Spanning-Tree-entity reports that port *port* has gone down:
    12.1 Delete all reservations (reservation-control-blocks) registered on *port*.
    12.2 Delete the entire state maintained for flows which have *port* as their input-port.

A.3 Switch Functions

We assume that the LLRMP-entity on the switch has access to the local MAC-address-table via a common interface. Within the example pseudo-code we use the function: *port_mask = lookup_address_table (MAC_address)*, where *MAC_address* is a unicast or multicast MAC address to be looked up, *lookup_address_table()* is the lookup function and *port_mask* is a bit mask (one bit per switch-port) reporting the switch-ports on which *MAC_address* is registered. An address is registered for a port if the corresponding bit in *port_mask* is set. Whenever an address was not found during the lookup then *lookup_address_table()* returns *port_mask* with all switch-ports set (corresponding to the Flood-Forwarding rule).

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

A.4 Timer Values and Constants

The following constants are used within the algorithm:

RCNT	- retransmission count	(currently: RCNT = 2).
TQ	- query interval	(currently: 50 sec.).
TD	- maximum response delay time	(currently: 10 sec.).
TM	- minimum timer offset	(currently: 1 sec.).
TP	- reservation presence timeout	(currently: ((RCNT * TQ) + TD)).

A.5 Example Pseudo-Code

A.5.1 The Pessimistic Setup Strategy - Reservations with local ACK messages

The Host Algorithm

1. On startup of the local interface *i* (e.g. boot up time):

```
i.query_timer = TM                          /* Init query-timer for interface i. */
i.query_count = 2                           /* At startup: send two query messages. */
start_query_timer (i)                       /* Start query-timer for interface i. */
```

2. On expiration of the query-timer *i.query_timer* for interface *i*:

```
send_query_message (i)                      /* Send a query-message on interface i. */
set flag i.arbiter_status                   /* We are the segment-arbiter. */
i.query_count -= 1                          /* Decrement query retransmission count. */
if (i.query_count > 0) then
    i.query_timer = TM                      /* Fast query retransmission as e.g. at startup */
else
    i.query_timer = TQ                      /* Normal query interval. */
    i.query_count = 1                       /* Normal case: send just one query message. */
    i.refresh_timer = random (TM, TD)       /* Set our refresh-timer to: (TM < time < TD). */
    i.refresh_count = 1                     /* Just one refresh. */
end
```

3. On receipt of an LLRMP query-message (*query*) from interface *i*:

```
/* Check whether the querier and the local node have the same management priority. */
if ( ((query.arbiter_flag ==  i.arbiter_flag ) &&   /* Check management priority. */
     (query.src_id < i.my_id)) | |                  /* Querier has lower source id. */
     ((query.arbiter_flag is set) &&                /* Querier has higher management priority. */
      (i.arbiter_flag is not set)) ) then

     i.query_timer = (RCNT * TQ) + TM           /* Update our timer, so we do not send queries. */
     i.query_count = 1
     clear flag i.arbiter_status                /* We are not the segment-arbiter. */
 else
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
    if (i.arbiter_status is set) then          /* We are the arbiter for this segment. */
      /* Our query will stop the second query from query.src_id and tell him that we are the arbiter. */
       send_query_message (i)                  /* Send a query-message through interface i. */
    end
 end
 /* In response to any query-message, send a reservation refresh delayed by a random time. */
 i.refresh_timer = random (TM, TD)             /* Set the refresh-timer to: (TM < time < TD). */
 i.refresh_count = 1                           /* Just one refresh. */
```

4. On expiration of the refresh-timer *i.refresh_timer* for the local interface *i*:

```
for (all registered flows f which enter the segment through the local interface i) do:
     request = build_request_message (f, f.QoS, i)   /* Build a reservation-message. */
end
send_request_message (request, i)              /* Refresh state held for us at other hosts/bridges.*/
```

5. On receiving a reservation request (e.g. from RSVP) to reserve or modify the quality of service specified in the reservation request *QoS_request* for flow *flow_id* on the local interface *i*:

```
/* Lookup the local reservation data base for an existing flow-control-block f. */
if (there is a flow f on i such that f.flow_id == flow_id) then
   if (f.QoS == QoS_request) then              /* Check what resources are already allocated. */
      return                                   /* The QoS param's are already allocated: return.*/
   end
   if (f.ack_flag is set) then                 /* An existing reservation request is pending. */
      return                                   /* Return, inform e.g. RSVP about the event. */
   end
   if (f.QoS > QoS_request) then               /* User-request to decrease the QoS for flow f. */
      f.QoS = QoS_request                      /* Enter the QoS to be reserved on segment i. */
      request = build_request_message (f, f.QoS, i) /* Build a reservation-message. */
      send_request_message (request, i)        /* Send the reservation-message through i. */
      inform_scheduler (f, f.QoS, i)           /* Inform the local classifier and scheduler. */
      return                                   /* Nothing more to do: return from this function. */
    end
 end
/* This is either a request for a new flow, or we have an existing flow f with f.QoS < QoS_request. */
if (flag i.arbiter_status is set) then         /* We are the arbiter for this segment (i). */
   result = admission_control (f, flow_id, QoS_request, i)   /* Perform admission control. */
   if (result == REJECTED) then                /* The reservation request was rejected. */
      inform the user that the request failed  /* Inform e.g. RSVP. */
      return                                   /* Nothing more to do: return from this function. */
   end
end
if (f is an existing entry) then               /* There is existing state for flow f. */
   f.QoS = QoS_request                         /* Enter the QoS to be reserved on segment i. */
else
   f = create_flow_state (flow_id, QoS_request, i)  /* Create state for flow f ; f.QoS = QoS_request. */
end
if (flag i.arbiter_status is set) then         /* We are the arbiter and did admission control. */
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
    clear flag f.ack_flag                          /* This request does not need to be acknowledged. */
    inform_scheduler (f, f.QoS, i)                 /* Inform the scheduler about the QoS for flow f. */
  else                                             /* We are not the arbiter for this segment. */
    set flag f.ack_flag                            /* Request an ack-message from the arbiter. */
  end
  request = build_request_message (f, f.QoS, i)    /* Note: request.ack_flag carries f.ack_flag. */
  send_request_message (request, i)                /* Send the reservation-message through i. */
```

6. On receipt of an LLRMP *reservation-message* from interface *i*:

```
  if (flag i.arbiter_status is not set)            /* We are not arbiter for this segment (i). */
                                                   /* Optionally: learn the reservation state. */
    return                                         /* Nothing to do: drop the message and return. */
  end
  /* We are the arbiter for the segment connected through interface i. */
  for (each reservation request (request) within the reservation-message received) do
    /* Lookup the local reservation data base for an existing flow-control-block f.*/
    If (there is a flow f on i such that f.flow_id == request.flow_id) then
      f.timer = TP                                 /* Refresh the reservation-presence-timer. */
      if (f.QoS > request.QoS_request) then        /* Request to decrease existing resources. */
        f.QoS = request.QoS_request                /* Enter the new QoS parameters for flow f. */
      end
      if (f.QoS == request.QoS_request) then       /* The required resources are already allocated. */
        /* Check whether the source has already received an acknowledgement-message (ack). */
        if (flag request.ack_flag is set) then     /* Source requests an ack-message. */
          ack = build_ack_message (f, f.QoS, i)    /* Build an acknowledgement-message. */
        end
        continue_with_next_request                 /* Nothing more to do: continue - next request. */
      end
    end
    /* This is either a request for a new flow, or a request for an existing */
    /*        flow f with f.QoS < request.QoS_request (request to increase the reservation). */
    result = admission_control (f, request.flow_id, request.QoS_request, i)  /* Admission control.*/
    if (result == REJECTED) then                   /* The request passed the admission control. */
      if (f is an existing flow) then
        reject = build_reject_message (f.flow_id, f.QoS, i) /* Keep existing resources:QoS = f.QoS.*/
      else
        reject = build_reject_message (request.flow_id, 0, i)  /* This is a new request:QoS = 0. */
      end
    else                                           /* The admission control was passed. */
      if (f is an existing flow) then
        f.QoS = request.QoS_request                /* Enter the QoS param's admitted for segment i. */
      else
        /* Create state for the new flow f ; f.QoS = request.QoS_request. */
        f = create_flow_state (request.flow_id, request.QoS_request, i)
        f.timer = TP                               /* Set reservation presence timer. */
      end
      ack = build_ack_message (f, f.QoS, i)        /* Build an ack-message to be sent through i. */
    end
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
    end
    /* We have finished the processing of all requests within the reservation-message received. */
    if (a reject-message (reject) was assembled for interface i) then
        send_reject_message (reject, i)              /* Send the reject-message through interface i.*/
    end
    if (an acknowledgement-message (ack) was assembled for interface i) then
        send_ack_message (ack, i)                    /* Send the ack-message through interface i. */
    end
```

7. On receipt of an LLRMP *reject-message* from interface *i*:

```
    for (each reject (reject) within the reject-message received) do
        if (reject.phop == i.my_id) then             /* The reject is addressed to us. */
            /* Lookup the local reservation data base for an existing flow-control-block f.*/
            if (there is a flow f on i such that f.flow_id == reject.flow_id) then
                if ((f.phop == i.my_id) &&           /* We are the originator of flow f on the segment. */
                    (f.QoS > reject.QoS)) then       /* Flow f wanted more than available. */
                    f.QoS = reject.QoS               /* Enter QoS param's from the arbiter. */
                    clear flag f.ack_flag            /* This is a reply: we don't need another one. */
                    inform_scheduler (f, f.QoS, i)   /* Inform the scheduler about the QoS for f. */
                    inform the user that the request failed  /* Inform e.g. RSVP. */
                end
            end
        else                                         /* The reject-message is not for us. */
                                                     /* Optionally: learn the reservation state. */
        end
    end
```

8. On receipt of an LLRMP *acknowledgement-message* from interface *i*:

```
    for (each acknowledgement (ack) within the acknowledgement-message received) do
        if (ack.phop == i.my_id) then                /* The acknowledgement is addressed to us. */
            /* Lookup the local reservation data base for an existing flow-control-block f.*/
            if (there is a flow f on i such that f.flow_id == ack.flow_id) then
                if ((f.phop == i.my_id) &&           /* We are the originator of flow f on the segment. */
                    (f.QoS == ack.QoS_request)) then /* The request is acknowledged. */
                    clear flag f.ack_flag            /* No further ack-message is required. */
                    inform_scheduler (f, f.QoS, i)   /* Inform the scheduler about the QoS for flow f. */
                end
            end
        else                                         /* The acknowledgement-message is not for us. */
                                                     /* Optionally: learn the reservation state. */
        end
    end
```

9. On receipt of an LLRMP *notify-message* from the local interface *i*:

```
    for (each entry (notify) within the notify-message received) do
        if (notify.phop == i.my_id) then             /* This notify is addressed to us. */
            if (there is a flow f on i such that f.flow_id == notify.flow_id) then    /* Lookup the data-base. */
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
            if (f.phop == i.my_id) then          /* We are the originator of f on this segment (i). */
               inform the user about the event   /* Inform e.g. RSVP. Pass notify.info. */
            end
         end
      end
   end
```

10. On expiration of the presence-timer *f.timer* for flow *f* registered on interface *i:*

```
delete_flow_state (f, i)                         /* Delete any state for flow f. */
```

11. On upper layer request (e.g. from RSVP) to tear-town the resources for flow *flow_id* on interface *i*.

```
if (there is a flow f on i such that f.flow_id == flow_id) then   /* Lookup the local data base. */
   clear flag f.ack_flag                         /* No acknowledgment required for the tear-down.*/
   inform_scheduler (f, 0, i)                    /* Inform the scheduler: QoS = 0 for flow f. */
   request = build_request_message (f, 0, i)     /* Build a reservation-message: QoS = 0. */
   send_request_message (request, i)             /* Tear-down the resources along the data path.*/
   delete_flow_state (f, i)                      /* Delete any state for flow f. */
end
```

The Switch Algorithm

There are three special functions used within the example pseudo-code. The function: *lookup_and_reserve()* looks up the MAC-address-table and initiates the resource reservation for the required output-ports. It then either calls function *reserve_resources_on_port()* or function *delete_resources_on_port()*. Function *reserve_resources_on_port()* checks whether and what resources need to be reserved for the flow on a particular output-port. If required, it reserves (or modifies) them, updates the reservation state and informs the scheduler. The other function: *delete_resources_on_port()* deletes any reservation state held for the flow on a particular output-port. Pseudo-code for all three functions is provided at the end of this section.

1. On startup of switch-port *port* (e.g. boot up time of the switch):

```
port.query_timer = TM                            /* Init query-timer for port port. */
port.query_count = 2                             /* At startup: send two queries through port port. */
start_query_timer (port)                         /* Start query timer for port port. */
```

2. On expiration of the query-timer *port.query_timer* for switch-port *port*:

```
send_query_message (port)                        /* Send a query-message through port port. */
set flag port.arbiter_status                     /* We are the arbiter for segment port. */
port.query_count -= 1                            /* Decrement the query retransmission count. */
if (port.query_count > 0) then
   port.query_timer = TM                         /* Fast query retransmission as e.g. at startup. */
else
   port.query_timer = TQ                         /* Normal query interval. */
   port.query_count = 1                          /* Normal case: send just one query-message. */
   port.refresh_timer = random (TM, TD)          /* Set our refresh-timer to: (TM < time < TD). */
   port.refresh_count = 1                        /* Just one refresh. */
end
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

3. On receipt of the query-message (*query*) from switch-port *input_port*:
   /* Check whether the querier and the local node have the same management priority. */
   if (((*query*.arbiter_flag == *input_port*.arbiter_flag) && /* Check the management priority. */
   (*query*.src_id < *input_port*.my_id)) || /* The querier has a lower source-id. */
   ((*query*.arbiter_flag is set) && /* The querier has a higher management priority. */
   (*input_port*.arbiter_flag is not set))) then

   input_port.query_timer = (RCNT * TQ) + TM /* **Update our timer, so we do not send queries. */**
   input_port.query_count = 1
   clear flag *input_port*.arbiter_status /* We are not the arbiter for segment *input_port*. */
   else
   if (*input_port*.arbiter_status is set) then /* We are the arbiter for this segment. */
   /* Our query will stop the second query from *query*.src_id and tell him that we are the arbiter. */
   send_query_message (*input_port*) /* Send a query-message through *input_port*. */
   end
   end
   /* In response to any query-message, send a reservation refresh delayed by a random time. */
   input_port.refresh_timer = random (TM, TD) /* Set the refresh-timer to: (TM < *time* < TD). */
   input_port.refresh_count = 1 /* Just one refresh. */

4. On expiration of the refresh-timer *port.refresh_timer* for switch-port *port*:
   for (all *reservation-control-blocks r* registered on port *port*) do
   request = build_request_message (*r*.f, *r*.QoS, *port*) /* Build a reservation-message on *port*.*/
   end
   send_request_message (request, *port*) **/* Refresh state e.g. held for *f* at other bridges. */**

5. On receipt of an LLRMP *reservation-message* from switch-port *input_port*:
   for (each reservation request (*request)* within the *reservation-message* received) do
   /* Lookup the local reservation data base for an existing flow-control-block *f*. */
   if (there is a flow *f* such that *f*.flow_id == *request*.flow_id) then
   f.timer = TP /* Refresh the reservation-presence-timer. */
   /* Check whether the topology changed and the source moved to another input-port. */
   if (*f*.input_port != *input_port*) then /* Topology change: source changed input-port.*/
   if (flag *f*.resv_flag is set for *input_port*) then /* There is an existing reservation on *input_port*.*/
   clear flag *f*.resv_flag for port *input_port* /* Keep record where reservations are. */
   delete_resv_state (*f, input_port*) /* Remove the reservation *r* for *f* on *input_port*. */
   inform_scheduler (*f,* 0, *input_port*) /* Inform the scheduler:QoS = 0 on *input_port* .*/
   end
   move entry *f* into hash table for *input_port* /* Rehash table entry for flow *f*. */
   clear flag *f*.ack_flag for port *input_port* /* Clean up the acknowledgment-state. */
   set flag *f*.status_flag for port *input_port* **/* Keep record of the topology change. */**
   f.input_port = *input_port* /* Enter the new input-port. */
   set flag *request*.ack_flag /* Treat this request like one not ack'ed.*/
   f.QoS = 0 /* Clear the QoS state for segment *input_port*. */
   end
   end

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
    /* Don't process this request if it has not been acknowledged and we are not the arbiter. */
    if ((flag request.ack_flag is set) &&                 /* This request has not yet been acknowledged. */
        (flag input_port.arbiter_status is not set)) then  /* We are not the arbiter on segment input_port. */
        /* The next acknowledgement-message will trigger the reservation on other output segments. */
        continue_with_next_request                        /* Nothing more to do: continue. */
    end
    /* If required, perform admission control for the input segment (f.input_port).  */
    if (flag input_port.arbiter_status is set) then       /* We are the arbiter for this segment. */
        if ((f is a non existing flow) ||                 /* This is a reservation request for a new flow. */
            ((f is an existing flow) &&                   /* This is reservation request to increase the  */
             (f.QoS < request.QoS_request)) then          /*          resources for an existing flow f. */
             result = admission_control (f, request.flow_id, request.QoS_request, input_port)
             if (result == REJECTED) then                 /* The reservation request is rejected. */
                 if (f is an existing flow) then          /* There are existing resources reserved for f. */
                     reject = build_reject_message (f.flow_id, f.QoS, input_port)   /* Keep existing f.QoS. */
                 else                                     /* This is a request for a new flow. */
                     reject = build_reject_message (f.flow_id, 0, input_port)       /* New flow: QoS = 0. */
                 end
                 continue_with_next_request               /* Nothing more to do: continue - next request. */
             end                                          /* The request passed the admission control. */
             ack = build_ack_message(f, request.QoS_request,input_port) /* Build an ack-message.*/
        else                                              /* The required resources are already allocated. */
            if (flag request.ack_flag is set) then
               /* The source has not yet received an ack: send another acknowledgement-message. */
               ack = build_ack_message (f, request.QoS_request, input_port)  /* Build an ack-message. */
            end
        end
    end  /* if arbiter_status is set. */
    /* The reservation request is acknowledged on the input segment: f.input_port. */
    if (f is a non existing flow) then                    /* There is no flow-control-block for flow f. */
        f = create_flow_state (request.flow_id, 0, input_port) /* Create state for flow f but : f.QoS =  0. */
        f.input_port = input_port                         /* Enter the input-port. */
        f.timer = TP                                      /* Set the reservation-presence-timer. */
    end                                                   /* State for flow f existed. */
    /* We finished the processing of the request request for the input-segment f.input_port. */
    /* Look up the MAC address table. If required initiate the reservation for an output-port. */
    lookup_and_reserve (f, request.QoS_request, input_port) /* See the pseudo-code for this function. */
end  /* for each request. */
/* We have finished the processing of all requests within the reservation-message received. */
for (any active switch-port port) do
    if (a reservation-message (request) was assembled on port port) then
        send_reservation_message (request, port)          /* Send the reservation message through port.*/
    end
    if (a reject-message was assembled on port port) then
        send_reject_message (reject, port)                /* Send the reject-message through port. */
    end
    if (a notify-message was assembled on port port) then
        send_notify_message (notify, port)                /* Send the notify-message through port. */
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
        end
    end
```

6. On receipt of an LLRMP *reject-message* from switch-port *input_port:*

```
for (each reject-entry (reject) within the reject-message received) do
    if (reject.phop == input_port.my_id) then          /* This reject is addressed to us. */
      /* Lookup the data base to find an existing reservation-control-block r for flow f. */
      if (there is a reservation entry r on input_port such that r.f.flow_id == reject.flow_id) then
        /* We are the originator of flow f on this segment (input_port). */
        if (r.QoS > reject.QoS) then                   /* f wanted more than available on input_port. */
          r.QoS = reject.QoS                           /* Enter the QoS param's from the arbiter. */
          set r.f.status_flag for port input_port      /* Keep record that the request was rejected. */
          clear r.f.ack_flag for port input_port       /* This is a reply: we don't need another one.*/
          inform_scheduler (r.f, r.QoS, input_port)    /* Inform the scheduler about the QoS for f. */
          /* Inform the data source (flow_src_id) that the request on segment input_port failed.*/
          notify = build_notify_message (r.f, reject.info, r.f.input_port)
        end
      end
    else                                               /* The reject is not addressed to us. */
                                                       /* Optionally: learn the reservation state. */
    end
end
/* We have processed all rejects within the reject-message: check if a message needs to be sent.*/
for (any active switch-port port) do
    if (a notify-message was assembled on port port) then
      send_notify_message (notify, port)               /* Send the notify-message through port. */
    end
end
```

7. On receipt of an LLRMP *acknowledgement-message* from switch-port *input_port*:

```
for (each acknowledgment (ack) within the acknowledgement-message received) do
    if (ack.phop == input_port.my_id) then             /* The acknowledgement is addressed to us. */
      /* Lookup the local reservation data base for an existing reservation-control-block r. */
      if (there is a reservation entry r on input_port such that r.f.flow_id == ack.flow_id) then
        /* We are the originator of flow f on the segment (input_port). */
        if ((r.f.ack_flag is set for input_port) &&     /* The request has not yet been ack'ed. */
            (r.QoS == ack.QoS_request)) then           /* This ack acknowledges the request. */
            clear r.f.ack_flag for port input_port     /* No further ack-message is required. */
            inform_scheduler (r.f, r.QoS, input_port)  /* Inform the scheduler about the new QoS. */
        end
      end
      continue_with_next_ack                           /* Nothing more to do: continue - next ack. */
    end
    /* We are not the originator of f on segment input_port: try to find a flow-control-block f.*/
    if (there is a flow f on input_port such that f.flow_id == ack.flow_id) then
      /* If the topology changed then ignore the ack: the next refresh will trigger a state-move. */
      if (f.input_port != input_port) then             /* The source moved to another input-port. */
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
            continue_with_next_ack                  /* Nothing more to do: continue. */
          end
        else                                        /* No existing flow-control-block for this flow. */
          f = create_flow_state (ack.flow_id, 0, input_port)   /* Create state for flow f but : f.QoS = 0. */
          f.input_port = input_port                 /* Enter the input-port. */
          f.timer = TP                              /* Set the reservation-presence-timer. */
        end
        /* Look up the MAC address table. If required initiate the reservation for an output-port. */
        lookup_and_reserve (f, ack.QoS_request, input_port) /* See the pseudo-code for this function. */
    end   /* for (all ack's). */
    /* We have finished the processing of all ack's within the acknowledgement-message received. */
    for (any active switch-port port) do
        if (a reservation-message (request) was assembled on port port) then
            send_reservation_message (request, port)    /* Send the reservation-message. */
        end
        if (a notify-message was assembled on port port) then
            send_notify_message (notify, port)          /* Send the notify-message through port. */
        end
    end
```

8. On receipt of an LLRMP *notify-message* from switch-port *input_port*:

```
for (each entry (notify) within the notify-message received) do
    if (notify.phop == input_port.my_id) then       /* This notify is addressed to us. */
        /* Lookup the data base to find an existing reservation-control-block r for flow f. */
        if (there is a reservation entry r on port input_port such that r.f.flow_id == notify.flow_id) then
            notify = build_notify_message (r.f, notify.info, r.f.input_port)
        end
    end
end     /* We processed all entries within the notify-message. */
for (any active switch-port port) do
    if (a notify-message was assembled on port port) then
        send_notify_message (notify, port)          /* Send the notify-message through port. */
    end
end
```

9. This sequence is triggered when the switch learns a new multicast MAC address or updates an existing multicast address entry. In this case, the GARP entity on the switch sends a notification to the LLRMP containing the multicast address *maddr* and the corresponding port registration mask *port_mask*:

```
/* Lookup the data base: find all flows which use address maddr as MAC destination address. */
for (all flows f with f.dest == maddr) do           /* Flow has destination maddr. */
    f.fwd_mask = port_mask                          /* Enter the new forwarding mask for this flow. */
    clear f.input_port in f.fwd_mask                /* Never forward through the input-port. */
    for (any active switch-port port) do
        if (port port is set in f.fwd_mask) then    /* There is a receiver registered on port. */
            if (f.resv_flag for port port is not set) then  /* There is no existing reservation for f on port. */
                reserve_resources_on_port (f, port) /* Reserve resources for flow f on port port. */
            end
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
            else                                        /* There is no receiver registered on port. */
              if (f.resv_flag for port port is set) then    /* There is a reservation for f on port port. */
                delete_resources_on_port (f, port)      /* Delete resources for f on port port. */
              end
            end
        end
    end
    /* We finished the processing of all flows that use maddr as MAC destination address. */
    for (any active switch-port port) do
        if (a reservation-message (request) was assembled on port port) then
            send_reservation_message (request, port)    /* Send the reservation-message.*/
        end
        if (a notify-message was assembled on port port) then
            send_notify_message (notify, port)          /* Send the notify-message through port. */
        end
    end
```

10. This sequence is triggered when the RSVP Snooping Module detects a change of the QoS requested for a flow. In this case, this module sends a notification to the LLRMP indicating the session identifier *session_id* and the port *port* where the change occured:

```
for (any flow f with f.flow_id.session_id == session_id) do
    if (port port != f.input_port) then
        /* Check what resources need to be reserved/modified/released on port port. */
        reserve_resources_on_port (f, port)
    end
end
/* We finished the processing of all flows f with f.flow_id.session_id == session_id. */
for (any active switch-port port) do
    if (a reservation-message (request) was assembled on port port) then
        send_reservation_message (request, port)    /* Send the reservation-message. */
    end
    if (a notify-message was assembled on port port) then
        send_notify_message (notify, port)          /* Send the notify-message through port. */
    end
end
```

11. On expiration of the reservation-presence-timer *f.timer* for flow *f* registered on switch-port *port:*

```
for (any port output_port whose flag f.resv_flag is set) do
    /* Delete any resources reserved for flow f on output_port and send a tear-down. */
    delete_resources_on_port (f, output_port)           /* Delete the reservation-control-block. */
    if (a reservation-message (request) was assembled on port output_port) then
        send_reservation_message (request, output_port) /* Tear-down any downstream resources.*/
    end
end
delete_flow_state (f, port)                             /* Delete any state maintained for flow f. */
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

12. This sequence is triggered when a switch port goes down. In this case, the Spanning Tree entity (or the Port Manager) on the switch sends a notification to the LLRMP reporting that port *port* has gone down:

```
/* Remove all flow-control-blocks f registered on port. Remove all resources for f on other ports. */
for (all flow-control-blocks f registered for port port) do
    for (any port output_port whose flag f.resv_flag is set) do
        delete_resv_state (f, output_port)              /* Delete reservation state r on output_port. */
    end
    delete_flow_state (f, port)                         /* Delete any state maintained for flow f. */
end
/* Remove all reservation-control-blocks r registered on port. Keep resources for f on other ports.*/
for (all reservation entries r registered for port port) do
    clear r.f.resv_flag for port port                   /* Keep record where reservations are. */
    clear r.f.status_flag for port port                 /* Clean up state for port port. */
    clear r.f.ack_flag for port port                    /* Clean up state for port port. */
    delete_resv_state (r.f, port)                       /* Delete reservation-state r for f on port. */
end
```

```
/* ======= Functions  ======== */
```

13. Function *lookup_and_reserve ()*:

```
/* This functions looks up the MAC address table. If required it initiates the reservation. */
lookup_and_reserve (f, QoS, input_port) begin
    /* Lookup the flow's destination address in the MAC address table of the switch. */
    port_mask = lookup_address_table (f.dest)       /* Get the forwarding mask. */
    clear port input_port in port_mask              /* Never forward through the input-port. */
    /* Check whether any QoS parameter or the forwarding mask has changed. */
    if ((f.QoS != QoS) || (f.fwd_mask != port_mask)) then /* The QoS parameters have changed. */
        set f.status_flag for port input_port       /* Keep record of the change. */
    end
    f.QoS = QoS                                     /* Enter the QoS for the input-segment. */
    f.fwd_mask = port_mask                          /* Enter the new forwarding mask. */
    /* Do not forward a reservation-message which has only refreshed previous state. */
    if (flag f.status_flag is cleared for all switch-ports) then/* No parameter has changed in the past. */
        return                                      /* Nothing more to do here: return. */
    end
    /* For each output-port port, check what reservations are required. Build the request. */
    for (any active switch-port port) do
        if ((f.status_flag for input_port is set) ||    /* Only process a port, if a parameter- */
            (f.status_flag for port is set)) then       /*        or status-change occured. */
            if (port port is set in f.fwd_mask) then    /* There is a receiver registered on port. */
                reserve_resources_on_port (f, port)     /* Reserve resources for f on port port.*/
            else                                        /* There is no receiver registered on port.*/
                if (f.resv_mask for port is set) then   /* There is an old reservation for f on port. */
                    delete_resources_on_port (f, port)  /* Delete resources for flow f on port port. */
                end
            end
        end
    end
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
    clear flag f.status_flag for port input_port          /* We checked all ports:clear the input-port.*/
  end
```

14. Function *reserve_resources_on_port()*:

```
  /* This functions checks what resources need to be reserved for flow f on port output_port. */
  reserve_resources_on_port (f, output_port) begin
    clear f.status_flag for port output_port          /* Since we now check the state on output_port.*/
    /* Determine the QoS to be allocated for flow f on port output_port. */
    if (RSVP_Snooping is supported) then              /* Optional: RSVP Snooping  support. */
      QoS = get_Snooping_entry (f, output_port)       /* Get the required QoS for f.flow_id,output_port. */
    else                                              /* No RSVP Snooping support. */
      QoS = f.QoS                                     /* Same QoS as on the input-segment. */
    end
    if (f.resv_flag for output_port is set) then      /* There is already a reservation on output_port.*/
      if (r.QoS > QoS) then                           /* Request for decreasing the resources. */
        r.QoS = QoS                                   /* Enter the new QoS parameters. */
        if (r.QoS == 0) then                          /* This is a reservation tear-down. */
          clear f.ack_flag for output_port            /* We don't need an ack-message for a tear-down. */
        end
        request = build_request_message(f, r.QoS, output_port)   /* Build a reservation-message.*/
        inform_scheduler (f, r.QoS, output_port)      /* Inform the scheduler about the QoS for flow f. */
      end
      if (r.QoS == QoS) then                          /* The required resources are already reserved. */
        if (r.QoS == 0) then                          /* No resources required on this port: delete state. */
          clear f.resv_flag for output_port           /* Keep record where the reservations for f are. */
          delete_resv_state (f, output_port)          /* Delete reservation state r for f on output_port. */
        end
        return                                        /* Nothing more to do: return from this function.  */
      end
    end
    /* This is either a request for a new reservation or for an existing one with r.QoS < QoS. */
    if (QoS != 0) then                                /* Resources need to be reserved on output_port.*/
      if (flag output_port.arbiter_status is set) then /* We are the arbiter for this segment. */
        result = admission_control (f, f.flow_id, QoS, output_port)   /* Perform admission control. */
        if (result == REJECTED) then                  /* The reservation request was rejected. */
          notify = build_notify_message (f, result, f.input_port)     /* Build a notify-message. */
          set f.status_flag for port output_port      /* Keep record that the request was rejected. */
          return                                      /* Nothing more to do: return from this function.  */
        end
        clear f.ack_flag for output_port              /* We are the arbiter: we don't need an ack. */
      else                                            /* We are not the arbiter for this segment. */
        if (f.ack_flag for output_port is set) then   /* There is a request pending (not yet ack'ed). */
          set f.status_flag for port output_port      /* Keep record that we couldn't reserve resources. */
          return                                      /* Nothing more to do: return from this function.  */
        end
        /* We are not the arbiter on segment output_port: we need an acknowledgment-message. */
        set f.ack_flag for port output_port           /* Request an ack-message from the arbiter. */
      end
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.

```
        if (f.resv_flag for output_port is set) then     /* There is reservation state on output_port. */
           r.QoS = QoS                                   /* Enter the QoS for flow f on output_port. */
        else
           r = create_resv_state (f, QoS, output_port)   /* Create reservation state for f on output_port.*/
           set f.resv_flag for port output_port          /* Keep track of reservation state created. */
        end
        /* Build a reservation-message. If we are the segment-arbiter then inform the scheduler. */
        request = build_request_message (f, r.QoS, output_port)  /* Build a reservation-message.*/
        if (f.ack_flag for output_port is not set) then  /* This request is already acknowledged. */
           inform_scheduler (f, r.QoS, output_port)      /* Inform the scheduler about the QoS for flow f. */
        end
     end   /* if (QoS != 0) */
  end
```

15. Function *delete_resources_on_port()*:

```
  /* Delete resources for flow f on port output_port. Tear-down any downstream resources. */
  delete_resources_on_port (f, output_port) begin
     clear f.status_flag for port output_port          /* Clean up state for port output_port. */
     clear f.ack_flag for port output_port             /* No acknowledgement required for a tear-down. */
     clear f.resv_flag for port output_port            /* Keep record where reservations are. */
     if (r.QoS != 0) then                              /* Existing resources on output_port. */
        r.QoS = 0                                      /* Release any resources held for this flow. */
        request = build_request_message (f, 0, output_port)   /* Tear-down any downstream resources.*/
        inform_scheduler (f, 0, output_port)           /* Inform the scheduler: QoS = 0 for flow f. */
     end
     delete_resv_state (f, output_port)                /* Delete reservation state r for f on output_port. */
  end
```

Copyright (c) 1996 Hewlett-Packard Company. All rights reserved.