NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

NASA-IVV-95-011
WVU-SRL-95-011
WVU-SCS-TR-95-31
CERC-TR-TM-95-013

Design, Implementation, and Verification of the Reliable Multicast Protocol

by Todd L. Montgomery

National Aeronautics and Space Administration

West Virginia University

Design, Implementation, and Verification of the Reliable Multicast Protocol

Thesis

Submitted to the Graduate Program in Engineering

West Virginia University

In Partial Fulfillment of the Requirements for

The Degree of Masters of Science

in Electrical Engineering

by

Todd Montgomery

Morgantown

West Virginia

1994

Acknowledgements

The author would like to thank his wife, Lisa, for her encouragement, understanding, and endurance. Without her this would not have been possible. Also, the author would like to thank his parents, sister, brother-in-law, and niece for unquestionable support and love.

An immeasurable amount of appreciation and gratitude go to Brian Whetten of USC Berkeley, the co-designer and inspirator of RMP, and to Dr. John R. Callahan, for his undying belief and support. These two gentleman have provided enormous expertise, knowledge, and most of all, *friendship.*

Appreciation and respect are also given to the other members of the Academic Examining Committee (AEC), Dr. Afzel Noore and Dr. Powsiri Klinkhachorn. Most of all, Dr. Afzel Noore, has allowed the author freedom to examine and work on such an intensely demanding project and provided nothing but encouragement.

In addition, a special group of people need to be acknowledged: Steve Husty, Brian Cavalier, Jeff Morrison, Yunqing Wu, Wei Sun, Matthew Fuchs, and Yahya Alsalqan. These individuals have provided support, encouragement, and knowledge in various ways.

A special note of gratitude goes to Nick Maxemchuk and Jo-Mei Chang whose work on the Reliable Broadcast Protocol forms the basis for RMP.

Finally, a great deal of appreciation goes to NASA Cooperative Research Agreement NCCW-0040, NASA Grant NAG 5-2129, and the NASA Headquarters Office of Safety and Mission Assurance (OSMA) under which this research has been supported.

Contents

List of Tables

List of Figures

List of Acronyms

ACK	Acknowledgment
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CSCW	Computer Supported Cooperative Work
DVMRP	Distance Vector Multicast Routing Protocol
FDDI	Fiber Distributed Data Interface
FIFO	First-In-First-Out
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IP	Internet Protocol [21]
Kbps	Kilobit per second
KBps	Kilobyte per second
LAN	Local Area Network
Mbps	Megabit per second
MBps	Megabyte per second
MBone	Internet Multicast Backbone

MOSPF	Multicast Open Shortest Path First
MTU	Minimum Transfer Unit
NACK	Negative Acknowledgment
OSPF	Open Shortest Path First
QoS	Quality of Service
RBP	Reliable Broadcast Protocol [3]
RIP	Routing Information Protocol
RMP	Reliable Multicast Protocol
RPC	Remote Procedure Call
SMV	Symbolic Model Verification
TCP	Transmission Control Protocol [22]
TRP	Token Ring Protocol [3]
TSP	Timestamp abbreviation
TTL	Time-To-Live
UDP	User Datagram Protocol [20]
WAN	Wide Area Network

Chapter 1

Introduction

As memory, storage, reliability, and bandwidth increase, decentralization of data storage increases. This introduces new and complex issues that must be dealt with. Reliability, resiliency, ordering of messages, and fault-tolerance of communications are all low-level concerns of application developers that are introduced by the decentralization of computation into distributed systems.

The lack of reliable broadcast, or reliable multicast, primitives has introduced a host of unnecessarily complex and inefficient schemes for providing application specific fault-tolerance, reliability, and resiliency. Although addressing the difficult issues, usually these schemes are not reusable in other applications because the closely knit connection between the application and its communication mechanisms. In many cases these are so closely interconnected that one is not distinguishable from the other. This virtually eliminates interoperability and can seriously effect application evolution. Distributed application should have a robust transport primitive that provides reliable delivery, ordering of messages, and fault-tolerance, thus unburdening the application developer and allowing concentration on the application development itself.

The *Reliable Multicast Protocol (RMP)* is such a primitive. RMP provides a totally ordered, reliable, atomic multicast service on top of an unreliable multicast service. RMP is based on the set of *Reliable Broadcast Protocols* presented by J. M. Chang

and N. F. Maxemchuk [3]. RMP substantially expands on this by using multicast technology instead of broadcast technology, including several extensions to provide extra functionality, and changing the fault recovery model.

Chapter 2 discusses some background concepts and provides a basic introduction to IP multicast technology. Chapter 3 presents the protocol model, design, and a representation as a finite state machine. Chapter 4 presents an overview of the first implementation of the protocol and the issues and lessons learned. Chapter 5 discusses the verification approaches to the protocol and issues related to protocol verification in general. Chapter 6 presents some performance results obtained by the first implementation. Finally, Chapter 7 discusses some conclusions about the protocol design, implementation, and verification and presents future research that is planned. In addition, Appendix A and Appendix B present the message formats of the protocol and the complete state tables.

1.1 Research Objectives

The research objectives of this investigation are:

1. Co-design the Reliable Multicast Protocol.

2. Implement the design of the protocol and experiment with features to provide a continually evolving and comprehensive protocol design and implementation.

3. Interact heavily with the verification effort in an attempt to maintain a close model representation of the system and to assist the effort in determining and using the appropriate tools.

Chapter 2

Background

This chapter presents a background to IP Multicast technology and to other distributed application development abstractions.

2.1 IP Multicasting

Multicasting is a technique used to pass copies of a single packet to a subset of all possible destinations. When this subset is the entire set of all possible destinations, this is called *broadcasting*. This technique has been supported for several years on some local area networks. Most notably of which are Ethernet and Fiber Distributed Data Interface (FDDI) networks. On these, a multicast packet to all of the hosts has the same overhead as a unicast packet to just one of them.

When the destination subset is less than the entire set of all possible destinations, some addressing scheme must be used to represent the subset, called the multicast *group*. When a host desires to receive multicast traffic destined for a certain multicast group, the host should, in effect, join the multicast address. This addressing is an Internet standard [4]. An IP multicast address is a Class D address, and is always used as destination address. The valid range for IP multicast addresses expressed in dotted decimal notation is 224.0.0.0 to 239.255.255.255. The 224.0.0.0 address is reserved and can not be used by any multicast group. In addition, address 224.0.0.1 is reserved

for the *all hosts group* on the directly connected network, and addresses 224.0.0.2 to 224.0.0.255 inclusive are reserved for routing protocols and low-level topology protocols. This leaves the address set of 224.0.1.0 to 239.255.255.255 open for general use.

The IP multicast interface is currently supported at the TCP/IP layer of hosts that have been configured to support IP multicast technology [1]. This layer provides an address and a *protocol port.* The address is specified as an IP multicast address, and the port is an abstraction that allows multiple destinations within a given host to be distinguished [2]. The underlying transport layer for IP multicast is either the User Datagram Protocol (UDP) or the raw IP datagram protocol. Both of these protocols employ "best-effort" on reliability. That is to say that the datagram is not guaranteed to arrive intact at the destinations, and that no ordering guarantees are given relative to other datagrams. UDP however does incorporate a checksum to detect damaged packets. IP datagrams do not incorporate this checksum.

Immediately the need for a reliable, atomic, and fault tolerant protocol should be apparent. With an unreliable and unordered base transport mechanism forming the bottom layer, some sort of protocol is needed to provide reliability, atomicity, and fault-tolerance to those applications that demand it.

2.1.1 Multicast Routing

The IP multicast model can be expanded to encompass extended LANs and internetworks by modifying the routers that connect the LANs together. Normal IP routers do not support multicasting. The multicast traffic does not leave the local network via the IP router unless the router supports IP multicasting. Commercial router hardware

[1] Current vendors include Sun Microsystems, Silicon Graphics, Digital Equipment Corporation, and Hewlett-Packard.

[2] The port designation is usually an integer value of 16 bits.

is quickly upgrading to support IP multicasting. As an ad hoc solution, many network providers are using "tunneling" to provide routing of multicast traffic. Tunneling is a scheme that involves encapsulating an IP multicast packet inside a regular IP unicast packet and sending the packet to another multicast capable router. This is supposed to be done by a dedicated host on the local network or subnet. However, currently most multicast routing is being done by hosts that serve other purposes in addition to performing multicast routing. A multicast router, or *mrouter*, may have several tunnels pointing to various other networks or subnets. When a destination mrouter receives a multicast packet, it will strip the encapsulation information from the packet and multicast the packet out onto the local network. The destination mrouter may also forward the packet along its own tunnels, thereby extending the range of the packet to other networks.

Multicast routers use one of two routing protocols for routing multicast packets to other mrouters. The first is the Distance Vector Multicast Routing Protocol (DVMRP). This protocol is the one implemented by the majority of mrouters at this time. Its operation is detailed in an Internet standard [19]. This protocol maintains topological information by means of a routing protocol based on the Routing Information Protocol (RIP) [7], but enhances the protocol by implementing a multicast forwarding algorithm.

The second multicast routing protocol, MOSPF, is being used in a commercial router that supports IP multicasting. MOSPF is the multicast extension to the Open Shortest Path First (OSPF) routing protocol [17],[18]. This routing protocol does not need to use tunnels and does not send more than one copy over a link. Other commercial router providers are in the process of either adopting MOSPF, DVMRP, or developing their own routing protocols.

Distribution of multicast traffic on an Internet scale must be controlled in some fashion. Currently two ways exist to control multicast packet distribution. They are:

- "Pruning" of the routing tree to adaptively restrict traffic, or
- Limit multicast packet lifetime.

The pruning approach is currently being tested on a limited scale in some DVMRP mrouters. In this way only branches of the routing tree that have members that are part of a multicast address receive the multicast packets for that group. Other mrouters, except MOSPF, only trim leaves from the routing tree and allow all traffic to be forwarded down branches that have no interested hosts. Thus some other mechanism is needed to restrict the scope of a multicast. This is done through placing lifetime restrictions on packets and lifetime thresholds on each of an mrouters tunnels. The basic premise is that only packets with a Time-To-Live (TTL) greater than 1 should be able to leave the local network or subnet in which they were created. The mrouter on that local network also decides, based on the TTL of the packet and the threshold value of its tunnels, whether the packet is forwarded out through any of its tunnels. Metrics associated with each tunnel decrement the TTL of the packet as it is transmitted across. Higher TTL packets therefore have a much larger scope than smaller TTL packets because the TTL can stay larger than most threshold values as the packet travels among subnets. The Internet Engineering Task Force (IETF) has imposed these suggested TTL conventions on scope: [3].

TTL	*Scope*	*TTL*	*Scope*
0	Same host	64	Same region
1	Same subnet	128	Same continent
32	Same site	255	Unrestricted

2.1.2 Internet Multicast Backbone

The virtual network of interconnected multicast routers forms the topology of one of the Internets fastest growing resources, the Internet Multicast Backbone or *MBone*. The MBone is currently of global proportions with more than 20 countries supporting

[3]The "site" and "region" scopes are flexible.

connections [4].

The MBone allows a single stream of information to be received by a *large* number of hosts that are distributed globally. The MBone started as an experiment to broadcast live video and audio of the proceedings of the IETF meetings. It has expanded to include several more broadcasts, including other large conferences, a few talk shows, and live television and radio feeds.

Bandwidth is a major issue for the MBone community. This is especially true because the MBone topology is predominantly composed of mrouters that are not dedicated solely to routing multicast traffic. One video stream consumes about 128 Kbps, and one audio stream consumes about 64 Kbps. With these numbers it has been assumed that the desired MBone bandwidth limit should be approximately 512 Kbps. That is approximately four simultaneous video streams or eight simultaneous audio streams. At any one time this is very seldom exceeded although tighter restrictions on MBone event scheduling for broadcasting conferences and other events is starting to occur.

The MBone provides more than audio and video dissemination. Several other applications exist and are being developed to use the MBone as its internetwork multicast transport mechanism. Among these are efforts to develop distributed database systems, fault-tolerant simulators, and Computer Supported Cooperative Work (CSCW) applications, such as shared tools.

2.2 Distributed Application Development

The widespread availability and use of such protocols as TCP has freed the developers of applications that use unicasts to communicate from having to implement message reliability and ordering into their applications. The lifting of lower-level burdens has

[4] More than 900 multicast routers compose the topology.

allowed developers to think at a more abstract level and to draw out useful concepts that can used by other developers.

2.2.1 Process Groups

A *process group* is a set of processes that interact to perform a distributed operation. The operation need not be symmetrical. The case may be that during the course of the operation one process bears more of the communication or computation burden than another process. The development of a distributed application by envisioning the application as composed of a process group or as a process in a process group is very useful.

Two process group communication models, or architectures, are frequently used. These are the *publisher-subscriber* and *client-server* models. The publisher-subscriber model allows processes to join, or *subscribe*, to a group. Group processes may send to the group, or *publish*, messages for the group to receive. This model does not require the explicit naming of the message destination. This is very useful for applications that desire to avoid the restriction of having to explicitly name the destination of a message. This also addresses the problem of naming message destinations in a mobile distributed application, such as agents.

The client-server model is more traditional. It involves the operation of a *server* process that waits for requests from *clients*. Clients send a message to a server and usually wait for a reply, which the server sends back. In many cases the naming of the client and the server must be explicit.

Two other group models that are also useful are *Diffusion Groups* and *Hierarchical Groups* [1]. Diffusion groups are groups that are composed of a single set of senders in the group and a single set of passive receivers in the group. This style is the most frequently used style of group for most normal multicast applications. A hierarchical group consists of multiple component groups. The component group and the inter-

connection of them may use other models of communication. This approach is useful for applications where scalability and/or security are major concerns.

2.2.2 Event Ordering

Within a group correct ordering of messages may or may not be of critical performance. It is entirely dependent on what distributed operation the group is performing. Most distributed applications desire at least *source ordering* of messages. This means that each message from a source is ordered with other messages from the same source in a First-In-First-Out (FIFO) style.

Certain group applications desire even stricter ordering of messages. Synchronization concerns and mutual exclusion may need to be addressed as well. When operations need to be performed on data or objects that have no mutual connection, then a *causal ordering* scheme may be sufficient [1], [12]. This scheme allows messages to be processed in different orders at different processes so long as the operations have no relationship and the objects operated upon have no relationship. If a relationship does exist between data or operations, then *atomic* ordering, or *total ordering*, is needed. In this scheme all messages are processed in the same order at each process in the group. This scheme allows the execution model of the system to appear to have a property called *Virtual Synchrony* [5] [1].

Virtual Synchrony provides many benefits to a distributed application. It is a system model that allows the developer to assume a more simplified and systematic execution. Applications that manage global state information, or that desire mechanisms to transfer state information to other group members, can easily use this model to implement such devices.

[5] From Ken Birman - "Intuitively, this means that the user can program as if the system scheduled one distributed event at a time.[26]"

2.2.3 Reliability, Resiliency, and Fault Tolerance

Communication within a group may need to be reliable. Cases where reliable delivery is not a concern are real-time video and audio transmission. Reliable communication in these cases is not of much use unless strict real-time guarantees can also be put on the messages so that the quality and latency desired by the application can be achieved. Many applications, however, desire that their messages be reliably sent. Especially when transactions and global state information is concerned, reliability is very important.

In the case of failures, resiliency may also be desired. Resiliency of messages attempts to impose that under certain failure assumptions the message will be delivered. This concept goes hand in hand with fault tolerance. Fault tolerance attempts to assure that under certain failure assumptions the system will be able to recover transparently to the application and proceed. Together these concepts attempt to provide a system model that is robust in the face of failures and has the ability to recover from failures transparently and without message loss.

Chapter 3

Design of the Reliable Multicast Protocol

This chapter discusses the design of RMP and presents the protocol itself. First the primary features of RMP are introduced. Second, the general RMP model described. Next, the data structures and their operations are detailed. Finally, the last section details a state machine representation of the protocol.

3.1 Primary Features of the Reliable Multicast Protocol

RMP provides a transport mechanism by which the user can design and implement fully distributed, fault-tolerant applications without the need to deal with the lower level primitives of communication. In order to do this, RMP provides several features, both in terms of performance and functionality. These are:

- High throughput for *Totally Ordered* messages, with low latency
- Virtual Synchrony
- Support of process group models
- Efficient changes to the process group
- Scalability of process groups, and
- Flexibility of choice for resiliency and fault-tolerance levels

Since RMP is aimed at providing a transport level service, performance is a very high priority. The protocol needs to be efficient and robust. The key point sought

by RMP is high throughput for totally ordered messages with low latency. The total ordering of messages is imparitive to providing virtual synchrony. However, RMP provides a means to control the ordering of messages on a per message basis.

RMP supports the publisher-subscriber and client-server communication models for the application developer. The client-server model allows RMP to scale to hundreds of simultaneous users. Scalability is also encouraged by the support of the hierarchical group model.

Flexibility in communication models, message resiliency, fault tolerance, and ordering guarantees allow RMP to provide an open architecture for the application to use. This allows RMP to be used as a basic cornerstone that forms the foundation for a whole collection of layered functionality.

3.2 Protocol Model

RMP is based on a model of complex interaction between processes operating on interconnected hosts. It is important to discuss the basic entities involved in this interaction, to present the interaction model, and show how the RMP design has adjusted the model to achieve the desired features.

3.2.1 RMP Entities

RMP is organized around three entities: *RMP Processes*, *Token Rings*, and *Token Lists*. An RMP Process, or *process*, is the basic entity of a larger system that uses RMP to communicate. An RMP Process can be thought of as a member of a process group that is using RMP to provide its transport mechanism. Multiple processes may exist on the same host. Under this requirement, it is necessary to uniquely identify each one. This is accomplished by taking the IP address of the host and combining that with the protocol port that the process will use for communication.

This combination is referred to as the *RMP Process ID*, and should be unique across the entire internetwork.

A group of processes, communicating to achieve message ordering, is referred to as a *Token Ring*. Each process may be a member of multiple token rings. Alternatively, processes may not be a member of any token ring and may communicate with a token ring through the use of a client-server communication model. Processes that are executing on a host that is not multicast capable may also be part of a token ring. Each token ring is given a textual string representing its *Token Ring Name*. This name is similar to the text strings used for current Internet host and domain names. Instead of only specifying one domain or one host, token ring names represent a dynamic collection of processes. The mapping of a token ring name to a set of IP Multicast address and port values can be handled implicitly by the process. This may be done by using a random hash function to generate a unique address and port set of values directly from the text string. Alternatively, this mapping may be handled by an external multicast address management authority. As a third possibility, a token ring name need not be directly mappable at all. The IP Multicast address and port values may be explicitly chosen by the developer.

The list of members of a token ring is called the *Token List*. Each process in the list maintains a current version of this list for reference operations for protocol operation. A token list is always created by one process, called the *Originator*, which generates a *Token List ID* for the list. This Token List ID is composed of two pieces of information. The first piece is the Originator RMP Process ID, and the second piece is a unique counter associated with the Originator. This counter may be the current time in millisecond resolution or some other value unique to the RMP Process and to the host. The guarantee of the RMP Process ID being unique across the entire internetwork, and the guarantee that the Originator counter is unique to the host, precludes the need to maintain unique IP Multicast address and port combinations

across multiple token rings. This is the result of the fact that collisions to the same address and port may be resolved by filtering packets based on Token List IDs. Token Lists IDs are changed each time a new token list is generated, whether it is the result of a membership change or a failure. In addition, a Token List ID has a Time-To-Live, currently 45 minutes. If a site that generated the current Token List ID notices that the ID is older than this, then the site must generate a new Token List ID. This is to assure that IDs do not become invalidated because of long inactive times of operation.

Token lists are also referred to as *Membership Views* because they represent a view of the current membership of the token ring. RMP extends this view to include other information about a token ring member. Most notably, a member may hold one or more *Locks.* These locks are mutually exclusive. The semantics attached to a lock are totally dependent on what the application requires the lock to do. Six of these locks are set aside as *Handlers.* A handler is a mechanism that allows a message to be sent to a group and have only one member reply or handle that message. In effect, the handler marks the message as requiring a reply from only one member of the token ring.

Processes that are not part of the token ring may send a message to the group, and optionally be either notified of its delivery to the group or receive a reply from one of the members. This model of RMP communication, referred to as *Multi-RPC*, allows RMP to support a client-server communication model. This allows processes that desire information from the token ring, but do not want to pay the overhead price of actually joining the Token Ring, to communicate directly with the ring.

RMP is designed to take advantage of an unreliable multicast service as its base communication. However, for increased flexibility, it also supports hosts that are not multicast capable. This concept extends multicasting to include processes that can only communicate through unicast methods. This is accomplished through two mechanisms. The first is denoting processes that are not multicast capable in the token

list and mandating that such members require unicast transmission of all messages sent to the ring. The use of RMP Process IDs to identify members makes this trivial. The second mechanism is the use of a forwarding flag on each message. This allows non-multicast capable members to send a message to a multicast capable member of the ring and have that member forward the message to the ring via a multicast.

3.2.2 Interaction Model - Post-Ordering Rotating Token

Traditionally, two schemes are primarily used to provide reliability on top of unreliable communications media. The first, *Positive Acknowledgment*, involves requiring a receiver to explicitly send a notification that a message was received. This notification is commonly called an ACK. This scheme does not scale well to multiple destinations because the number of ACKs sent per data message becomes high, therefore reducing available bandwidth and hurting performance. This approach also does not provide any ordering guarantees on the received messages at the receiver. However, the sender may place sequence numbers on the messages providing the order in which each was sent.

Negative Acknowledgment schemes place the burden of detection of transmission loss on the receiver. Each message a sender sends is stamped with a sequence number to differentiate that message from the ones before it and after it. If the receiver receives one message with a sequence number higher than the one it was expecting, the receiver can request a retransmission of the lost message. This request is commonly called a NACK. Ordering with respect to each sender in this approach is inherent. Using this approach, there is no way to determine when a receiver has actually received a message. Therefore the sender must save all messages indefinitely. For some applications this approach may be optimal. For many other systems, such as distributed databases, the infinite buffer required is too costly. This approach provides no global ordering for multiple senders. However, real time information can be placed in each message,

specifying the global time each was sent. This involves a clock synchronization step.

RMP is based on a modified version of the family of protocols presented by J. M. Chang and N. F. Maxemchuk [3], referred to here as TRP, *Token Ring Protocol.* TRP makes the general system of many senders and many receivers appear to be a combination of two simpler systems. To the receivers, the system appears as a system with one sender by having all the messages serialized by one site. This site is called the *Token Site.* The token site sends an ACK, or acknowledgment, containing a special sequence number, called a *timestamp.* This ACK is in response to a data message sent to the group. In this way the system operates as a positive acknowledgment scheme to the message sender, who knows whether or not his message has been received by at least one receiver. The system also operates as a negative acknowledgment scheme to the other receivers, who know if they miss a message because of the imposed timestamp ordering. Missing messages can then be requested by sending a NACK to the other receivers or the sender requesting a retransmission of the missing message. In addition, a message source places a sequence number on a message to order that message with respect to other messages it has or will send. Sequence numbers and timestamps are two different entities. Sequence numbers provide ordering of messages with respect to the *same* site. Timestamps, however, provide *global* ordering across all sites.

The infinite buffering of the negative acknowledgment scheme still applies to this approach. This results in the need for some sharing of the token site responsibility among members of the token list [1]. This sharing is done by rotating the token as a consequence of generating the ACK for a message. For a receiver to accept the token, it must have all the timestamped messages. This maintains consistency and assures that a message that is lost can be recovered. Livelock is avoided by making the token pass mandatory within a specified time interval. If a site does not see a

[1]The group of receivers and senders, also called a *Token Ring.* The similarity between the Token Rings in TRP and Token Rings in RMP is intentional.

message that may be acknowledged within this time period, then the site must pass the token by sending an ACK that does not acknowledge any message. This special ACK is called a Null ACK. In order to eliminate the need to keep sending Null ACKs while no messages are being sent, the token ring may go into *quiescence*, or inactivity, after the token has passed around to every member of the token list once with Null ACKs. The ring becomes active again once a message is sent to it.

Delivery of a message, or *committing* a message, to the application is possible after N transfers of the token, where N is the token ring size. This ensures that each receiver has the message before the application sees it. This is referred to as *Totally Resilient,* or *safe,* delivery. In addition, the message is termed *stable* when it reaches this point. If K sites accept the token after the message is timestamped, then $K + 1$ sites have the message and $K + 1$ sites would have to fail before the message is lost. This is the concept of *K-Resiliency* of messages. A message can be assumed to have *agreed* delivery semantics if more than half of the sites have the message. This is also referred to as *Majority Resilient* delivery.

Because of the rotating of the token and the ordering of messages after they are sent, this model is referred to as the *Post-Ordering Rotating Token* approach. Pre-Ordering approaches restrict the sending of messages to sites that have the token. Some protocols which use this approach are [14] and [11]. This approach suffers from large penalties when used on high latency networks, such as WANs. Also this approach does not allow asynchronous sending of messages as done in the TRP approach.

A fault-tolerant system must specify the failure assumptions under which recovery is possible. RMP shares the following TRP failure assumptions:

1. A site failure means the site stops processing. The site does not interject corrupting information into the group.
2. A message failure can be the result of an overfull buffer at either the receiver or the sender, or it may be the result of a transmission failure. [2]

[2] Given current networks, this is very rare ($\ll$ 1% of packets)

3. A failure is detected by the group when communication with the group and a site fails after R attempts. R must be chosen such that a failure is mistakenly detected infrequently, but large enough to provide timely notification of actual failures.

Failures are detected in TRP and in RMP by the use of retransmissions after a timeout period. A sender keeps retransmitting its message until it receives an ACK for that message. A token transfer is retransmitted until a positive confirmation is received that the new token site has accepted the token. This positive confirmation can be another token pass or it may be a confirmation message sent to the original site. Lastly, all negative acknowledgments, or NACKs, are retransmitted until the requested missing packet is received. Therefore each one of these messages require a form of positive acknowledgment. If that positive acknowledgment is not received in a given period, the non-responding site is assumed to have failed. The TRP protocol presented so far covers the the normal operation of the protocol, called *Normal Phase* operation. Once a failure is detected, the *Reformation Phase* of the protocol begins.

The TRP reformation protocol involves three phases by the fault detecting site and two phases for the other sites, or *slaves*. The fault detector, called the *Reform Site*, first forms a valid new Token List. Next the new list must pass a majority and a resiliency test [3], the list must be sent to the slaves, and a new token site must be elected. The last phase involves the authorization of the new token site, the generation of a new token, and the passing of the new token to the new token site. After this, normal operation resumes. The formation of a new list is achieved through the use of a voting scheme with the reform site being the arbitrator.

To illustrate the ordering of messages and the normal operation of the Post-Ordering Rotating Token approach, examine Figure 3.1. In the figure, three sites (A,B, and C) are shown. They are connected by a multicast capable communication

[3]The majority test tries to ensure no two reformations can concurrently continue, and the resiliency test assures that no messages from the old list are lost.

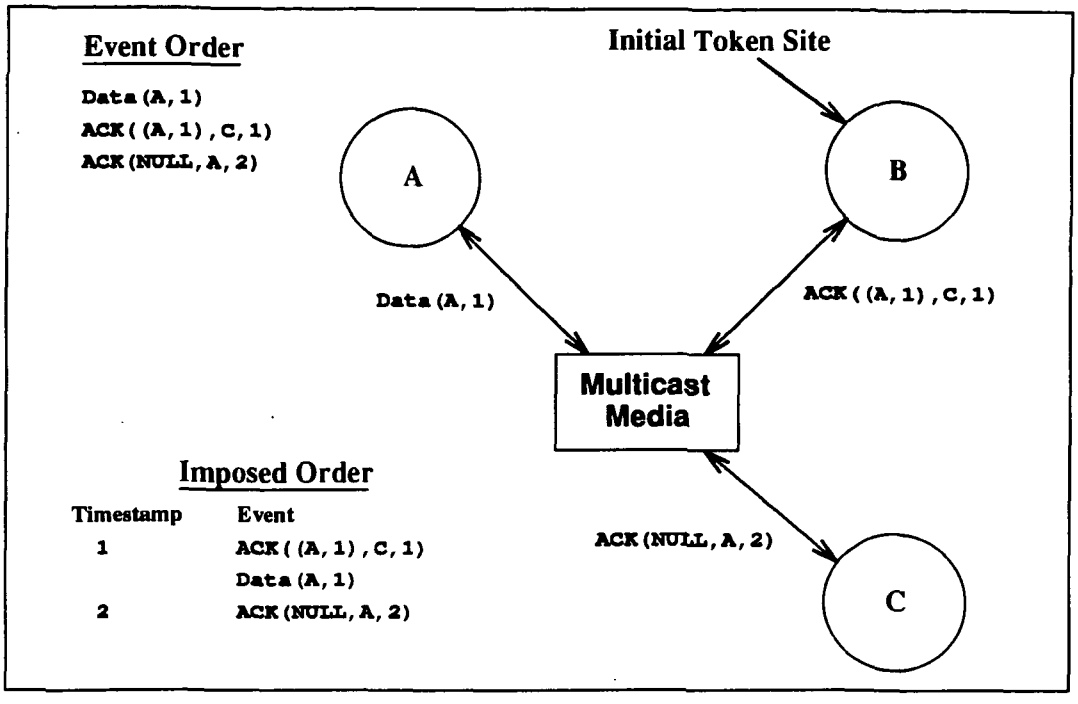


Figure 3.1: Rotating Token Example

media. Initially site B is the token site. The order of events seen on the network are shown as *Event Order*. First a message containing data is sent to the network from site A. Site A places a sequence number on the message of 1. This is shown as Data(A,1) in the figure. Site B, being the current token site, notices the message and generates an ACK, ACK((A,1),C,1). This ACK contains the site and sequence number for the message it is acknowledging, (A,1), the next token site, C, and a timestamp, 1, to globally order the message. The imposed ordering of messages is shown as *Imposed Order*. Site C is now the token site. No message is sent within a certain period of time after site C becomes the token site, therefore, site C must pass the token. Site C does this by generating a Null ACK and sending it, ACK(NULL,A,2). The token is therefore passed to site A, and the Null ACK is ordered by giving it a timestamp of 2. Delivery of the site A message will not occur until site B becomes the token site again.

3.2.3 Modifications to the Token Ring Protocol

As has been stated, RMP is a modification of the Token Ring Protocol presented in Section 3.2.2. RMP modifies TRP by the following additions and changes:

- Allowing multiple messages to be acknowledged per ACK
- Changing message commitment policy to be on a per message basis
- Allowing New Token Lists to be generated without Reformation
  - This allows efficient Token List changes
  - Allowing other information to become part of the membership view
- Changing the Reformation protocol
  - Not allowing members to be added to the list during Reformation
  - Allowing minority partitions to continue, if desired
  - Allowing atomicity to be violated, but notifying application of it occurring
  - Providing adjustable resiliency on a per Token Ring basis

It can be seen that under heavy load, where several sites are simultaneously sending, that messages would tend to queue up before being acknowledged. The solution to this problem is to allow multiple messages to be acknowledged and ordered per ACK sent. This has a secondary effect of, under heavy load, lowering the message size to ACK size ratio. This increases network utilization since a larger majority of data is being sent through the system while a lower portion of the bandwidth is being taken up by acknowledgments.

Various types of applications demand different levels of atomicity and resiliency than the Totally Resilient TRP policy of message delivery. To support these differing levels, RMP has changed the committing policy of message to be more flexible. Each message is to be committed based on its own desired *Quality of Service (QoS)* . This QoS ranged from *Unreliable*, where the message is committed upon reception, to *Totally Resilient*, where the message delivery has the same semantics as in TRP. To facilitate this, each message is also given a timestamp within the global ordering.

QoS Level	*Description*
Unreliable	Delivery is immediate upon reception. Lost messages are *not* requested. These messages are not assigned sequence numbers by the sending site.
Reliable	Delivery is immediate upon reception. Lost messages are requested. These messages do receive a sequence number.
Source Ordered	Delivery is after all messages from the same source and with lower sequence numbers have been delivered.
Totally Ordered	Delivery is after all messages with lower timestamps have been delivered.
K Resilient	The same as *Totally Ordered* but with $K-1$ passes of the token required as well.
Majority Resilient	The same as *K Resilient* but with K being equal to $(N+1)/2$, where N is the size of the ring.
Totally Resilient	The same as *K Resilient* but with K being equal to N, where N is the size of the ring.

Table 3.1: QoS Levels

This timestamp is implied rather than explicitly assigned, as is done with ACKs. The timestamp of a message follows from the ACK timestamp in a monotonically increasing fashion. Therefore if an ACK contains two messages, and the ACK has a timestamp of 3, then the first message has an implied timestamp of 4, and the second message has an implied timestamp of 5. The different levels of QoS are shown in Table 3.1.

TRP does not cover the process of reaching normal operation from a totally memberless token ring. No attention is paid to membership changes either. Because efficient membership changes are needed for group communication, RMP has extended the base model to cover membership view changes. Membership view changes are not necessarily just additions and deletions from the token list. RMP combines pieces of information into the token list to effectively implement Locks and Handlers as mentioned in Subsection 3.2.1. Efficient view changes can be done by noticing that a membership view is really just a global state change. This change can easily be achieved by a totally ordered [4] message to the ring. And obviously, the change can be even more efficient if it were not actually a message, but a token transfer. Therefore a member could request a change via a specific type of message, referred to as a *List*

[4] Resiliency is not needed because any fault would invalidate the new list anyway.

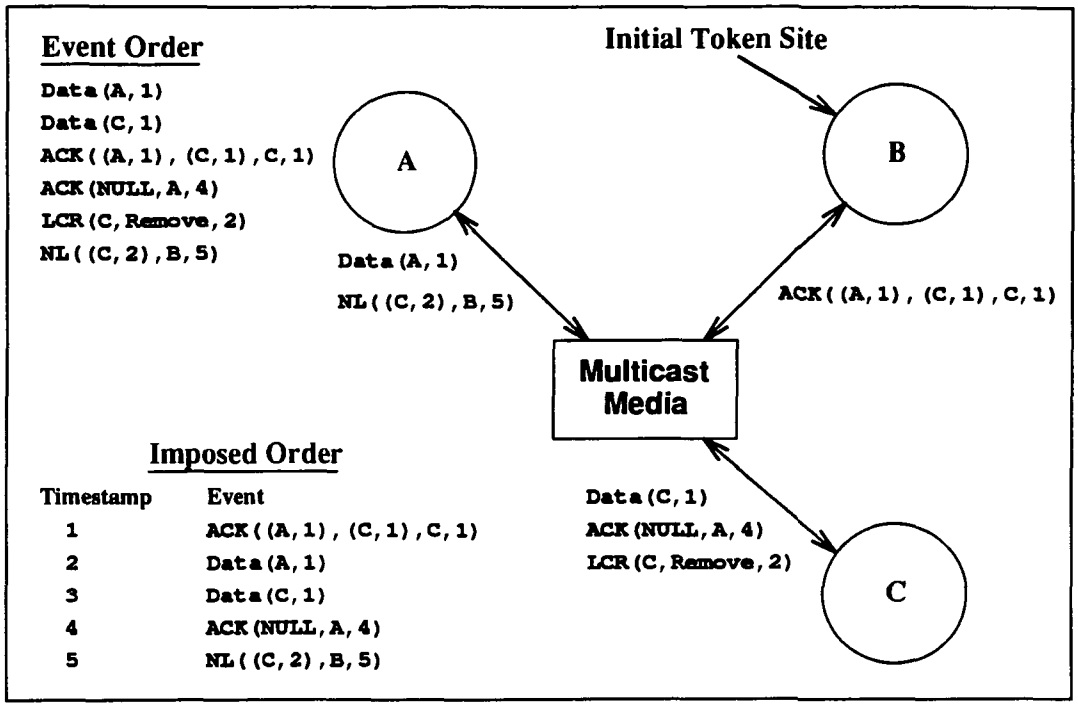


Figure 3.2: Modified Rotating Token Example

Change Request or LCR. The current token site serializes all requests and generates a new token list, referred to as a *New List*, timestamps the new list, and sends the new list as a token transfer. Thus an ordering of the change within the global ordering is achieved and Virtual Synchrony is maintained. Each LCR must have a corresponding new list that handles just that request. In this way, new lists act as acknowledgments for LCRs. Multiple changes per new list would violate Virtual Synchrony and nullify the effect desired by the membership view change.

The Rotating Token example can now be modified to include the ability to acknowledge multiple messages and the ability to change the membership views. The new example is shown in Figure 3.2. The initial token site is once again site B. Site A sends a message with sequence number 1 and almost simultaneously site C sends a message with sequence number 1 as well. Site B sees the site A message just before the site C message and therefore orders the two messages by sending an ACK,

ACK((A,1),(C,1),C,1). The ACK will be placed in the *Imposed Order* with a timestamp of 1. The messages will also be placed in the order with timestamps of 2 and 3. These timestamps are *implied* because of the order they are placed in within the ACK. The new token site is C. As in the first example, site C generates a Null ACK to pass the token to site A. The Null ACK is given a timestamp of 4, ordering it after both the other messages. Site C decides that it wants to remove itself from the ring. To perform this operation, site C sends an LCR that contains a sequence number of 2, ordering it with respect to the first message from site C, and requesting site C to be removed from the ring. Because site A is the current token site, site A generates a new list, NL((C,2),B,5), that does not contain site C in it and sends the new list to the ring. As a consequence of generating the new list, the token is passed to site B. The new list is ordered within the global ordering by being given a timestamp of 5. The new list that was generated corresponds with the LCR sent from site C with a sequence number of 2.

Both of the Data messages shown have a corresponding QoS that is not shown. Suppose that the last delivered timestamp was 0, the last delivered sequence number from site A was 0, and the last delivered sequence number from site C was 0. Now also suppose that the QoS of both Data messages is totally ordered. Therefore, once the ACK for (A,1) and (C,1) are received, those Data messages can both be committed. And once the new list is received, it also may be directly committed [5]. New lists have an implied QoS of totally ordered.

Reformation in RMP has been almost totally redesigned from TRP. The TRP Reformation protocol did not take into direct consideration what exactly should be done upon a failure during the reformation process. For this reason, and to provide a more flexible and adjustable failure recovery model, a reformation protocol for RMP was developed.

[5] Because timestamps 1–4 have already been delivered.

The flexible failure recovery model desired by RMP is one that contains the ability to return to normal operations in the face of specified minimum size partitions. If a ring required that at least four members be in the ring after a failure, then the ring should be able to recover if all the members fail except four. The majority test of TRP ensured that when a partition occurred that a majority of the old sites had to be in the new list of sites. This is not always desirable or necessary for continued operation and is therefore not part of RMP. But this test also has a side effect. No two simultaneous reformations may continue if a majority of the sites are needed for a valid list to be created. RMP makes up for this by checking at each step of the protocol to detect and recover from simultaneous reformations. The TRP reformation protocol also allowed new members to join a ring when it was in reformation. This does not follow Virtual Synchrony and therefore has been taken out of the RMP reformation protocol.

The resiliency test of the TRP reformation protocol is used to ensure that messages from the old list are not lost. RMP adopts this test, but also relaxes it so that a new list may be formed that can have missing messages. This allows the application to decide if those missing messages were important enough to warrant a shutdown of the ring. In some cases an application may keep a small buffer of sent messages for recovery from just this type of situation.

3.3 The Data Structures and Algorithms

The specification of any communications protocol needs to be very clear and concise. The specification of RMP contains a set of message formats, called *packet* formats, two simple data structures, some algorithms that operate on these data structures, and a finite state machine.

3.3.1 Packet Types

RMP is a packet based system. This means that each message sent is a single packet. Fragmentation and reassembly of large messages into a stream is the task of a higher level interface [6]. The idea of a message and a packet in RMP is, at this level, the same.

RMP uses many different types of packets to communicate. This is necessary because of the large amount of state information involved with the protocol. RMP packet types are split into two different categories. The first category is the actions associated with the packet type. These are *Control Packets*, *Failure Recovery Packets*, and *Non-Member Packets*. Data Packets do not fall into any of these actions. The second category is the state machine extension that the packet addresses and is used in. These are *Normal Operation*, *Multi-RPC Extension*, *Membership Change Extension*, and *Failure Recovery Extension*. The packet types and their corresponding categories are shown in Table 3.2.

Each RMP packet contains one header split into two parts. The first part is the same for all RMP packets. This is called the RMP Fixed Header. The RMP Fixed Header contains the following information:

- A bit marking the packet to be "forwarded" to the group
- The RMP Protocol Version feducial
- The type of RMP packet
- The Token List ID for the packet

Each RMP packet also has an additional part that contains information relevant to protocol operation. The exact format of each packet is shown in Appendix A on page 92. For now it is only important to understand the types of RMP packets and what they are generally used for.

[6] The maximum packet size is chosen based in order to optimize performance.

Packet Type	*Algorithm*	*Description*
Data	Normal Operation	Contains data from one member of the group
Control Packets		
ACK	Normal Operation	Orders Data Packets and Non-Member Data Packets
NACK	Normal Operation	Requests retransmissions of one or more packets
Confirm	Normal Operation	Confirms a Token Pass
New List	Membership Change	Contains a Membership View
List Change Request	Membership Change	Requests a change to the current membership view
Non-Member Packets		
Non-Member Data	Multi-RPC	Contains data from a process that is not a member of the group
Non-Member ACK	Multi-RPC	Response to a Non-Member Data Packet
Failure Recovery Packets		
Recovery Start	Reformation	Start Reformation
Recovery Vote	Reformation	Members Vote of group properties
Recovery ACK New List	Reformation	Member has received New List
Recovery Abort	Reformation	Abort Current Recovery, Start again

Table 3.2: RMP Packet Types

Some RMP packets require a form of positive acknowledgment. Any time an RMP packet requires a positive acknowledgment, that packet must be periodically retransmitted until the positive acknowledgment condition is met. If after a set number of retransmissions, X, that condition has not been met, then it is reasonable to assume that a failure has occurred and reformation should be initiated. The packets that require positive acknowledgment and the conditions for meeting that positive acknowledgment are given in Table 3.3. Control of the timeout period length between retransmissions is a flow control topic, see Section 3.5. The maximum number of retransmissions before initiating reformation should be configurable based on network properties.

3.3.2 Data Structures

There are two data structures that are used by the RMP protocol, the *DataQ* and the *OrderingQ*. An individual implementation may have several more data structures, but

Packet Type	*Positive Acknowledgment Conditions*
Data	Reception of ACK Packet that contains Packet
ACK	Reception of ACK, New List, or Confirm Packet with Timestamp $\geq$ Timestamp of Packet
NACK	Reception of requested Packet(s)
Confirm	(none)
Non-Member Data	Reception of Non-Member ACK Packet that contains Packet
Non-Member ACK	(none)
New List	Reception of ACK, New List, or Confirm Packet with Timestamp $\geq$ Timestamp of Packet
List Change Request	Reception of a New List Packet containing response to request
Recovery Start	Creation of a valid New List Packet
Recovery Vote	Reception of a New List Packet from Reform Site
Recovery ACK New List	(none)
Recovery Abort	(none)

Table 3.3: Packet Positive Acknowledgments

the RMP algorithms and state descriptions use only these two data structures.

When Data Packets, List Change Request Packets, or Non-Member Data Packets arrive, they are first put into a FIFO for processing later. This FIFO is called the DataQ. The DataQ has no priority other than the time of arrival and is merely used as a means to determine in what order packets were received.

When ACK Packets or New List Packets arrive, they are put into another kind of FIFO. This FIFO is called the OrderingQ. As the name implies, the OrderingQ is the mechanism which primarily imposes ordering on packets. In addition, the OrderingQ also provides a means of detecting lost packets. Each member of the OrderingQ, also called a *slot*, is one of these four types of packets: Data Packet, Non-Member Data Packet, ACK Packet, or New List Packet. Each slot of the OrderingQ has an associated priority which orders the FIFO. The priority of each slot is based on the packet's timestamp. This timestamp is either explicit, in the case of ACK Packets and New List Packets, or it is implied, in the case of Data Packets and Non-Member Data Packets. No two members of the OrderingQ are allowed to have the same timestamp.

In addition to each slot of the OrderingQ having a timestamp for ordering, each

slot also has an associated "state" based on the disposition of the associated packet. That state is one of the following:

Packet Missing: Packet is missing, but NACK has *not* been sent.

Packet Requested: Packet is missing, and NACK has been sent.

Packet Received: Packet has been received.

Packet Delivered: Packet has been delivered to the application.

The DataQ has no direct associated operations that must be performed on it. But the OrderingQ has a set of algorithms that are used to change the state of each slot and to add and remove slots from the structure. These algorithms may change the elements of the DataQ as a result of their operations on the OrderingQ.

3.3.3 Data Structure Algorithms

RMP uses four algorithms to perform operations on its OrderingQ to add and remove slots and to change the state of individual slots. Each algorithm is executed based on the state of the site as presented in Section 3.4. These algorithms are:

- Update-OrderingQ,
- Attempt-Packet-Delivery,
- Add-ACK, and
- Add-New-List

In addition, a fifth algorithm, called *Pass-Token*, is used to determine which Data Packets, Non-Member Data Packets, or List Change Requests are handled by the next token pass.

The Update-OrderingQ algorithm, shown in Figure 3.3, should perform these actions each time it is invoked:

1. Check for missing timestamps and add slots if needed
2. Send NACKs for missing packets

```
Update-OrderingQ()
     for each (slot in the OrderingQ (starting with lowest timestamp)) Loop
             If (slot timestamp not equal to last slot timestamp + 1) then
                     EnQueue as many empty slots to cover missing timestamps
                     for each (new slot to be Enqueued) Loop
                             Send NACK for missing timestamp
                             Mark slot state as Packet Requested
                     End Loop.
             End If.
             If (slot state is Packet Missing) then
                     Search DataQ for missing packet
                     If (packet is found in DataQ) then
                             Remove packet from DataQ
                             Place packet in OrderingQ
                             Mark slot as Packet Received
                             Attempt-Packet-Delivery(slot)
                             Update information about packet source
                     Else
                             Send NACK for packet
                             Mark slot as Packet Requested
                     End If.
             Else If (slot state is Packet Requested) then
                     Search DataQ for missing packet
                     If (packet is found in DataQ) then
                             Remove packet from DataQ
                             Place packet in OrderingQ
                             Mark slot as Packet Received
                             Attempt-Packet-Delivery(slot)
                             Update information about packet source
                     End If.
             Else If (slot state is Packet Received) then
                     Attempt-Packet-Delivery(slot)
                     Update information about packet source
             Else If (slot state is Packet Delivered) then
                     Update information about packet source
             End If.
     End Loop.
     while (the number of ACK Packets and New Lists Packets in OrderingQ
           is greater than the number of members of the Token Ring) Loop
             DeQueue lowest timestamp and discard packet
     End Loop.
End Update-OrderingQ.
```

Figure 3.3: Update-OrderingQ Algorithm

3. Update information about each packet source
    - The next *Expected* sequence number from that source
    - The next *Delivered* sequence number from that source
4. Attempt to deliver any packet which qualifies

The third item is probably one of the more important. Each Token Ring member must maintain information about each other member of the ring, and some non-members, for Multi-RPC to function correctly. As part of that information, two sequence numbers must be kept. The first sequence number is the next *Expected* from the member. This number is incremented each time an ACK is received for one of that member's Data Packets, or a New List Packet is received for one of that member's List Change Request Packets. It is this sequence number which is checked to determine if the current Token Site may generate an ACK containing a packet from that member. This provides a consistent mechanism which assures that ACKs only contain monotonically increasing sequence numbers from any one source. This restriction is also maintained for Non-Member Data Packets. The second sequence number is the next *Delivered* from that member. This number is incremented when a packet is delivered to the application from that source. This ensures that only monotonically increasing sequence numbers are delivered to the application from any one source.

The Attempt-Packet-Delivery algorithm, shown in Figure 3.4, takes as an argument the OrderingQ slot to attempt to deliver. If the slot meets the requirements to be delivered, based on the type of packet and the QoS of the packet if it is a Data Packet or Non-Member Data Packet, then the packet is eligible to be delivered to the application. There is one exception and this is for New List Packets which remove the site from the Token Ring. In order for a site to commit a New List Packet of this type, the site must also be in the *Not Token Site* state, see Subsection 3.4.3, as well as meet the other requirements.

```
Attempt-Packet-Delivery( slot )
    If (slot is a Data Packet or Non-Member Data Packet) then
            If (slot packet has QoS equal to Unordered) then
                Commit the packet to the application
                Mark slot as Packet Delivered
            Else If (slot packet has QoS equal to Source Ordered) then
                If (all of the smaller sequence numbers from that
                    source have been delivered) then
                        Commit the packet to the application
                        Mark slot as Packet Delivered
                End If.
            Else If (slot packet has QoS equal to Totally Ordered) then
                If (all of the timestamps smaller than the slots
                    timestamps have been delivered) then
                        Commit the packet to the application
                        Mark slot as Packet Delivered
                End If.
            Else If (slot packet has QoS equal to K Resilient) then
                If (all of the timestamps smaller than the slots
                    timestamps have been delivered and the token
                    has passed K number of times) then
                        Commit the packet to the application
                        Mark slot as Packet Delivered
                End If.
            Else If (slot packet has QoS equal to Majority Resilient) then
                If (all of the timestamps smaller than the slots
                    timestamps have been delivered and the token
                    has passed number of members/2 times) then
                        Commit the packet to the application
                        Mark slot as Packet Delivered
                End If.
            Else If (slot packet has QoS equal to Totally Resilient) then
                If (all of the timestamps smaller than the slots
                    timestamps have been delivered and the packet is
                    about to be Dequeued from OrderingQ) then
                        Commit the packet to the application
                        Mark slot as Packet Delivered
                End If.
            End If.
    Else If (slot is a New List Packet)
            If (all of the timestamps smaller than the slots timestamps have
                been delivered) then
                    Commit the New List and notify application
                    Mark slot as Packet Delivered
            End If.
    Else If (slot is an ACK Packet) then
            Mark slot as Packet Delivered
    End If.
End Attempt-Packet-Delivery.
```

Figure 3.4: Attempt-Packet-Delivery Algorithm

```
Add-ACK()
     EnQueue a slot into the OrderingQ with the same timestamp as ACK
     Mark slot as Packet Received
     for each (ACK Identifier in ACK Packet) Loop
             EnQueue a slot into the OrderingQ with timestamp of ACK + ACK Id
             Mark slot as Packet Missing
             Set information in slot to reflect ACK Identifier information
     End Loop.
End Add-ACK.
```

Figure 3.5: Add-ACK Algorithm

```
Add-New-List()
     EnQueue a slot into the OrderingQ with the same timestamp as New List
     Mark slot as Packet Received
     Scan DataQ for List Change Request Packet matching New List Packet
     If (a List Change Request Packet is found matching New List Packet) then
             Remove List Change Request Packet from DataQ and discard
     End If.
End Add-New-List.
```

Figure 3.6: Add-New-List Algorithm

The next two algorithms, Add-ACK and Add-New-List, essentially do the same thing, but do it on two different packet types. Add-ACK, shown in Figure 3.5, adds an ACK Packet to the OrderingQ and also adds slots for the Data Packets or Non-Member Data Packets associated with the ACK. By convention, the additional slots have timestamps that increase monotonically from the ACK Timestamp. Add-New-List, shown in Figure 3.6, adds a New List Packet to the OrderingQ and also tries to match it up with a List Change Request Packet from the DataQ. If it can make the match, then the List Change Request Packet is removed from the DataQ and discarded.

A careful observer may notice that duplicate ACK and New List Packets may be detected by first checking the timestamp and then searching the OrderingQ for that timestamp. Packets with timestamps less than the last delivered timestamp are most certainly duplicates, and the same is true of any timestamp which is already in the

```
Pass-Token()
     for each (member of the DataQ) Loop
             If (member is a List Change Request Packet and request can
                 be granted and packet is eligible) then
                     Generate a New List Packet for request
                     Send New List Packet
                     Exit Loop
             Else If (member is a Data Packet or a Non-Member Data Packet
                      and is eligible to be acknowledged) then
                     Generate ACK Packet containing as many Data Packets
                        and Non-Member Data Packets as are eligible in the
                        DataQ
                     Send ACK Packet
                     Exit Loop
             End If.
     End Loop.
     If (ACK Packet or New List Packet could not be generated) then
             Return to calling routine reporting Token Not Passed
     Else
             Return to calling routine reporting Token Passed
     End If.
End Pass-Token.
```

Figure 3.7: Pass-Token Algorithm

OrderingQ. For Data and List Change Request Packets, the detection of duplicates is only slightly more complicated. First the sequence number and source of the packet should be checked and compared with the information known about the source. Any sequence number less than the next expected sequence number from that source is a duplicate. Next a search of the DataQ may turn up that the packet is already in the DataQ. Then finally a scan of the OrderingQ can be done to determine if the packet has already been processed. Detecting duplicate Non-Member Data Packets is slightly more complex, see Subsection 3.4.2. This is just one way in which duplicate detection can be done. It is by no means the *most* efficient.

The Pass-Token algorithm does not perform any operation on the DataQ or OrderingQ, but is nevertheless essential. This algorithm, shown in Figure 3.7, is the mechanism which generates an ACK Packet or generates a New List Packet and sends the packet, therefore passing the token. Pass-Token does not have to generate a packet, however. As shown in the algorithm, only Data Packets and Non-Member Packets

which are eligible to be acknowledged can be acknowledged. Only List Change Request Packets that are eligible, and can be granted, may generate a New List Packet. A packet is eligible if its sequence number is the next expected sequence number from its source. Packets with a QoS of *Reliable* are always eligible and need not meet the the sequence number criteria. A request can *not* be granted only if the actual request operation is unknown or violates the semantics of the operation (e.g. asking to be added to the token ring when already a member).

3.4 Finite State Machine Representation of RMP

To facilitate the verification effort and to precisely specify the protocol operation, RMP has been broken down into a finite state machine representation. This state representation is meant to apply to each site, or member, of the token ring. So collectively, all the members of the group make up a global state that is a composition of all the individual member states.

The state machine is driven by *Events*. These correspond to packets arriving, timers expiring, and combinations of conditions being met. The different events and their corresponding descriptions are shown in Table 3.4. This state machine representation can have multiple transitions on one particular event and condition. This is shown in the tables by *Pass Event to Next State*. In the state tables presented, it is assumed that duplicate packets are detected and removed before being acted upon. The *Condition(s)* in the table apply after the *Action(s)* have been performed, but before actual state transition has taken place.

First the Normal Operation States are presented, then the extensions to this base model are shown and discussed. The extensions are Multi-RPC, Membership Change, and Reformation. Each extension builds on the previous model to enhance functionality. The entire state tables are presented in Appendix B on page 117. Only small

Event	*Description*
Data	Reception of a Data Packet
ACK	Reception of an ACK Packet
NACK	Reception of a NACK Packet
Confirm	Reception of a Confirm Packet
Non-Member Data	Reception of a Non-Member Data Packet
Non-Member ACK	Reception of a Non-Member ACK Packet
New List	Reception of a New List Packet
List Change Request	Reception of a List Change Request Packet
Recovery Start	Reception of a Recovery Start Packet
Recovery Vote	Reception of a Recovery Vote Packet
Recovery ACK New List	Reception of a Recovery ACK New List Packet
Recovery Abort	Reception of a Recovery Abort Packet
Transmission Failure	A Packet requiring positive acknowledgment has been retransmitted X number of times without receiving that positive acknowledgment
Token Pass Alarm	Expired Timer for mandatory Token Pass
Confirm Token Pass Alarm	Expired Timer for Confirm notification
Check Non-Members Alarm	Expired Timer for flushing Non Token Ring Members from Membership List (local change)
Random Timeout Alarm	Expired Random Timeout Timer
Mandatory Leave Alarm	Expired Mandatory leave timer
Commit New List	A New List Packet is committed
Join Request	Application requests to join a Token Ring

Table 3.4: Event Descriptions

portions of these complete tables are presented here.

3.4.1 Normal Protocol Operation

The normal operation of RMP can be broken down into four states. Each site of the token ring is in one of these four states at any one time during normal operation. The states are *Token Site*, *Not Token Site*, *Getting Packets*, and *Passing Token*. The sending of Data Packets is entirely asynchronous and does not depend on what the current state is.

The *Token Site* state consists of that site being the current token site. The state is characterized as waiting for the arrival of a packet that can then be acted upon to transfer the token. Upon a transition into the state, the site must do two things. First it must set up one of two different timers. If an ACK event has placed the site into this state, then the site must set up a Confirm Token Pass timer if the token ring is ready to go *quiescent*. If not then the site must set up a Token Pass timer. The conditions to determine if the ring is ready to go quiescent is if the last N ACK events have been Null ACK. This assures that the token is passed N times before the ring can go quiescent. This will commit all packets, including packets with QoS of *Totally Resilient* before the ring goes quiescent.

Secondly, the site must initiate the Pass-Token algorithm. If the algorithm does pass the token, then the site must enter the *Passing Token* state and cancel the timer it just set. If the algorithm does not pass the token, no action is to be taken. The timer is allowed to run. If an event occurs later that passes the token, the timer must be canceled. Events and conditions that initiate a transition out of this state and actions that are done as a consequence of events are shown in Table 3.5.

After sending a packet which transfers the token, the site must enter the *Passing Token* state. This state allows the site to wait until it has positive confirmation that the next token site has actually accepted the token. This confirmation can be achieved

Event	*Condition(s)*	*Next State*	*Action(s)*
Data	Token Passed	Passing Token	place Packet in DataQ Pass-Token
Data	Token *not* Passed	Token Site	place Packet in DataQ Pass-Token
ACK	Site named Token Site	Token Site	Unicast Confirm to Packet source
NACK	(none)	Token Site	Send any packets that were requested and present
Token Pass Alarm	(none)	Passing Token	Generate Null ACK Send ACK Packet
Confirm Token Pass Alarm	(none)	Token Site	Unicast Confirm to last Token Site

Table 3.5: Normal Operation (Token Site)

Event	*Condition(s)*	*Next State*	*Action(s)*
Data	(none)	Passing Token	place Packet in DataQ Update-OrderingQ
ACK	ACK Timestamp $\geq$ Last Token Pass Timestamp	Not Token Site	Pass ACK to Next State
Confirm	ACK Timestamp $\geq$ Last Token Pass Timestamp	Not Token Site	(none)
NACK	(none)	Passing Token	Send any packets that were requested and present

Table 3.6: Normal Operation (Passing Token)

in the form of a Confirm event or as another token pass. This positive confirmation does not cause the site to perform any actions, at least not directly. If the token pass was an ACK, then the site must pass that ACK into the next state, in this case the *Not Token Site* state, where the event will be acted upon.

The events and conditions associated with the *Passing Token* state are shown in Table 3.6. Notice the interaction between the *Token Site* and *Passing Token* states. Periodic retransmissions of an ACK from the old token site, now in the *Passing Token* state, prompts the new token site to send a Confirm to the old site to notify it that it indeed does have the token. Once the site transitions out of *Passing Token* state, through reception of a positive confirmation of the token transfer, the site enters the *Not Token Site* state. The events and conditions associated with this state are shown

Event	*Condition(s)*	*Next State*	*Action(s)*
Data	(none)	Not Token Site	place Packet in DataQ Update-OrderingQ
ACK	Site *not* named Token Site	Not Token Site	Add-ACK Update-OrderingQ
ACK	Site named Token Site OrderingQ consistent	Token Site	Add-ACK Update-OrderingQ
ACK	Site named Token Site OrderingQ *not* consistent	Getting Packets	Add-ACK Update-OrderingQ
NACK	(none)	Not Token Site	Send any packets that were requested and present

Table 3.7: Normal Operation (Not Token Site)

in Table 3.7.

The state which most sites spend a lot of time in is the *Not Token Site* state. This state is the state where a site is not the token site and is just receiving packets and processing them. Once the site is named as the next token site, it must process that token transfer, which is probably an ACK, and check to see if its OrderingQ is consistent. A consistent OrderingQ means that the site has received all contiguous timestamps and associated packets up to and including the token transfer that is making it token site. This includes any packets that are part of the token transfer. If its OrderingQ is consistent, then the site is eligible to become token site and may transition directly into that state, performing all the necessary actions. If the site is missing one or more packets in the OrderingQ, then it must enter a state called *Getting Packets*. This state and its events and conditions are shown in Table 3.8.

The *Getting Packets* state is a state that prospective token sites stay in until they have made their OrderingQ consistent. Each time a Data Packet or ACK Packet is received the site must check OrderingQ consistency. When the OrderingQ becomes consistent, a transition to the *Token Site* state may be performed, along with subsequent actions associated with such a transition.

There is one action that is not depicted in the tables. This action is performed each time a Data Packet arrives. Before the packet may be placed in the DataQ, its QoS

Event	*Condition(s)*	*Next State*	*Action(s)*
Data	OrderingQ consistent	Token Site	place Packet in DataQ Update-OrderingQ
Data	OrderingQ *not* consistent	Getting Packets	place Packet in DataQ Update-OrderingQ
ACK	OrderingQ consistent	Token Site	Add-ACK Update-OrderingQ
ACK	OrderingQ *not* consistent	Getting Packets	Add-ACK Update-OrderingQ
NACK	(none)	Getting Packets	Send any packets that were requested and present

Table 3.8: Normal Operation (Getting Packets)

is checked. Data Packets with QoS values of *Unreliable*, *Reliable*, or *Source Ordered* may qualify for delivery immediately. This qualification is the same as that used in the Attempt-Packet-Delivery algorithm. If a packet meets its qualification for delivery in this way, it may be delivered as well as put in the DataQ. Data Packets with a QoS value of *Unreliable* are not put into the DataQ and are always delivered immediately upon reception.

The four normal operation states along with the data structures and data structure algorithms provide the solid foundation that allows the three extensions to be built upon. It is important to notice that if only one member exists in the group, that the site need not ever transition out of the *Token Site* state. This is a trivial case since the site merely acknowledges that it received its own packet.

3.4.2 Multi-RPC Extensions

The inclusion of the Multi-RPC Extension does not add any states to the representation. It does, however, add some new events. These events are *Non-Member Data*, *Non-Member ACK*, and *Check Non-Members Alarm*.

The only thing that is necessary to do to provide Multi-RPC on top of normal operation is to handle Non-Member Data Packets in the same way that Data Packets are handled. This is fundamentally what is being done in Table 3.9. Two additional

Event	*Condition(s)*	*Next State*	*Action(s)*
	Token Site State		
Non-Member Data	Sequence Number $\geq$ *Expected* from source Token Passed	Passing Token	place Packet in DataQ Pass-Token
Non-Member Data	Sequence Number $\geq$ *Expected* from source Token *not* Passed	Token Site	place Packet in DataQ Pass-Token
Non-Member Data	Sequence Number $<$ *Delivered* from source	Token Site	Unicast Non-Member ACK to source
	Passing Token State		
Non-Member Data	(none)	Passing Token	place Packet in DataQ Update-OrderingQ
	Not Token Site State		
Non-Member Data	(none)	Not Token Site	place Packet in DataQ Update-OrderingQ
	Getting Packets State		
Non-Member Data	OrderingQ consistent	Token Site	place Packet in DataQ Update-OrderingQ
Non-Member Data	OrderingQ *not* consistent	Getting Packets	place Packet in DataQ Update-OrderingQ
	All States		
Check Non Members Alarm	(none)	(same state)	Remove all "Clients" that have "timed out"

Table 3.9: Multi-RPC Extensions

things need to be added however. The first is how and when to send Non-Member ACK Packets to the Non-Member Data sending site. And the other is the periodic flushing of "Clients" from the token list.

Sites that send Non-Member Data Packets to a token ring, usually expect a reply to their message. That reply can be in the form of an acknowledgment stating that at least one of the sites has delivered the message to its application, or the site that responds to the message may send back a reply. In the specifications here, we assume that all Non-Member Data Packets desire a response and that the acknowledgment comes in the first form. If a site does send a response to the message it does so in addition to the first Non-Member ACK Packet. This means that the site in effect will get two responses to its Non-Member Data message. The first is the notification of delivery. The second is the actual responding message. These two messages need not be sent from the same

token ring member. The site that generates and sends the ACK that orders the Non Member Data Packet in the OrderingQ is also responsible for sending the Non-Member ACK Packet to the message source when the message is committed to the application. This does not entirely capture the desired behavior of Multi-RPC. In Appendix A on page 113 there are two fields that are part of every Non-Member Data Packet that must be mentioned. These are the *No ACK* and *Multiple Copies* fields. These fields are bits marking how that Non-Member Data Packet should behave. The *No ACK* field specifically tells the ring members *not* to send a Non-Member ACK Packet for that Non-Member Data Packet. The *Multiple Copies* field specifically notifies the ring members to ignore duplicate detection for this Non-Member Data Packet. In effect this allows the packet to be delivered to the application multiple times. The first time that the packet is seen it will be acknowledged, ordered, and delivered. Subsequent receptions of the packet are immediately delivered to the application without the need to order or acknowledge the packet. The application then is responsible for generating a reply in the form of a Non-Member ACK Packet.

When a site not in the token ring sends a Non-Member Data Packet to the ring, it is added *locally* to each members token list. The member is marked as being a "Non-Token Ring Member" or *Non-Member*. Thus the token will never be passed to it. The reason for storing that information at all is to keep an active record of the sequence-numbered messages sent by the Non-Member site. If a token ring member is presently in the *Token Site* state and it receives a Non-Member Data Packet, then it has to examine the sequence number and source of the packet. If the sequence number is greater than or equal to what the next *Expected* sequence number is from the site, then the member knows that the message is valid and it may be put into the DataQ for processing. If the sequence number is less than the next *Delivered* sequence number from that site, then the member must send a Non-Member ACK Packet to

the site notifying it that the message was already delivered [7]. Notice that duplicate detection of Non-Member Data Packets can only be done on messages with sequence numbers that fall between the next *Expected* and the next *Delivered* from the site. These messages are most certainly duplicates and may be dropped.

In order to keep the token lists of the actual token ring members from growing out of control over time, periodically the Non-Members must be flushed out of the token list. This is done through the use of an timer, called *Check Non-Members Alarm.* When it expires, the site checks the times of the last known message from each of the Non-Members in its token list. If any of those times exceed the max allowable timeout period [8], then the Non-Member is removed from the list.

If a New List is generated while a Non-Member is still active in a sites token list, then the Non-Member *is* included in the New List and is marked as a Non-Token Ring Member. This brings new ring members up to date with past Multi-RPC operation.

3.4.3 Membership Change Extensions

The Membership Change Extension is meant to provide an efficient means of installing membership views to the group members. In order to achieve this, the addition of three new states and several new events are needed. The new states are *Not in Ring*, *Joining Ring*, and *Leaving Ring.* The new events introduced are *List Change Request*, *New List*, *Commit New List*, *Join Request*, *Mandatory Leave Alarm*, and *Transmission Failure.* The *Not In Ring* state is the initial starting state of all sites. Each site starts in this state and may never transition out of it if it never desires to join a token ring. It is this state that Multi-RPC uses for sending Non-Member Data Packets. The *Joining Ring* state is the state that is used as a waiting state while the token ring generates

[7] This assures that if the Non-Member never receives the actual Non-Member ACK from the responsible site, that the Non-Member will eventually be notified that its message has been delivered.

[8] Currently this is $2 * IPTTL = 2 * 256 = 512$ seconds

a New List and prepares to add a new member. The *Leaving Ring* state is that state that is used as a waiting state while the ring operates after a member is removed. The member can not actually be considered *out* of the ring until it is assured that the leaving member does not hold any packets that no other member does not have.

As shown in Subsection 3.2.3, New Lists can be installed as new membership views by a simple token transfer. By adding the List Change Request and New List events to the four normal operation states, as shown in Table 3.10, the operational model can be specified. In this way, the List Change Requests act in a way analogous to a Data Packet and the New List acts similar to an ACK. List Change Requests may be sent asynchronously just as Data Packets are transmitted.

A site may not directly start up in one of the four normal operation states. It must transition through the Membership Change additional states shown in Table 3.11. When an application requests to join a token ring, that site transitions to the *Joining Ring* state and sends a List Change Request asking to be added to the ring. If after a certain number of retransmissions there is no reply, then the site may safely form its own ring with itself as the only member. However, if the site receives a New List that also names the site the token site, then the site transitions directly into the *Token Site* state. This directly and immediately involves the site in acknowledging packets and perhaps changing membership views. Notice that the new member does not have a consistent OrderingQ, in terms of the other sites, but is allowed to go into the *Token Site* state. This is because the site really does have a consistent OrderingQ. From the site's point of view it is consistent because all it knows about, and all it should know about at this point, is that a New List was sent that added it to the ring. The site will not need to get any older packets. Thus Virtual Synchrony is maintained.

In the same way, a site may not directly leave the token ring from any of the four normal operating states. A transition through the Membership Change additional states is needed. When an application requests that a site be removed from the token

Event	*Condition(s)*	*Next State*	*Action(s)*
	Token Site State		
List Change Request	Token Passed	Passing Token	place Packet in DataQ Pass-Token
List Change Request	Token *not* Passed	Token Site	place Packet in DataQ Pass-Token
New List	Site named Token Site	Token Site	Unicast Confirm to Packet source
	Passing Token State		
List Change Request	(none)	Passing Token	place Packet in DataQ Update-OrderingQ
New List	New List Timestamp $\geq$ Last Token Pass Timestamp	Not Token Site	Pass New List to Next State
	Not Token Site State		
List Change Request	(none)	Not Token Site	place Packet in DataQ Update-OrderingQ
New List	Site *not* named Token Site	Not Token Site	Add-New-List Update-OrderingQ
New List	Site named Token Site OrderingQ consistent	Token Site	Add-New-List Update-OrderingQ
New List	Site named Token Site OrderingQ *not* consistent	Getting Packets	Add-New-List Update-OrderingQ
Commit New List	New List does not contain site	Leaving Ring	(none)
	Getting Packets State		
New List	OrderingQ consistent	Token Site	Add-New-List Update-OrderingQ
New List	OrderingQ *not* consistent	Getting Packets	Add-New-List Update-OrderingQ

Table 3.10: Membership Change Extensions

Event	*Condition(s)*	*Next State*	*Action(s)*
	Not In Ring State		
Join Request	(none)	Joining Ring	Send a List Change Request to join Token Ring
	Joining Ring State		
New List	Site is named Token Site	Token Site	Add-New-List Update-OrderingQ Commit New List Packet
Transmission Failure	(none)	Token Site	Form own Token Ring
	Leaving Ring State		
New List	Timestamp $\geq$ New List that removed site $+N$	Not In Ring	(none)
New List	Timestamp $<$ New List that removed site $+N$	Leaving Ring	(none)
ACK	Timestamp $\geq$ New List that removed site $+N$	Not In Ring	(none)
ACK	Timestamp $<$ New List that removed site $+N$	Leaving Ring	(none)
NACK	(none)	Leaving Ring	Send any packets that were requested and present
Mandatory Leave Alarm	(none)	Not In Ring	(none)

Table 3.11: Membership Change Additional States

ring, it sends a List Change Request asking to be removed. The removed site must operate normally until it commits a New List that removes it from the ring *and* the site is in the *Not Token Site* state. This may warrant that the site may have to delay committing the New List until it is in this state. The site then transitions into the *Leaving Ring* state. The site stays in this state until it notices that there have been as many token passes as there are new members in the ring. Then the site knows that it will not have any packets that the other sites may need. Upon seeing a token pass that meets this criteria, the site may then transition to the *Not In Ring* state.

Upon a transition into the *Leaving Ring* state, the site must set a *Mandatory Leave Alarm*. When the alarm expires and the site has not seen a token transfer that meets the criteria, then the site must go ahead and transition into the *Not In Ring* state. This is to ensure that failures do not deadlock a leaving member in the *Leaving Ring* state.

3.4.4 Reformation Extension

The RMP Reformation Protocol adds five new states to the specification and also adds several events, four packet reception events and one alarm event. The goal of the state representation is to specify a two step process. The first step is the generation and synchronization of a valid new token list. The second step is the installation of this new token list at each site. Before the process can begin a failure has to be detected. The additions to the normal operation states shown in Table 3.12 permit each site in the ring to transition from normal operation to the reformation states. The fault-detecting site, called the Reform Site, detects a failure by not receiving a response for a packet requiring positive acknowledgment after R attempts, shown as a *Transmission Failure* event in the tables. The reform site then enters the *Start Recovery* state and starts sending Recovery Start Packets.

A Recovery Start Packet contains the reform sites initial "vote" for the new list's

Event	*Condition(s)*	*Next State*	*Action(s)*
All Normal Operation States			
Transmission Failure	(none)	Start Recovery	Send Recovery Start Packet
Recovery Start	(none)	Sent Vote	Unicast Recovery Vote Packet to Reformation Site

Table 3.12: Reformation Extensions

highest contiguous timestamp and a version number for the new token list. This timestamp is the highest known delivered timestamp that the reform site knows about at the time. Due to varying QoS values of messages, this timestamp, called *SynchTSP*, must represent a timestamp with a QoS value of *Reliable* or better. If this timestamp is part of an ACK, then the SynchTSP is the timestamp of the last packet in the ACK. This is the *desired* synchronization point. Multiple failures and selected site failures may prevent this point from being reached. When this occurs, it is referred to as an *Atomicity Violation.* A list created in the presence of an Atomicity Violation is not immediately invalid. It is the application's prerogative as to whether the list is kept or not.

A site that is involved with a reformation, but is not the reform site, is called a *slave.* When a slave receives a Recovery Start Packet it must transition out of one of the four normal operation states and into the *Sent Vote* state. As part of this transition, the site must send a Recovery Vote Packet to the reform site. This vote contains the site's "vote" on the SynchTSP as well as the sites current point of synchronization. The current synchronization point is determined by the highest timestamp in the site's OrderingQ that is consistent.

Once the reform site and the slaves have entered the *Start Recovery* and *Sent Vote* states respectively, the reform site follows the events and conditions shown in Table 3.13. While the reform site is in the *Start Recovery* state it is attempting to bring itself up to the current SynchTSP. Any Data, Non-Member Data, ACK, or New Lists Packets received may advance the SynchTSP or bring the reform site synchronization

Event	*Condition(s)*	*Next State*	*Action(s)*
Data	(none)	Start Recovery	place Packet in DataQ Update-OrderingQ Update SynchTSP
Non-Member Data	(none)	Start Recovery	place Packet in DataQ Update-OrderingQ Update SynchTSP
ACK	Timestamp $\leq$ SynchTSP	Start Recovery	Add-ACK Update-OrderingQ Update SynchTSP
ACK	Timestamp > SynchTSP	Start Recovery	Add-ACK Update-OrderingQ Update SynchTSP to Packet Timestamp
New List	Timestamp $\leq$ SynchTSP	Start Recovery	Add-New-List Update-OrderingQ Update SynchTSP
New List	Timestamp > SynchTSP	Start Recovery	Add-New-List Update-OrderingQ Update SynchTSP to Packet Timestamp
Transmission Failure	Packet type was Rec. Start	Created New List	Create New List Send New List
Recovery Vote	Version is incorrect	Abort Recovery	Send Recovery Abort
Recovery Vote	Source in old List Vote MaxTSP > SynchTSP	Start Recovery	Update SynchTSP to Vote MaxTSP
Recovery Vote	Source in old List Vote MaxTSP $\leq$ SynchTSP	Start Recovery	Update Site Vote
Recovery Vote	Source in old List OrderingQ consistent Have Vote for each site Vote MaxTSPs = SynchTSP	Created New List	Create New List Send New List
Recovery Abort	(none)	Abort Recovery	(none)
Recovery Start	Source is *not* Reform Site	Abort Recovery	Send Recovery Abort

Table 3.13: Reformation Extension (Start Recovery)

point up to the same value as the SynchTSP. Any change to the SynchTSP as the result of a new vote from a slave or as a result of the reception of a new packet, prompts the reform site to restart the periodic retransmission of its Recovery Start Packet with the new SynchTSP value.

Only two ways exist for the reform site to leave the *Start Recovery* state. The first way is the site is able to obtain a Recovery Vote from each member of the old list, have all the votes synchronized to the SynchTSP, and have an OrderingQ that is consistent up to the SynchTSP. Under this condition a valid new token list has been formed and the reform site can then initiate the installation of the new list. The second way to leave the *Start Recovery* state is for the reform site to receive a notification that its Recovery Start Packet has been retransmitted the set number of times as required for failure detection. The reform site must then form a new list. If this list has enough members to meet the minimum partition size criteria [9] and the SynchTSP point has been reached by at least one slave or the reform site, then the new list is valid and reformation may continue as in the case above. If the minimum partition size criteria can not be reached, then the new list is invalid. The reform site must install the new list and then transition into the *Not In Ring* state. The last possibility is that the minimum partition size criteria is met, but there are missing packets which are preventing the list from being synchronized to the SynchTSP. In this case, the list is *assumed* valid, but the reform site marks the new list as containing possible atomicity violations. After reformation is complete, the application is notified of this. The application may then continue operation, or it may leave the newly reformed ring to restart again on its own.

The installation of a new list, whether valid or invalid, is accomplished by sending the New List until either it is retransmitted the set number of times for a failure to be

[9]Upon joining a token ring, a site specifies its vote for the minimum partition size for the ring. The actual minimum partition size is the highest of these votes.

Event	*Condition(s)*	*Next State*	*Action(s)*
Recovery ACK New List	Missing ACKs from 1 or more Sites	Created New List	Mark Source as ACK sent
Recovery ACK *New List*	Have ACKs from *All Sites* List is valid	Passing *Token*	Add-New-List *Commit New List* Send Null ACK
Recovery ACK New List	Have ACKs from All Sites List is *invalid*	Not In Ring	Add-New-List Commit New List
Transmission Failure	Packet was New List	Abort Recovery	Send Recovery Abort
Recovery Abort	(none)	Abort Recovery	(none)
Recovery Start	Source is *not* Reform Site	Abort Recovery	Send Recovery Abort

Table 3.14: Reformation Extension (Created New List)

detected, or an Recovery ACK New List has been received for each new list member. In the case of a valid list, the reform site then transitions to *Passing Token* and sends a Null ACK, passing the token to one of the members of the new list. The timestamp used for the Null ACK is equal to the $SynchTSP + 1$. An invalid list prompts the reform site to transition to the *Not In Ring* state and notify the application. The events and conditions of installing the new list are shown in Table 3.14.

At any time during this stage if the reform site receives a Recovery Vote or a Recovery Start Packet with the wrong version of the new token list, or if it detects another reformation occurring, then the reform site must abort the recovery. If the reform site retransmits the New List the set number of times for a failure to be detected and does not have a Recovery ACK New List from each member of the list, then it must abort the recovery.

The slave's point of view, shown in Table 3.15 and Table 3.16 , is much easier. The slave tries to become synchronized as best it can. If it receives an ACK or New List that has a higher timestamp than the SynchTSP, then it must notify the reform site of this. Likewise, at any time if the synchronization point of the site changes, then the site must notify the reform site. This notification is done by restarting the

Event	*Condition(s)*	*Next State*	*Action(s)*
Recovery Start	Source is Reform Site	Sent Vote	Unicast Recovery Vote to Reform Site
New List	Source is Reform Site Timestamp = SynchTSP + 1 Version is correct	ACK New List	Unicast Recovery ACK New List to Reform Site
New List	Source is Reform Site Timestamp = SynchTSP + 1 Version is correct List is *invalid*	Not In Ring	Unicast Recovery ACK New List to Reform Site
New List	Version is incorrect	Abort Recovery	Send Recovery Abort
Data	(none)	Sent Vote	place Packet in DataQ Update-OrderingQ Update Recovery Vote
Non-Member Data	(none)	Sent Vote	place Packet in DataQ Update-OrderingQ Update Recovery Vote
ACK	Timestamp $\leq$ SynchTSP	Sent Vote	Add-ACK Update-OrderingQ Update Recovery Vote
ACK	Timestamp > SynchTSP	Sent Vote	Add-ACK Update-OrderingQ Update Recovery Vote
New List	Timestamp $\leq$ SynchTSP	Sent Vote	Update Recovery Vote
New List	Timestamp > SynchTSP	Sent Vote	Update Recovery Vote
Transmission Failure	Packet was Recovery Vote	Abort Recovery	Send Recovery Abort
Recovery Abort	(none)	Abort Recovery	(none)

Table 3.15: Reformation Extension (Sent Vote)

Event	*Condition(s)*	*Next State*	*Action(s)*
New List	Source is Reform Site Version is correct	ACK New List	Unicast Recovery ACK New List to Reform Site
New List	Version is incorrect	Abort Recovery	Send Recovery Abort
ACK	Source is Reform Site Site is *not* named Token Site	Not Token Site	Add-New-List Commit New List Add-ACK Update-OrderingQ
ACK	Source is Reform Site Site is named Token Site OrderingQ consistent	Token Site	Add-New-List Commit New List Add-ACK Update-OrderingQ
ACK	Source is Reform Site Site is named Token Site OrderingQ *not* consistent	Getting Packets	Add-New-List Commit New List Add-ACK Update-OrderingQ
Recovery Abort	(none)	Abort Recovery	(none)

Table 3.16: Reformation Extension (ACK New List)

retransmissions of the Recovery Vote Packet and changing the values of the packet. Once a slave receives a New List that contains the correct version and has the site as a member, then the slave transitions to the *ACK New List* state and sends a Recovery ACK New List Packet to the reform site. If a slave receives a New List with the wrong version, then it must abort the recovery. If the set number of retransmission for failure detection is reached for the Recovery Vote Packet then the slave must abort the recovery. If a slave receives a New List that is marked as invalid, then the slave sends a Recovery ACK New List to the reform site and then transitions to the *Not In Ring* state.

Once a slave enters the *ACK New List* state it retransmits its Recovery ACK New List only when it receives another New List that matches the New List that put it into this state. A New List with an incorrect version prompts the slave to abort the recovery. When the slave receives a Null ACK from the reform site, it is sure that reform is done and the site can then transition to the appropriate normal operation state.

Event	*Condition(s)*	*Next State*	*Action(s)*
Random Timeout Alarm	(none)	Start Recovery	Send Recovery Start
Recovery Start	Version is correct	Sent Vote	Unicast Recovery Vote to Reformation Site
Recovery Start	Version is incorrect	Abort Recovery	Send Recovery Abort
Recovery Vote	Version is incorrect	Abort Recovery	Send Recovery Abort
New List	Version is incorrect	Abort Recovery	Send Recovery Abort
Recovery ACK New List	Version is incorrect	Abort Recovery	Send Recovery Abort

Table 3.17: Reformation Extension (Abort Recovery)

Whenever a site, reform or slave, receives a Recovery Abort Packet, or decides to abort the recovery themselves, they must transition into the *Abort Recovery* state, shown in Table 3.17. Upon this transition, a random timeout is set. The site that has this timeout expire first is the new reform site and starts the process over again, but the version number is increased by one for the list. The other sites follow upon receiving a new Recovery Start Packet with a subsequent canceling of there respective random timeout alarms. Any site that receives a Recovery Start, New List, Recovery Vote, or Recovery ACK New List while in the *Abort Recovery* state with an incorrect version must send a Recovery Abort Packet for that version.

3.5 Flow Control and Congestion Control

Because RMP uses a primarily NACK based error detection scheme, there is no direct feedback path through which receivers can signal losses through low buffer space or congestion. Reliable multicast protocols also suffer from the fact that throughput for a multicast group must be divided among the members of the group. This division is usually very dynamic in nature and therefore does not lend itself well to a priori determination. These facts have lead the flow and congestion control schemes of RMP to be made completely orthoganol to the protocol specification. This allows several

differing schemes to be used in different environments to produce the best results. As a default, a modified sliding window scheme based on previous algorithms for TCP [10] is suggested and described below.

Flow control and congestion control are treated as exactly the same problem in this modified sliding window scheme. A sliding window flow control scheme is an adaptive mechanism that attempts to maintain a constant window of packets in transit. Packets in transit are packets that have been sent, but have not been acknowledged yet. Ideally this window corresponds to the current level of available resources. Other predictive flow control schemes have been proposed and are currently under investigation. These schemes are more applicable to high latency *long fat* networks, such as ATM. These networks require that hundreds of packets be in transit at once, and the consequences of trying to adaptively size the transmission window when congestion occurs is much too high.

Some very good work has been done in providing efficient congestion control for TCP [10]. It is this work that RMP has partially adopted and expanded upon for its flow and congestion control mechanisms. The main four adopted points of the TCP work are:

- Round-Trip-Time Variance Estimation
- Slow Start
- Dynamic Window Sizing on Congestion
- Exponential Retransmit Timer Backoff

Round-trip-time (RTT) of a message is the time it requires for a packet to be sent and a corresponding acknowledgment to arrive at the sender. Round-trip-time variance estimations provide a means of determining how large timeout periods should be on retransmissions based on the average measured length of the round-trip-time and the deviation in the time. It has been observed that when network paths become

congested, the variance on packet latency becomes very high compared to it average [10]. It is hoped that by continually estimating the variance and adjusting the average, that an accurate timeout period can be calculated that will virtually eliminate all spurious retransmissions. The elimination of spurious retransmissions allows more bandwidth and processing time to be dedicated to actual useful work, as well as reducing the probability of a false failure detection. The calculation of the timeout period can be effectively done using the following formulas:

$$Err = M - A$$
$$A = A + g_A(Err)$$
$$D = D + g_D(|Err| - D)$$
$$rto = A + 4D$$

The g_A and g_D terms are gain terms. M is the round-trip-time measurement. A is the round-trip-time average. D is the round-trip-time mean deviation. And rto is the next timeout period length. Experimentation has shown that 0.0625 and 0.125 are good values for g_A and g_D, respectively.

The slow start algorithm is used to increase the window size from its initial size of one packet to the maximum window size that does not cause congestion. The window size is measured in Minimum Transfer Units, or *MTUs*. 1 MTU represents a set number of bytes of data in transit. The value of 1 MTU is configurable based on network properties. The slow start algorithm starts by initializing the allowable maximum window size to be 1 MTU. Each time an ACK is received for a packet the window size is incremented 1 MTU. It may not be obvious, but this increases the window size exponentially. The window size will increase from 1 to W on a latency L network path in $L \log_2 W$ time. Thus slow start actually increases the window size

[10] "If the network is running at 75% capacity...one would expect the round-trip-time to vary by a factor of 16 [10]."

fairly rapidly. Once a sign of congestion occurs, then the window must be reduced. After this first reduction, slow start is not used. But a congestion avoidance scheme is used. This scheme increases the window in a more linear fashion to hopefully avoid congestion. This is done by incrementing the window size by 1/(Window Size) each time an ACK arrives for a packet. A window size of W will therefore only generate at most W ACKs, and an increase of $1/W$ will increase the window by 1 in one round-trip-time. This increases the window size linearly. In this way, resource limits are probed, but not overrun too quickly.

Under the observations that most lost packets are the result of congestion and not errors and that retransmissions must signal lost packets, then any retransmissions, or expired timers for retransmissions, signal congestion. Congestion must decrease the maximum window size. RMP decreases its window size by 50% each time congestion is encountered. After this decrease the window increases using the linear increase presented above.

Each time a timer expires and a retransmission is needed, the exponential retransmit timer backoff scheme doubles the timer. Once an ACK is received for the packet, however, the value is set to the *rto* value as calculated. This scheme is applied to all packets that require positive acknowledgments. The timer value must be clamped at a certain maximum value, however [11]. This scheme attempts to ensure that false alarms occur *very* rarely and that alarms signalling retransmission themselves should not cause even more congestion.

Flow control is addressed by allowing NACKs to also signal dropped packets. Sites that are overrun by senders will drop one or more packets, and will have to send NACKs for those packets. The NACK control policy is to multicast the NACK to the entire group. Thus the sender will see that its packet was dropped and can reduce its window size exactly the same way it would in congestion control, by 50%. Care must be taken

[11] currently 2 seconds

not to perform this decrease multiple times for the same packet, however.

Chapter 4

Implementation of the Reliable Multicast Protocol

The first implementation of RMP has occurred concurrently with its design and verification. This not only allowed the constraints of the implementation to have more impact on the design, but the implementation in several instances has allowed the design to test theories and to attempt to provide truly what application developers desired. Implementation of a complex protocol such as RMP for the first time is not a trivial task. Many lessons have been learned. It is important to describe the first RMP implementation and to discuss some of these lessons.

4.1 Major Implementation Decisions

Implementing a complex construct, such as RMP, demands careful consideration of numerous factors. The three main factors involved in determining how best RMP should be implemented were:

- At what abstraction layer should RMP operate in?
- What kind of control paradigm should RMP use internally?
- How best to allow RMP to be easily extendable and to function as a testbed for new ideas for its continued evolution?

4.1.1 Implementing Protocols in the User Level

Traditional protocols have been implemented as monolithic entities in the operating system kernel or in a single trusted user-level server. The reasons for this have primarily been concerns over security and performance. Recently other important issues, such as ease of code maintenance, debugging, customization, and the need to have concurrent, multiple operating protocols, have prompted many to implement protocols in the more manageable user level [16],[9], [13]. Typically these implementations have taken the form of user-level libraries of protocols which can be linked into the application. It has been shown that traditional protocols can be implemented in user space and suffer no major degradation in performance [16]. On multiprocessors, it has been shown that implementation in the user level has actually increased performance.

With these facts in mind, it was quite easily agreed upon that the initial RMP implementation should be done at the user level and not in the kernel or as a single user-level server. This choice does not preclude other implementations of RMP from being implemented in the kernel or as a server process.

4.1.2 Event-Driven Control

Because RMP was to be implemented in the user-level as a library for use in applications, a control paradigm had to be chosen to reflect the diverse ways in which developers implement their own distributed applications. The paradigm chosen was an event-driven approach. RMP relies on events occurring and states being transitioned to and out of on specific event conditions, thus the event-driven approach for the implementation seemed very close and natural. It is also believed that the verification effort will benefit greatly from this once the emphasis shifts from design verification to actual implementation verification.

This approach has lead RMP to be able to provide two different control paradigms to the application using it. The first is the event-driven approach, where the applica-

tion constructs a control loop and passes control to RMP which takes care of protocol operation. This scheme is more useful for types of processes which wait for something to occur, or for applications which desire to have very explicit control over when RMP gets to perform its handling of internal events. The second approach is the implicit control approach, or the interrupt driven scheme. The application does not have to explicitly give RMP control, RMP can interrupt the application, handle its events, and then return control back to the application implicitly. This approach is useful for applications which don't want to give control to RMP in an explicit fashion, but would rather have RMP handle its events in the "background".

Performance concerns were a large factor in choosing an event-driven control paradigm. Receiving and ordering one message is equivalent to two events (one for the message to be received and one for the ACK to be received). In order to achieve a throughput of 300 messages/second, a total of 600 events/second needed to be attainable. 600 events/second leaves just 1.67 msec. per event for processing. This processing time must also include time spent in the operating system kernel performing network operations. Given modern operating systems and architectures this seemed viable and achievable. The RMP library also had to support multiple simultaneous token rings operating through the same application interface, so the need to classify events corresponding to token rings played a large part in deciding that an event-driven approach was worthwhile.

4.1.3 Object-Oriented Implementation

With flexibility and ease of adaptation to an evolving design being so important to the RMP implementation, it should not be surprising that an object-oriented language, more specifically C++ [6], was chosen to implement RMP. An object-oriented language has allowed the critical platform/operating system dependent parts of the code to be separated from the actual internal data structure easily. In fact, in changing RMP to

use another network interface library only one object would need to be changed. The current RMP implementation has been developed using C++ on several platforms [1]. This modularization technique has already proven to be beneficial to debugging, see Section 5.2.

4.2 The RMP Internal Class Structure

The RMP class structure is very flat. Almost no inheritance has been used to provide polymorphism. This is because of the event processing requirements presented in Subsection 4.1.2. True run-time polymorphism is presently very expensive in terms of performance. This is especially true if the methods and objects using it are frequently being invoked [2]. To avoid this performance penalty, the use of virtual methods has been avoided. The general structure and modules for the implementation were broken down into four basic categories.

- Static Objects
- *Communication*
- Control
- Application Programming Interface (API)

Because the implementation needed to support timer retransmits and other alarms, a global alarm mechanism was needed. To efficiently handle memory, several global pools of pre-allocated buffer space needed to be used. An event driven approach demands that the queue of pending events to be serviced be global to the implementation. These constructs are all implemented as global, or static, data in the implementation. This is usually not a desirable practice because any application using RMP would

[1] RMP was originally developed on Sun Microsystems SPARCstations under the SunOS 5.3 and SunOS 4.1.3 operating systems. RMP has also been updated to run on Silicon Graphics workstations under Irix 5.2 and Irix 4.0.5 operating systems. Other operating system and platform support is soon to follow.

[2] Some preliminary tests showed that it could triple running time of even a simple program.

have to contend with the memory space being taken up even when RMP was not really using it to its full potential.

Almost all RMP I/O operations [3] are concentrated into one module called the Communicator. It is this module that is used to actual perform low level network system calls to send or to receive a packet. The DataQ and OrderingQ data structures along with one other queuing structures make up the control module. These structures are responsible for the protocol operation, maintaining the information about members of the token ring, and queuing messages for flow control. Each process using the implementation must maintain a copy of these structures that are specific to each token ring that it belongs to. For interaction with the application developer, a whole set of objects have been designed for use. These objects compose the RMP Application Programming Interface (API). Also a C interface to this API has been designed and implemented to allow C developers to use RMP as well.

Primitives

Several different base objects needed to be constructed for the RMP implementation. Although basic in functionality, these objects had to be very efficient and reliable. The processing of one event would at least entail a system call to the operating system to retrieve the message from the network [4] and several operations on internal data structures. This lead to the conclusion that the basic data structures themselves needed to be efficient.

The base objects needed were:

- Double Linked List Element
- Normal FIFO (Queue)
- Priority FIFO (Queue)

[3] An exception is the API operations that set up specific control loop operation.

[4] Usually about 0.5 msec.

- Unordered List
- Pool [5]

Optimizing these base objects was not very difficult. But in an effort to provide a generic set of these primitives, C++ templates [6] were originally used to construct the objects. These template objects were later discarded as inefficient and costly in terms of executable program size. These two observations, coupled with the fact that templates, though being part of the ANSI C++ Standard, are not implemented by all C++ compilers, led to the eventual implementation of these primitives as base objects that could then be used to provide the basis for specific derived components of the needed data structures.

4.2.1 Static Objects

The global, or static, objects that the RMP implementation must maintain are the mechanism to handle the various alarms and retransmit timers, the various pools of pre-allocated buffers, and the event queuing object.

Alarm Class

The first RMP implementation has been implemented under 4.3BSD and System VR4 UNIX systems. In order to implement retransmit timers and alarms, the software signal facilities [6] had to be used. This imparts the only non-control sequence restriction that the implementation imposes on the developer. The program using RMP must use RMP to set any alarms or timers it may need. It should not set up its own signal trap for the alarms. This would cause unknown behavior for protocol operation.

[5] A Pool, in this case, is actually a set of pre-allocated objects, stored for later use. The need for this is evident in the fact that the time to allocate an 8K region of memory for use as a buffer can be on the order of 0.25 msec.

[6] The signal that is trapped is SIGALRM, the alarm clock signal.

The object that controls the alarms is called the Alarms class. It contains an ordered queue, Ordered FIFO, of pending alarms ordered by the time when each expires. Each member of the queue holds information as to its corresponding token ring, its original expiration timer value, its time until expiration, and the type of alarm. All retransmit timers are grouped under one type of alarm. When an alarm expires, the alarm is placed on the queue of pending events where it will eventually be serviced.

Queue Structure Pools

For efficiency reasons, all the queuing structures primitives needed to use pre-allocated memory to store unused buffers. The pool sizes are initialized to reasonable values upon start up and are then used as needed. If for some reason the pool becomes empty and a new buffer is needed, it is allocated from system memory. Therefore under normal operating conditions, buffers are reused for the queuing structure elements, and if load becomes too heavy the system allocates more memory to handle the desired load. The whole reason for this is that the time to allocate memory can be quite high depending on the size of memory desired and the condition and fragmentation of memory at the moment. The buffer space for all of one type of pre-allocated element is stored together and the pool itself for that object is static in nature. This is necessary for efficient garbage collection and reclamation of multiple objects spread across multiple token rings.

PacketPool Class

As is done for queuing structure elements, the buffer space for packets is also pre-allocated and pooled for use as is needed. Packet buffers have their fixed header set to default values when the packet is placed back in the pool. This is to ensure that the vital information in that header is either valid because it is filled in or it is invalid

because the default values were still left after it was pulled from the pool for use.

EventQ Class

This RMP implementation is an event driven scheme. Events can be one of two things, either expired alarms or incoming packets. Outgoing packets and requests from the application are handled asynchronously to the event driven machine. This has allowed the state machine representation to transition directly into the implementation with very minimal modification.

The object responsible for ordering various events between various token rings, event types, and arrival times is the EventQ class. This object contains an ordered queue, Ordered FIFO, of pending events to be serviced. Each event contains information about that event relating to the event type, the token ring corresponding to the event, and an associated packet for the event. The need to make the queue global is for efficiency. Another approach would be to make several EventQs, one for each token ring. This would require some other mechanism to service each EventQ in some sort of order, and impose unnecessary delays in event servicing between token rings.

Events are placed on the EventQ from one of two sources. Expired alarm events are placed on the queue by the Alarms class, and incoming packets are placed on by the Communicator class. Each alarm type and packet type have their own corresponding event type. The EventQ orders the servicing of each of these events based on event types. Each type is given a certain priority and the queue is ordered first by priority then in FIFO order. The precedence of events based on type are shown in Table 4.1. The lower priority events are given more priority than higher priority events. The goals are to reduce as much as possible any race conditions that may arise, such as receiving an ACK for a packet before servicing the retransmit timer associated with it, and to optimize the events/sec. throughput of the system via modifying the event priorities.

Priority	*Event Type*	*Priority*	*Event Type*
-1	Transmission Failure	10	NACK
0	Recovery Abort	11	Non Member ACK
1	Recovery Start	12	Retransmit Alarm
2	Recovery Vote	13	Confirm Token Pass Alarm
3	Recovery ACK New List	14	Token Pass Alarm
4	Data	15	*Internal Reserved*
5	Non Member Data	16	*Internal Reserved*
6	List Change Request	17	Check Non Members
7	New List	18	Mandatory Leave Alarm
8	ACK	19	Random Timeout Alarm
9	Confirm	20	*Internal Reserved*

Table 4.1: Event Precedence

4.2.2 The Communicator Class

Most of the interaction with the system network I/O is concentrated into this module. By modularizing this functionality into a single object, RMP can be ported to various architectures easier and it can be tested easier. The portability issue is obvious, but the testing issue may not be. By making the Communicator object the connection between the network and the rest of RMP, a test scaffolding may be connected to the Communicator instead. In this way the state machine correspondence can be analyzed along with a set of test scenarios to determine the correct operation. The use of this test scaffolding is briefly discussed in Subsection 5.2.

4.2.3 Control Classes

A set of objects make up the control structure of the RMP implementation. Because operation of the protocol centralizes itself around an abstraction called a token ring, it is easily seen that the implementation should centralize operation around a token ring object. A DataQ and an OrderingQ must be part of each token ring along with a structure to hold member information and a queuing structure to hold packets for flow control.

ToBeSentQ Class

Flow control must sometimes queue packets for transmission because the current transmission rate is slower than the rate at which the application is trying to send. When this occurs the ToBeSentQ object queues the packets in FIFO order to be transmitted as flow control allows.

TokenRingInfo Class

It is necessary for the implementation to cache information about each member in the token ring. At the very least, the next expected and next delivered sequence numbers from each site must be maintained to determine when a packet is eligible to be delivered to the application and to maintain protocol operation. In addition, a list of currently active Token List IDs must be maintained. When a New List is committed it also changes the current Token List ID. The old Token List ID may be used by other members of the token ring because they have not seen the New List or they have not yet committed it. To allow for this case, the old Token List ID must be kept and packets containing it must be considered valid until it is certain that everyone in the token ring has the New List. This is accomplished once the token has rotated once around the ring after the New List was sent. Because the ring can be arbitrary in size and because there may be up to as many New Lists in the OrderingQ as there are members in the ring, there may be up to as many Token List IDs floating around as there are members in the token ring at any one time.

Another piece of information kept is the collection of reformation information that the reform site must maintain during different stages and steps of reformation. This information is composed of data about each site participating in the reformation. The sites synchronization point, the sites maximum timestamp, the sites minimum size requirement vote, and information about the site acknowledging the New List are all pieces of information that must be kept current during the reformation process.

TokenRing Class

The central object to this implementation is the TokenRing. This object holds a DataQ, an OrderingQ, a TokenRingInfo object, and a ToBeSentQ object. The TokenRing object also has a corresponding Communicator object that it uses to communicate with the network I/O system. A TokenRing services all events that are destined for it. That is to say, each event causes a specific token ring to handle that event. There is no concept of an event that may have multiple token ring destinations. Each TokenRing has an associated state that it must maintain. This state behavior follows the state machine defined in Chapter 3.

4.2.4 The RMP Application Programming Interface

The TokenRing object handles the protocol operation aspects of the implementation, and the RMP Application Programming Interface (API) handles the application interface aspects of the implementation. Full documentation of the RMP API is available. Only a brief description is given in this document.

RMP Class

The actual RMP object is a global (static) object that acts as a shell to hold a collection of other API objects. The RMP object also provides a simplified interface to the internal objects of the implementation, such as the Alarms object and the pre-allocated pools. Other RMP operations such as sending Multi-RPC messages and requesting to join token rings are done through this object.

RMPGroup Class

A TokenRing object always has a corresponding Communicator object that it uses as its interface to the network. The collection of both of these objects together actually demonstrates what an RMP Process Group really is when seen from a sites perspective.

The application should view the RMPGroup object as the actual RMP group that it is a member of. Thus when the application desires to send a message to a group, the object used is the corresponding RMPGroup. The same thing also is done for requesting and releasing handlers and locks.

RMPEvent Class

When the RMP internals decide to commit a packet to the application, the internals generate an RMPEvent object that is given to the application. This object contains the packet associated with the event and allows the application to read the information contained in the packet. The kinds of notifications and packets that the application may receive as RMPEvents are:

- Message reception
- Notification of receiving or releasing a Handler or Lock
- Notification of rejection of Handler or Lock requested action
- Notification of joining or leaving a group
- Notification of *membership view changes*
- Notification of atomicity violations and successful reformations

Most notification types of RMPEvents also return a token list so that the application can determine and account for any change of the membership view of the group internally to its own operation. This provides very powerful mechanisms that applications can use for process group management at execution time.

RMPMessage Class

In an ongoing effort to be as efficient as possible, the RMPMessage object was created. This object contains a pre-allocated packet that can be filled in by the application directly. Once filled in the packet could then be sent and another RMPMessage requested. In this way the overhead of copying the applications message to the RMP

internals can be avoided. For some applications this may decrease the latency of some messages, as well as, providing the application with a convenient garbage collection mechanism.

4.3 Portability and Optimization

For an protocol implementation to succeed it must not only be usable and functional, it must also be widely usable on a variety of system and efficient. Therefore portability and optimization become very important and pressing issues very early on in the development of a protocol implementation.

Portability is a very well understood problem. It is a problem that will continue to be explored and researched as systems become even more diverse and complex. The first RMP implementation has been designed to meet this problem head on and to keep it in mind during all aspects of development. The first area to help reduce the complexity of porting the implementation to a new platform or system has been the enclosure of network specific interfaces into a single object. This has already proven useful in introducing support of RMP on some 4.3BSD and System VR4 systems. The second area is separating the system and platform specific system calls into a set of C macros. This allows easy changes to a set of macros instead of changing core pieces of code to support a new system. Within a few months the RMP implementation should be supported on the most widely used multicast supported systems as well as a few non multicast supported systems. Future implementations of well known systems such as Microsoft Chicago are planning to support IP multicast technology. This paves the way for the implementation to support these systems as well.

Protocol implementation optimization is a very heavy area of activity. A large body of work has been done in the area of optimizing such heavily used protocols as TCP and UDP. The key points that one can learn from these efforts are minimize operating

system kernel involvement and maximize usefulness of each instruction. Even in the implementations early phases attention has being paid to how to optimize each and every piece of operation. Enough can not be said of the advantages of profiling the source code. Trouble spots are quickly drawn to attention and can be enhanced. As this implementation is being done in the user level, time spent in operations at the kernel level, such as sending and receiving network messages, is very costly. These operations need to be done only when needed and done efficiently. One thing that helps to accomplish this is the exploitation of functional overlap. This practice is also done in the protocol design itself. An example is providing the varying QoS levels. Each level builds upon one another and can be accomplished by using the semantics of the previous levels to perform the operation in an efficient manor. Another example is the support of Multi-RPC operation. Non-Member packets are treated exactly as regular Data packets, thus negating the need to specifically engineer another mechanism into the protocol to provide the concept. The first RMP implementation is far from being optimal at this time. It will be an ongoing task to eliminate the bad ideas and experiment with the new to hopefully provide an ever improving implementation.

Chapter 5

Verification of the Reliable Multicast Protocol

This chapter discusses the verification aspects of RMPs development and continued evolution. While the design and implementation of RMP have progressed concurrently, so has the verification effort of RMP progressed as well. The purpose of the verification effort is not to certify the protocol, but rather to debug the algorithms and to provide feedback to the design and implementation activities.

5.1 Verification Approaches

Because of the limitations of testing and simulation tools for debugging distributed applications, the verification effort has focused on formal modeling of the protocol [24]. It is believed that the verification effort will also be able to help in clarification of problems as the protocol implementation is used and "bugs" are discovered by users. Most problems of this nature are the result of concurrency or synchronization aspects that may impact the protocol design.

Upon starting the verification effort a great deal of attention has been paid to examining and finding an appropriate and powerful set of tools for modeling RMP. The criteria for choosing the tools has been:

Incremental: The approach should allow the construction of the model to progress from small, well understood pieces to larger, more complex constructs.

Automated/Semi-Automated: Tool support should be available if the approach is sufficiently complex.

Ease of Understanding: The approach should be "mainstream" in the formal methods community.

Handles Concurrency: The approach should be able to account for concurrent operation of multiple systems and the state explosion involved therein. This may be through either a direct or an implied mechanism.

Handles Nondeterminism: The approach should be able to handle situation where one of a multiple combinations of events can occur. This may be through a direct or implied mechanism.

Background: The technique should have a long history of real world substantive examples and should have a large experience base.

The incremental approach can be satisfied by many techniques. The concurrency and nondeterminism issues, however, can not be. It is very difficult to model these properties except implicitly. Some techniques were found that can handle these issues sufficiently.

It is important to keep in mind that verification of the source code is not and should not be the prime target of the verification effort. Maintaining fidelity between the verification model, the implementation, and the design, and examining and questioning the theory and model of the protocol are the main advantages to formal verification.

5.1.1 Symbolic Model Verification

One of the first approaches examined was the Symbolic Model Verification (SMV) system [5]. SMV is a technique for checking finite state systems, from completely synchronous to completely asynchronous, against the system specification expressed in a temporal logic Combinational Tree Logic (CTL). SMV allows for nondeterminism and concurrency. SMV directly attempts to model this by specifying state transitions explicitly.

SMV has been used very effectively in hardware verification. However, modeling RMP in SMV proved to be very difficult. The first problem was that SMV needed

very verbose specifications to capture concurrency and nondeterminism. To capture these properties, the SMV model became very disconnected from the RMP protocol specification. This was not intentional but necessary to be able to tackle the problem using SMV. The second problem was that, although SMV showed remarkable success in modeling gross state behavior of RMP, it has yet to show success in modeling the data structures. The reason for this is that to model state transitions on elements of aggregate structures, explicit and verbose specifications need to be done. This seemed to prolong the running time of the tool to an unacceptable level [1].

5.1.2 Murφ

An approach similar to SMV that has been examined is the approach taken by the tool called Murφ [8]. Murφ uses a high-level description language to describe finite-state asynchronous concurrent systems. Murφ does not suffer from the verbose specifications needed for SMV because it has many high-level language constructs, such as user-defined data types, procedures, and parameterized descriptions. Murφ descriptions consist of a set of transition rules that may contain procedures, global variables, data types, etc. The verification comes into play by the use of specifying *invariants* that should hold true for all individual states.

These specifications are evaluated and used to generate C++ code that can be run to test the invariants and the specifications. Assertions and deadlock testing can also be done. Murφ does its testing by enumerating all possible state combinations and checking for violations and errors. This allows Murφ to handle nondeterminism very well.

Because Murφ uses a combinational approach instead of a CTL approach it has much better running time than SMV. But Murφ does not attempt to test every possible path of execution, only portions of that path. This seems to be good enough for

[1] One execution was aborted after a 14 day running time.

evaluating the RMP state machine representation.

5.1.3 Prototype Verification System

Another quite different approach is the Prototype Verification System (PVS) [25]. PVS is an assisted proofing tool. It is not totally automatic. In fact, it is very mechanized and almost all the work of the specification and proof is left to the user. PVS does support very good modeling of abstract data types, and concurrency and nondeterminism can be modeled by the characterization of groups of states instead of expressing state transitions explicitly.

The developers of PVS have also developed several useful theories for use in PVS proofs that seem to directly apply to proofs for RMP properties. The first is the idea of a sequence. The second is the idea of behaviors. Behaviors are non-decreasing, non-zeno sequences. Another useful property is the since operator and its supporting lemmas. This operator allows properties such as safety and utility on RMP, the data structures, and supporting elements to be addressed. Time bounds also may be placed on specific properties.

The learning curve of PVS is very steep. The proof of even simple things can be quite involved and time consuming for beginners to the system. However, PVS has been used on some very intensive applications to great benefit.

Another approach that is being combined with PVS for the RMP verification effort is modal functions called *modal primitive recursive* (mpr) functions [27]. It has already been shown that the normal operation of TRP can be verified using this method [28]. This approach uses functions, sets, and relations to show that properties hold. The combination of these two approaches, PVS and mpr, seems to hold a lot of promise in verification of complex protocols like RMP.

5.2 Case (Scenario) Based Testing

An area of system exploration that is frequently glazed over too often is the importance and benefit of real world functional testing. This testing has been done in RMP. The RMP implementation was constructed from the ground up and tested at each step of the way by an ever expanding set of tests. After the event driven subsystem was developed and completed, an interactive debugging tool was developed to replace the Communicator object to provide an interactive or script based tester called the RMP debugger.

The RMP debugger allows the developer to test all aspects and characteristics of the internal state machine by creating packets and/or alarms and placing them in the state machine and examining the changes that are initiated as the machine executes in a controlled fashion. In this way, cases, or scenarios, can be created to test specific state behaviors or combinations of events. This has proved to be invaluable in developing and finding implementation specific problems that would have not been caught at such an early stage if such a tool were not available.

Chapter 6

Performance Results

This chapter presents the performance achieved by the first RMP implementation. Theoretical performance is discussed in the first section. The second section presents performance observed on a 10 Mbps Ethernet. The last section briefly discusses performance issues as have been observed on a portion of the Internet MBone.

6.1 Theoretical Performance of Model

Performance is a key issue in the development of a communications protocol. The performance a protocol eventually achieves is bounded by the design of the protocol. RMP is designed such that the delivery time for a totally resilient message is $O(N)$, where N is the size of the token ring. The delivery time for an unreliable message to be delivered to any one group member is $O(1)$. The delivery time for a reliable and totally ordered message to any one group member is also $O(1)$. Message stability, the time it takes for a message to be delivered to every member of the group, is $O(N)$.

Under normal situations with low error rates and normal to high traffic, TRP requires very close to two multicasts per message. As the traffic level decreases, this value increases to two multicasts and a unicast to confirm the token transfer. As the error rate increases, the number of packets sent increases, but it is always lower than that required with a normal positive acknowledgment scheme for groups of three or

more sites including the sender [3]. RMP expands on this by providing that under heavy traffic, the number of multicasts per message starts to drop. This improves throughput performance. TRP delivers packets to the application only after N token transfers after the message is received. RMP improves packet latency in this case by allowing varying QoS levels to vary this latency on a per packet basis.

Under the worst case, the network utilization of RMP in terms of data and space used by headers is for a two multicast per message case. The header of every RMP message is at least 24 octets. This is the RMP Fixed Header. In addition to this header, each additional type of RMP packet has an additional header. The header for a Data Packet is 24 more octets. And the header for an ACK Packet is 32 octets. In addition, an ACK may have one or more 24 octets worth of information. Under the worst case, where one Data Packet and one ACK Packet are needed, the total space needed for headers is therefore 128 octets. For an Ethernet, which fragments packets beyond 1512 octets in size, an additional 14 octets are used, so for a 1512 octet size message, the network utilization is (1526 - RMP Headers - 14)/1526 = 90.7%. For a typical 10Mbps Ethernet, RMP would have a theoretical maximum throughput of 1130 KB/sec. or 0.907 Mbps. This seems to be quite acceptable for the class of applications that RMP is designed to support.

6.2 LAN Performance

Actual performance of the protocol implementation is sometimes difficult to obtain accurately. The most difficult obstacle is finding a period of time to perform testing when the network is at its most restful state. This is especially true of testing a protocol that is designed to perform at the level of RMP.

All tests were run on several Sun Microsystems SPARCstation 5's running SunOS 5.3 operating system on a 10Mbps Ethernet. The network was operating under light

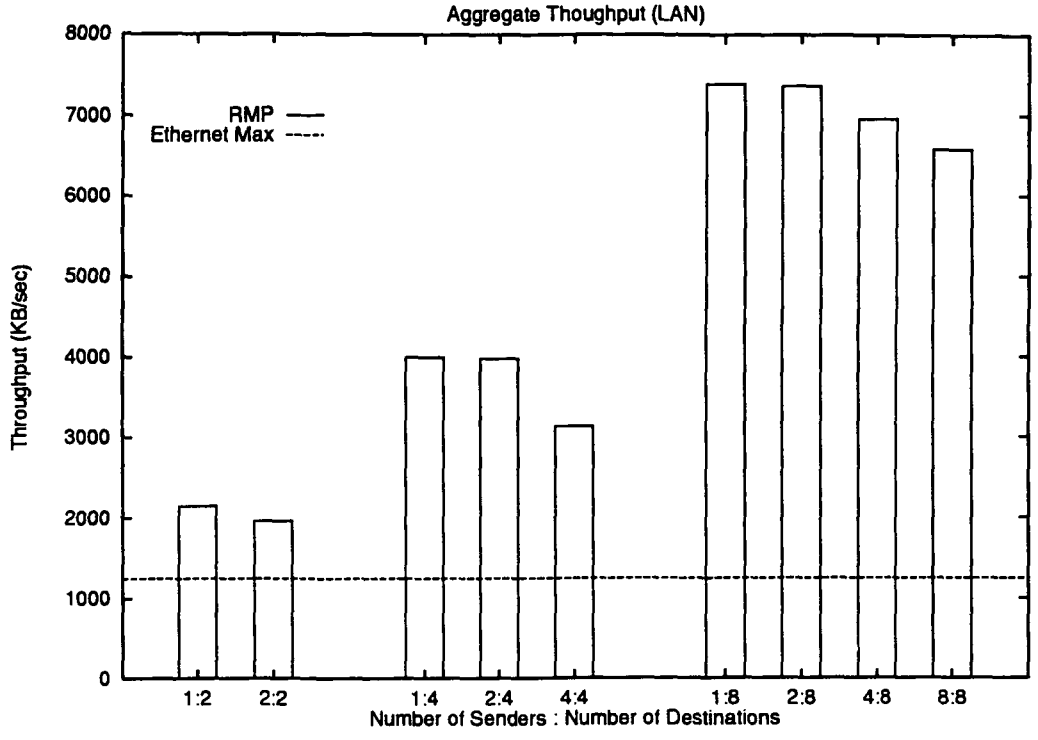


Figure 6.1: LAN Aggregate Throughput

load before and after the tests. Each test constituted of ten trials and the average of the ten trials is shown. Throughput measurements were taken by timing transfer of 5 megabytes of data and does not include packet headers. No fair comparison with other reliable multicast protocols can be made using these tests as performance numbers from other protocols have been collected on varying types of platforms and operating systems. However, no other reliable multicast protocol has been able to show as high throughput performance as RMP.

6.2.1 LAN Aggregate Throughput

Aggregate throughput is the throughput the application sees. It is computed by taking the amount of data sent from all of the senders and multiplying it by the number of destinations. A sender is also a destination. Figure 6.1 shows aggregate throughput

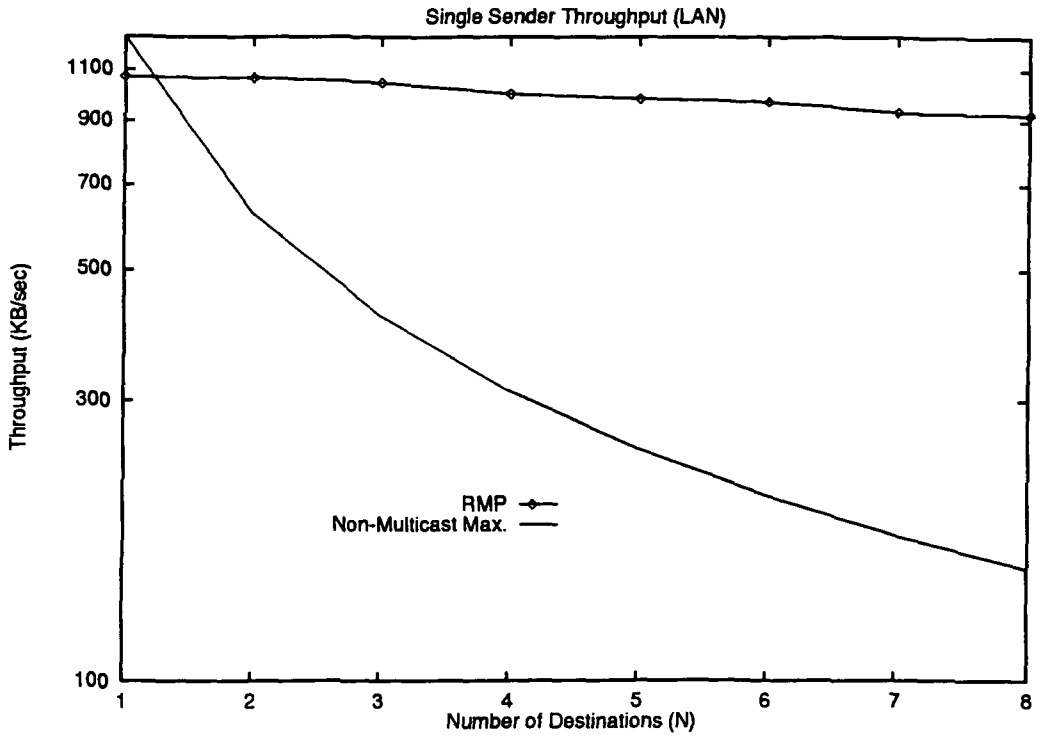


Figure 6.2: LAN Single Sender Throughput

for RMP with a variety of configurations. The QoS of each packet was *Totally Ordered.*

This is the case that most applications use totally ordered multicasting for, and it is actually the worst case for the throughput of the protocol because of the load on the senders. It is not possible for a non-multicast or non-broadcast scheme to break the Ethernet throughput boundary shown in Figure 6.1.

6.2.2 LAN Single Sender Throughput

Single sender throughput is shown in Figure 6.2. This is the aggregate throughput divided by the number of destinations. The sender is *not* counted as a destination in these tests. The non-multicast or non-broadcast maximum throughput is shown. This drops off as a factor of $1/N$. This limitation is a fundamental limitation of the network. As can be seen, RMP stays roughly constant as the number of destinations

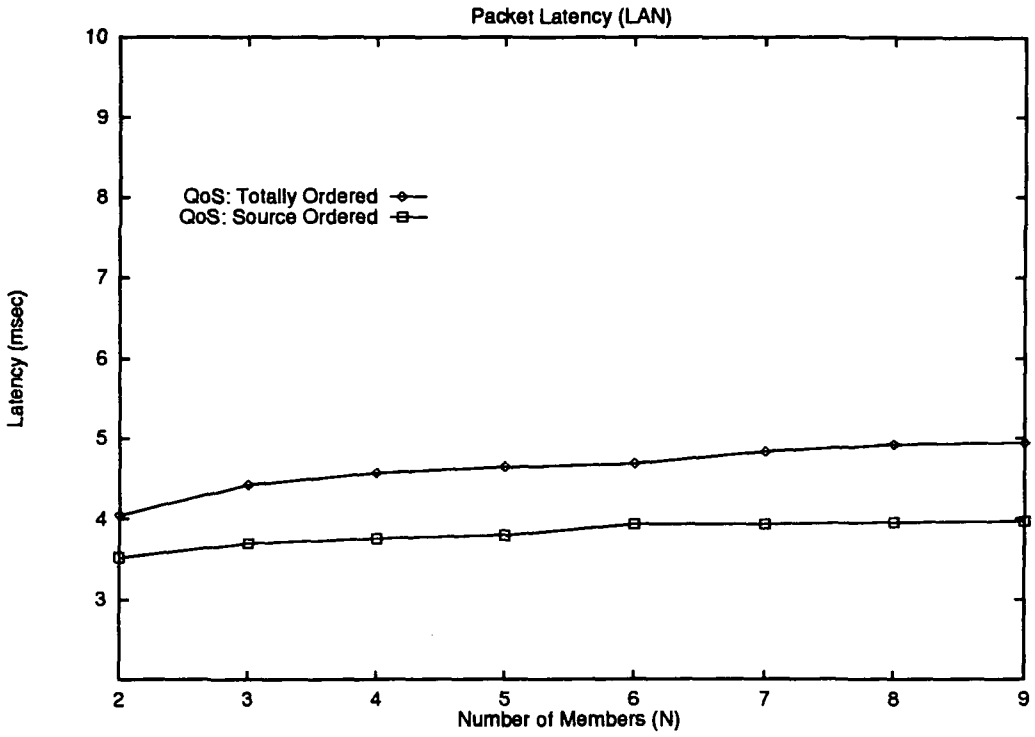


Figure 6.3: LAN Packet Latency

increases. The QoS of packets does not directly impact throughput. In these tests throughput was measured using packets of *Totally Ordered* QoS. Similar tests have shown the same throughput for QoS ranges from *Source Ordered* to *Totally Resilient.* The latency of packets, however, is very much effected by the packets QoS.

6.2.3 LAN Packet Latency

The latency of different QoS packets as the number of token ring members varies is shown in Figure 6.3. Packet latency was measured by timing the transmission of 10,000 minimum length messages [1]. Flow control was disabled and a stop-and-wait scheme was used to make sure that only one packet from all the members was in transit at any one time. In this scheme, latency is equal to the reciprical of the number of messages

[1]RMP Fixed Header + RMP Data Header = 48 octets

per second.

RMP performance can be seen to stay very constant as the number of destinations increases. Protocols that do not take advantage of multicast or broadcast scale linearly as a function of the number of destinations. Notice the difference that differing QoS levels introduce.

6.3 WAN Performance

It is very difficult to come up with respectable numbers for WAN performance given the state of the current MBone. The majority of mrouters are not designed to be routers and usually do not reside on routing dedicated machines. Therefore all multicast traffic the router sees must be reflected back unto the network and unicast through the mrouting tunnels. It is obvious that throughput for RMP on the MBone will be restricted by the slowest mrouter in the topology section being used. Therefore 90% or slightly more of the MBone available bandwidth would be the maximum seen. Tests have been run to verify this theory and they concur on numbers of destinations of up to eight. Packet latency however is a very real issue on long haul networks. Independent tests have been run to examine MBone latency when RMP was used as the ordering mechanism. It has been observed that on two interconnected Ethernets the latency for a QoS *Totally Ordered* packet maintains steady values of approximately 9.5 msec. as long as the load is kept symmetrical. For asymmetric loads, where one Ethernet contains more than half of the sites in the group, the latency was seen to vary from less than 7.0 msec. to over 10.0 msec. The variance is most likely due to link congestion and bursting traffic.

Chapter 7

Conclusions and Future Work

This chapter draws some conclusions about RMPs design, implementation, and verification. It presents the planned extensions to RMP, the RMP Extended Architecture, and outlines some future research for RMP.

7.1 Conclusions

Presented in this document are the Reliable Multicast Protocol, the design aspects of the protocol, the implementation notes for the protocol implementation, and the protocol verification approaches. To summarize:

- Total ordering of messages can be achieved efficiently by distributing ordering responsibilities among group members.
- Resiliency and fault-tolerance levels can be chosen by the application. The cost associated with such flexibility is very reasonable.
- Functionality of a protocol does not need to be completely *orthoganol*, or independent. Significant advantages can be attained by basing additional functionality upon solid basic primitives.
- Concurrent protocol design, implementation, and verification can be done with great benefits.
  - Concurrent design and implementation allows implementation restrictions to be made clearer and handled earlier than normal software life-cycle models.

C-2

- Concurrent verification during design and implementation allows a close fidelity between model, specification, and implementation to be maintained. This has a direct impact on noticing problems early and developing reasonable alternatives.

- Monolithic implementations of network protocols in operating system kernels impose undue load on the operating system and jeopardize protocol evolution. Alternatives, such as user-level implementations, are increasingly becoming very usable and preferable.

- Flow and congestion control of multicast protocols deserves and requires serious research, especially in the WAN environment.

7.2 Future Work

Acceptance of RMPs basic operating concepts and functionality is occurring very rapidly. Suggestions on usability, improvements, and extensions are being proposed. The first implementation is in the process of forming the foundation for numerous government sponsored projects, among them are a distributed battlefield simulation project and several distributed database projects.

RMPs design, development, and verification activities are planned to be ongoing long term activities. The design, implementation, and verification aspects presented in this document are just the start to an ever growing and hopefully prosperous project. To continue to be excepted and used, RMP must surely evolve and improve. In order to evolve, RMP must clearly define its boundaries and limitations. In doing so, other questions have been raised. The *RMP Extended Architecture* attempts to answer some of those questions. Every aspect of RMP (design, implementation, and verification) has definite directions that are being pursued.

7.2.1 The Extended Architecture

RMP has been designed to provide a base layer of functionality which can then be built upon to provide a larger array of features. RMP as presented in this document

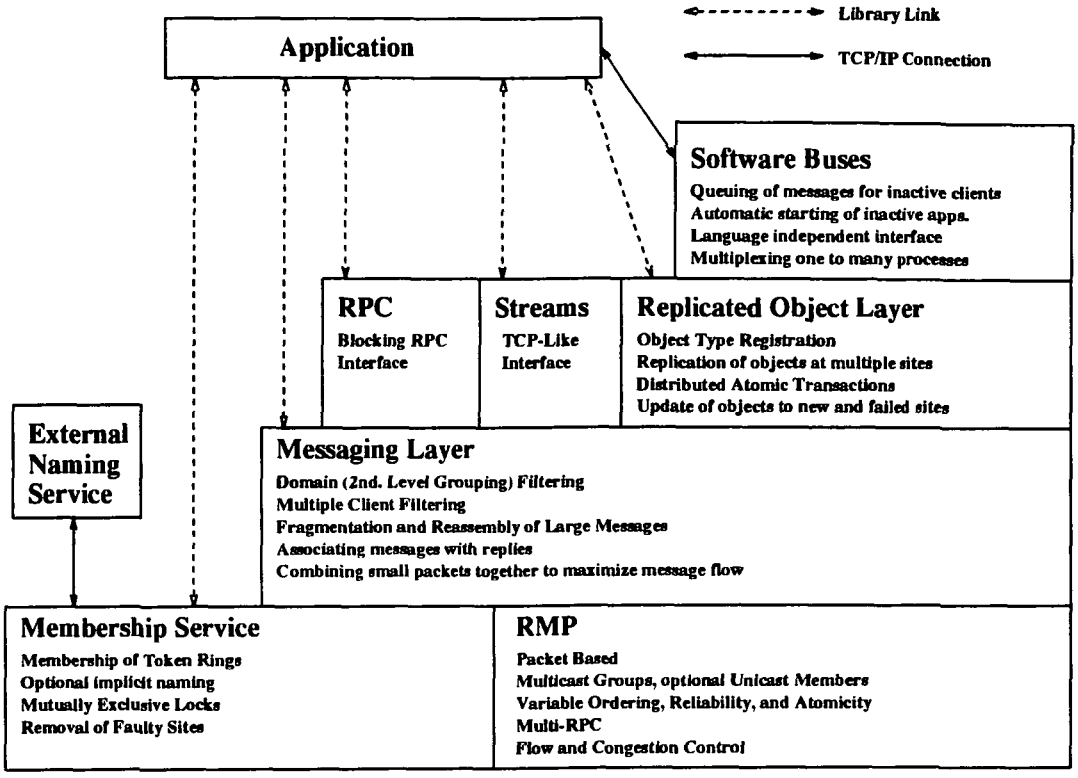


Figure 7.1: RMP Extended Architecture

is just this base layer. However, it is important to mention how RMP has been designed and how it is envisioned to interoperate with other functional layers. The architecture which has been developed to hold RMP is shown in Figure 7.1. This is the RMP Extended Architecture. It is extended in the sense that at this time only RMP itself has been implemented. Plans are already underway for implementation of the *Streams*, *RPC*, and the *Replicated Object Layer*. Work has already begun on the *Messaging Layer*. Work has also been initiated on constructing a Multicast Address Management service that RMP can use.

The *RMP* layer provides all the functionality and services presented in this document, except membership view changes. The *Membership Service* layer interfaces with the RMP layer to provide efficient membership view changes, add additional in-

formation to the membership views (Handlers and Locks), an optional implicit naming convention to map token ring names to IP multicast address, port, TTL tuples, and detection and removal of faulty sites. All of these features are currently supported by the Membership View Extension except for the detection and removal of faulty sites.

The *Messaging Layer* provides several very important features. The first is a second level group filtering mechanism. This can be thought of as adding protocol port numbers to token rings. The second feature is adding client filtering mechanisms. RMP applications that are constructed to be server applications for *Software Buses* require filtering mechanisms to determine to what client a message is to be delivered to and to determine those packets that it is not interested in. To provide a streams interface, a fragmentation and reassembly service must be constructed. This mechanism fragments large messages into smaller packets that may then be sent via RMP. The receivers of the packets then must reassemble the whole message before delivering it to the application. This mechanism is essential for providing the Streams and RPC interfaces. As an efficiency mechanism, the Messaging Layer also needs to combine multiple small messages into a single packet for RMP to use, thereby maximizing throughput. The Messaging Layer also associates messages with replies. This is needed so that the application need not examine each message to determine if it contains a Handler that it must service or is a reply to a message that it sent with a specified Handler.

One of the most desired and powerful layers is the *Replicated Object Layer*. This layer provides a pool of replicated objects that are kept stable across RMP applications by an internal cache consistency mechanism. Object typing is done at this level purely by size. At higher levels more functionality can be combined to provide stronger object typing, such as the typing used by the CORBA specification [15]. Replicated objects must transparently by kept up-to-date at all sites. New and/or failed sites must be updated with current object states. This provides what is called *stable storage*. More complicated updates and synchronization can be done through distributed atomic

transactions. These are performed by totally ordered messages and therefore are very efficient. Objects at this level are very rudimentary. This allows them to correspond to any piece of information that may mutate over time.

The *Software Bus* layer provides a common, language independent interface to the RMP transport system. Some Software Buses that are planned to be expanded to incorporate RMP are Polylith [23] and the MultiBus [2]. By providing remote execution facilities, queuing of messages for inactive applications, and increasing multiplexing ability the RMP Software Bus layer will provide a very efficient and powerful rapid prototyping environment.

7.2.2 Design Directions

The protocol design is still evolving. Although the base algorithms are very solid, other aspects such as the policy regarding efficient use of NACKs to optimize lost packet recovery and the policy for replying to NACKs to be efficient and avoid "NACK Explosions" on WAN environment are still under investigation. Efficient flow control methods are a serious area of research in networking. Adding the complexity of high latency, high bandwidth networks, such as ATM, and the general lack of experience with multicast group flow control adds even more questions. This is probably the *most* likely area of heavy research in the near future. The reformation algorithms themselves may need to be expanded and changed to fit new networking models. The ability to place Real-Time guarantees on RMP given realistic failure and loss assumptions is a very serious issue as well. The protocol design will continue to be very active for an extended period of time.

7.2.3 Implementation Directions

The RMP implementation also holds a large amount of unanswered questions. The implementation at this stage in design and development is fairly efficient, but imposes

a large amount of overhead to processes that use it. Direct reduction of this overhead is desirable. One avenue that is under consideration to accomplish this is, to use dynamic libraries of some UNIX operating systems. This concept also brings up some questions about how to efficiently handle multiple sites per Internet host.

The implementation also must handle code maintenance, change requests from users, and discrepancy reports from users. In this respect the support of RMP is perhaps made slightly more complicated because errors and "bugs" are more difficult to reproduce and eliminate on distributed systems than in traditional systems.

7.2.4 Verification Directions

Verification of RMP will also be a long term activity. As long as the design of RMP evolves and the RMP implementation changes, the verification effort will have to respond and provide feedback to help the designers and developers. In the short term, the tools presented in Chapter 5 are being used to develop models of the RMP system. In particular the PVS and Murφ tools are being heavily investigated and used. It will also be the verification efforts responsibility to maintain a high fidelity between the implementation and the design model. This is critical to ensuring RMPs acceptance. The use of network analysis tools are also being investigated for possible use in examining the RMP model. It is hoped that this analysis will prove to be very valuable in trimming the protocol for performance and efficiency as well as providing serious incites into potential problems in the future.

Bibliography

[1] K. Birman. *The Process Group Approach to Reliable Dsitributed Computing. Communications of the ACM*, 36(12):37–53, December 1993.

[2] J. Callahan and T. Montgomery. A Decentralized Software Bus based on IP Multicasting. In *Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 65–69. CERC, April 1994.

[3] J. M. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Ssystems*, 2(3):251–273, August 1984.

[4] S. Deering. Host Extensiosn for IP Multicasting. Technical Report RFC-1112, IETF, August 1989.

[5] J. Burch E. Clarke K. McMillan D. Dill and L. Hwang. Symbolic Model Checking 220 States and beyond. In *5th Annual Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society, June 1990.

[6] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1st. edition, 1990.

[7] C. Hedrick. Routing Information Protocol. Technical Report RFC-1058, IETF, June 1988.

[8] D. Dill A. Drexler A. Hu and C. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[9] P. Jain N. Hutchinson and S. Chanson. A Framework for the Non-Monolithic Implementation of Protocols in the x-kernel. In *Proceedings of USENIX High Speed Networking*, pages 13–30, August 1994.

[10] Van Jacobson. Congestion Avoidance and Control. In *SIGCOMM Proceedings*, pages 314–328. ACM, 1988.

[11] Y. Amir D. Dolev S. Kramer and D. Malki. Transis: A Communication Subsystem for High Availability. Technical Report CS9113, Hebrew University of Jerusalem, November 1991.

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] C. Maeda and B. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles.* ACM, December 1993.

[14] D. Agarwal P. Melliar-Smith and L. Moser. Totem: A Protocol for Messaging Ordering in a Wide-Area Network. In *First ISMM International Conference on Computer Communications and Networks*, pages 1–5, June 1992.

[15] OMG Mnagement Group. The Common Object Request Broker: Architecture and Specification. Technical Report OMG Document Number 91.12.1, OMG, 1991.

[16] C. A. Thekkath T. D. Nguyen E. Moy and E. D. Lazowska. Implementing Network Protocols at User Level. In *SIGCOMM Proceedings*, pages 64–73. ACM, September 1993.

[17] J. Moy. Multicast Extensions to OSPF. Technical Report RFC-1584, IETF, March 1994.

[18] J. Moy. OSPF Version 2. Technical Report RFC-1583, IETF, March 1994.

[19] D. Waitzman C. Partridge and S. Deering. Distance Vector Multicast Routing Protocol. Technical Report RFC-1075, IETF, November 1988.

[20] J.B. Postel. User Datagram Protocol. Technical Report RFC-768, IETF, August 1980.

[21] J.B. Postel. Internet Protocol. Technical Report RFC-791, IETF, September 1981.

[22] J.B. Postel. Transmission Control Protocol. Technical Report RFC-793, IETF, September 1981.

[23] J. Purtillo. Polylith: An Environment to Support Management of Tool Interfaces. In *ACM SIGPLAN Symposium on Language Issues in Programming Environements*, pages 12–18. ACM, June 1985.

[24] J. Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, SRI, December 1993.

[25] S. Owre J. Rushby and N. Shankar. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE)*, pages 748–752, 1992.

[26] K. Birman A. Schiper and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[27] V. Yodaiken. *A Modal Arithmetic for Reasoning About Multi-Level Systems of Finite State Machines.* PhD thesis, University of Massachusetts (Amherst), 1990.

[28] V. Yodaiken and K. Ramamritham. Verification of a Reliable Net Protocol. In *Formal Technigues in Real-Time and Fault-Tolerant Systems*, pages 193–215, January 1992.

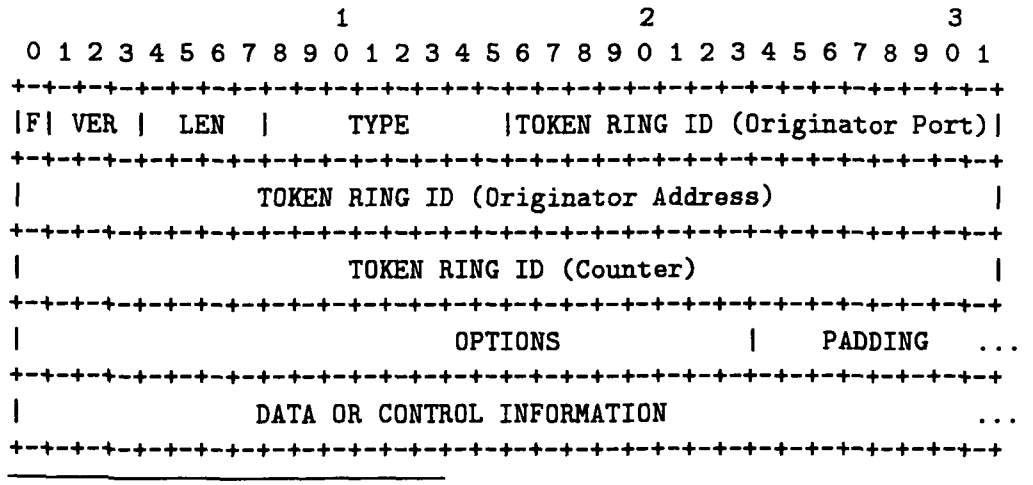
Appendix A

RMP Packet Formats

This appendix contains the packet formats for the RMP packet types discussed in Chapter 3. Each field of the packet is shown and briefly discussed. The packet formats assume a 32 bit IP Addressing scheme.

A.1 RMP Fixed Header

The RMP Fixed Header is included at the beginning of all RMP packets. It is the secondary means that packets bound for different token rings are filtered [1]. Following the fixed header is one of the additional headers shown below.



[1] The first being the multicast address and port of the token ring

F(FWD): Specifies whether the packets should be multicast to the token ring IP Multicast address and port or not. This field is used to unicast a packet from a non-multicast capable member to a multicast capable member that may forward the packet to the token ring.

VER: Specifies the version of the protocol. The current version is 1. Any packet received with an invalid version should be rejected.

LEN: Specifies the length of the options field in four byte words.

TYPE: Specifies the type of RMP packet. The valid types are:

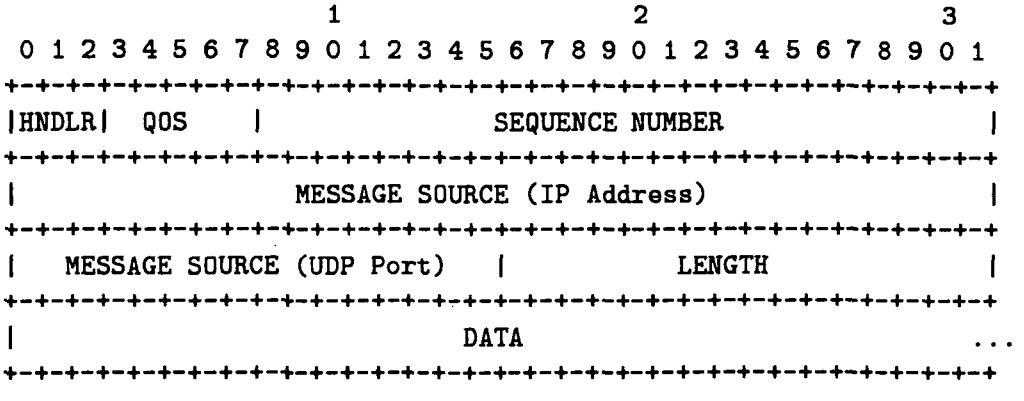
Value	*Packet Type*	*Value*	*Packet Type*
0	*Reserved*	7	Recovery Start
1	Data	8	Recovery Vote
2	ACK	9	Recovery ACK New List
3	Confirm Token Pass	10	Recovery Abort
4	NACK	11	Non Member Data
5	New List	12	Non Member ACK
6	List Change Request	13–15	*Reserved*

TOKEN RING ID: Specifies the current Token List ID of the token ring.

OPTIONS: No options defined at this time. This field may be used for further enhancements for debugging purposes, error codes, or checksum

A.2 RMP Data Header

The Data Packet type holds data sent from one member of the token ring. Following the fixed header is an additional header, the RMP Data Header. The format for this additional header is shown below. Following the data header is the actual message data.



HNDLR(Handler): The handler lock number, if any, for the data packet. The valid values are:

Handler Value	*Required Handler*
0	None
1–6	The process, if any, that holds the handler with the same number
7	The process, if any, that holds the highest priority handler. 1 is the highest priority and 6 is the lowest.

QOS: Specifies the desired QoS of the data packet. The semantics of the delivery of the packet are discussed on page 21. The valid values for the QoS field are:

QoS Value	*QoS Desired*
0	*Reserved*
1	Unreliable
2	Unordered
3	Source Ordered
4	Causally Ordered *(Optional)*
5	Totally Ordered
6–29	K Resilient, K set to $(QoS - 5)$
30	Majority Resilient
31	Totally Resilient

SEQUENCE NUMBER: Each Data packet source stamps each Data packet it sends, except Data packets with QoS of *Unreliable*, with a sequence number. Each source also keeps a sequence number counter for each token ring it may be a member of. The sequence number counter starts at 0 when the source sends its List Change Request packet to join the list. The first Data packet the source sends has a sequence number of 1. Therefore each Data packet, except *Unreliable* ones, has a monotonically increasing sequence number from 1. Because sequence numbers are 24 bit in length, they have a valid range of 0 to $2^{24} - 1$. And because this range is finite, all arithmetic and comparison with these numbers must be modulo 2^{24}. RMP requires that no more than 2^{23} packets can be created over a period equal to the maximum lifetime for a datagram packet in the network. This condition holds true when IP is used as the datagram service.

MESSAGE SOURCE: Specifies the RMP Process ID of the message source.

LENGTH: Specifies the size of the data field in octets.

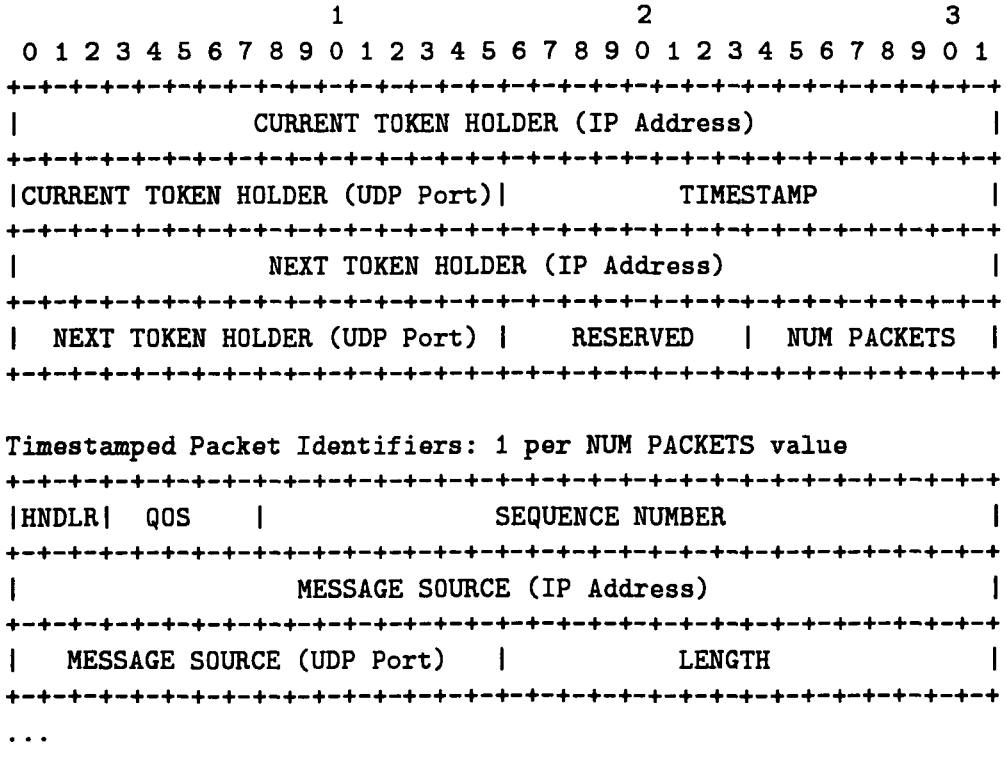
DATA: The data to be delivered.

A.3 Control Packets

Control Packets are packets that contain information vital to RMP normal operation and maintenance of membership views. The packet types that are classified as Control Packets are ACK Packets, Confirm Packets, NACK Packets, New List Packets, and List Change Request Packets. Any retransmissions of these packet types due to a NACK for that packet do *not* require modification of the packet header to show the current state. The original packet in its entirety is retransmitted. The control information for each packet follows directly after the fixed header for the packet.

A.3.1 ACK Packet

ACK Packets transfer the token to a new token site and impose ordering on one or more Data Packets and/or Non-Member Data Packets.



CURRENT TOKEN HOLDER: Specifies the RMP Process ID of the member sending the ACK packet.

TIMESTAMP: Specifies the timestamp of the ACK packet. Timestamps have a range of 0 to $2^{16} - 1$ and therefore all arithmetic and comparisons on them must be modulo 2^{16}.

NEXT TOKEN HOLDER: Specifies the RMP Process ID for the member that will become the next token site.

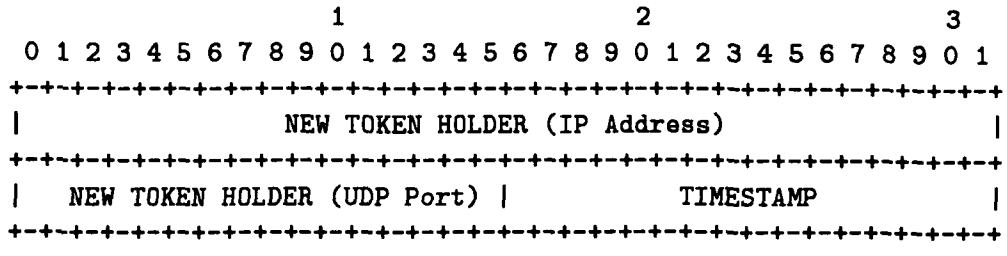
RESERVED: Unused field, zeroed when sent, and ignored when received.

NUM PACKETS: Specifies how many Data packets and Non Member Data packets are timestamped and acknowledged by this ACK packet.

Timestamped Packet Identifiers: There are **NUM PACKETS** sets of these identifiers that follow after the ACK header. Each identifier contains the same information as the Data packet of Non-Member Data packet it timestamps. The implied timestamps of the Data and Non member Data packets follow monotonically from the ACK timestamp field and are in the order shown in the ACK packet.

A.3.2 Confirm Token Pass Packet

The Confirm Token Pass Packet provides a positive acknowledgment for a token site passing the token.

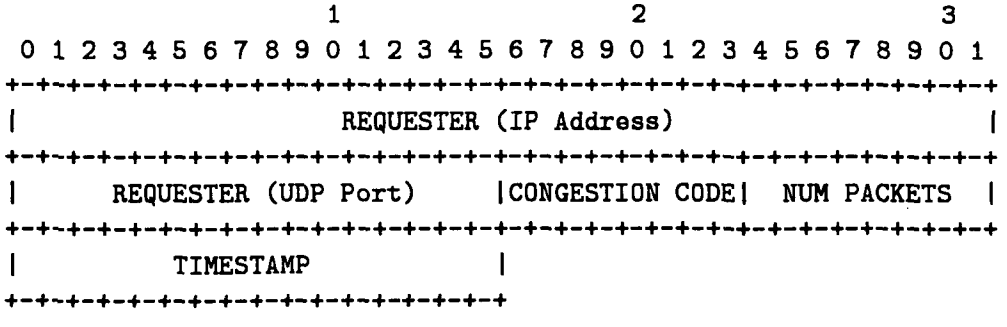


NEW TOKEN HOLDER: Specifies the RMP Process ID of the member that accepts the token and is sending the Confirm Token Pass packet.

TIMESTAMP: Specifies the timestamp of the ACK packet or New List packet that passed the token to the member.

A.3.3 NACK Packet

The NACK Packet is used to request retransmission of lost packets.



REQUESTER: Specifies the RMP Process ID for the member requesting the missing packets.

CONGESTION CODE: Specifies the reason for the NACK. The valid values are:

Value	*Semantic*
0	*Reserved*
1	Buffer Overrun
2	Probable network congestion
3–255	*Reserved*

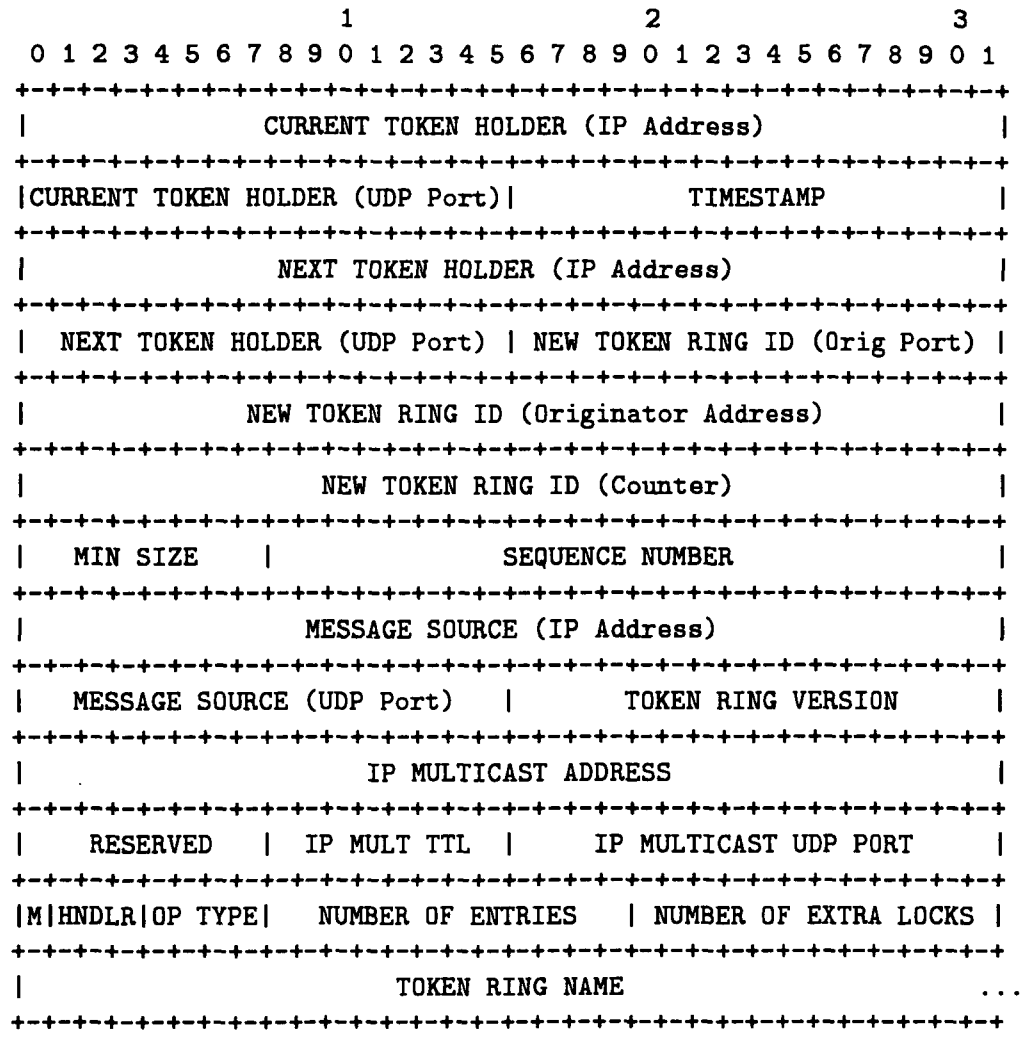
NUM PACKETS: Specifies the number of packets requested.

TIMESTAMP: Specifies timestamp of the first missing packet requested.

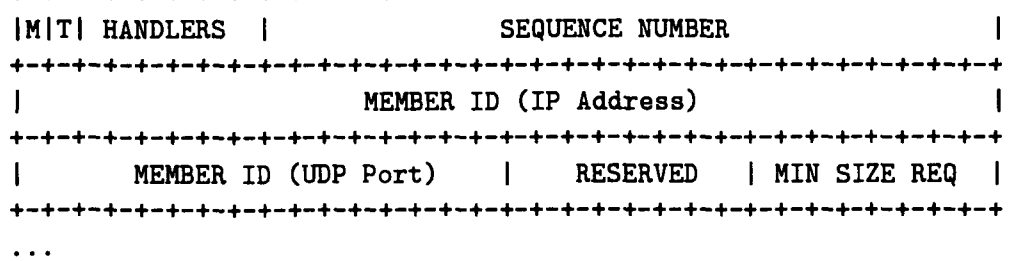
A.3.4 New List Packet

New List Packets contain a new membership view for the group and passes the token.

The format of the packet is shown below.



Token List Entries: 1 per NUMBER OF ENTRIES value



```
Extra Locks: 1 per NUMBER OF EXTRA LOCKS value
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          LOCK NUMBER          |RESERVD|     LOCK HOLDER       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
...
```

CURRENT TOKEN HOLDER: Specifies the RMP Process ID of the member sending the New List packet and passing the token.

TIMESTAMP: Specifies the timestamp of the New List packet.

NEXT TOKEN HOLDER: Specifies the RMP Process ID of the member that the token is being passed to.

NEW TOKEN RING ID: Specifies the new Token List ID for the token ring. This ID will be used as the TOKEN RING ID for each packet following this New List packet. The NEW TOKEN RING ID has a Originator IP Address and Originator Port that is equal to the RMP Process ID of the member that generates the New List packet, the CURRENT TOKEN HOLDER. The Counter is the value of the old TOKEN RING ID Counter +1. This Counter has a range of 0 to $2^{32} - 1$. Thus all arithmetic and comparison operations on these values must be done modulo 2^{32}.

NUMBER OF ENTRIES: Specifies the number of Token List members, or entries.

NUMBER OF EXTRA LOCKS: Specifies the number of locks being held by members of the group. This does not count handler locks.

MESSAGE SOURCE: Specifies the RMP Process ID for the source of the List Change Request packet that this New List packet corresponds to.

SEQUENCE NUMBER: Specifies the sequence number of the List Change Request packet that this New List packet corresponds to.

TOKEN LIST VERSION: Specifies the version of the new token list.

IP MULTICAST ADDRESS: Specifies the IP Multicast address used by the token ring. This is used to notify members who join through a unicast List Change Request packet sent to a well known member of the token ring what the IP Multicast address of the ring is.

IP MULT TTL: Specifies the IP Multicast TTL value for the token ring.

IP MULTICAST UDP PORT: Specifies the IP Multicast UDP Port used by the token ring.

M(Multicast Capable): Specifies if the member that sent the List Change Request packet corresponding to the New List packet is multicast capable or not.

HNDLR(Handlers): Specifies the handler value that the List Change Request packet corresponding the the New List packet was to be performed on. This field is only used in operation types 3–6.

OP TYPE: Specifies the type of operation that was performed on the token list. This field, in effect, details why the New List packet was generated. It also may report errors. The valid operation types are:

Value	*Operation Type Description*
0	*Reserved*
1	Requesting Member added to Token List
2	Requesting Member removed from Token List
3	Requesting Member received a Handler Lock
4	Requesting Member released a Handler Lock
5	Requesting Member was denied a Handler Lock
6	Requesting Member attempted to release a Handler Lock it did not hold
7	Reformation occurred and was successful
8	Reformation occurred with possible atomicity violations
9	Failed Reformation due to partition criteria violations (An Invalid List was created)
10	Time-To-Live of Token List ID Expired
11–15	*Reserved*

MIN SIZE: Specifies the minimum size of the token ring after a failure. This is the minimum partition criteria. This field is always the maximum of the individual MIN SIZE REQ fields of the token list entries. The valid values for this field are:

Value	*Minimum Partition Size*
0	*Reserved*
1–253	Equal to value
254	The majority of sites in the old list. Exactly half the number is not sufficient
255	All sites in the old list

TOKEN RING NAME: Specifies the null-terminated ASCII name for this token ring. This name must end on a word boundary, which may necessitate that 1–3 extra octets of padding be included after the trailing zero of the name. These octets must be set to zero.

Token List Entries: Each entry contains information on each member of the token list. There are NUMBER OF ENTRIES of these entries. When this list is committed to an application, the RMP Process examines the list and caches these values for later use in generating ACKs, examining sequence numbers, etc. When the process sees an entry corresponding to itself, it then marks the entry following it as the site it will

pass the token to when necessary. If the process is the last entry in the list, then the process marks the first entry in the list as the member it will pass the token to.

M(Multicast Capable): Specifies whether the member is multicast capable or not. Each member of the token list that is not multicast capable requires that each other member in the list unicast each packet that it sends to the list to these members as well as sending the packet to the IP Multicast group. Non-Members of the token ring have this field set to zero.

T(Token Ring Member): Specifies whether the RMP Process is a member of the token ring or not. Non-Members periodically get flushed from the list. The inclusion of Non-Members into the token list is used to bring new token list members up to date with respect to recent Multi-RPC activities.

HANDLERS: Specifies the Handler mask for the member. Each bit position represents a handler lock value. Non-Members of the token ring have this field set to zero.

SEQUENCE NUMBER: Specifies the next sequence number from the RMP Process that is to be delivered. This is used for Non-Members as well.

MEMBER ID: Specifies the RMP Process ID for the token list entry. This is used for Non-Members as well.

RESERVED: Unused, zeroed when sent, and ignored when received.

MIN SIZE REQ: Specifies the members vote for the minimum partition size that is allowed to form after a failure. Non-Members set the field to zero.

Extra Locks: The token ring has 4095 locks that members may request and release. Each one is assured to be mutually exclusive, only one member may be in possession of it at one time. The first six locks (1–6) are handler locks. Locks 7–4095 are extra locks. These locks have semantics that are totally dependent on what the application desires to use them for. Each lock that is being used is represented by a LOCK NUMBER, LOCK HOLDER tuple in the New List Packet.

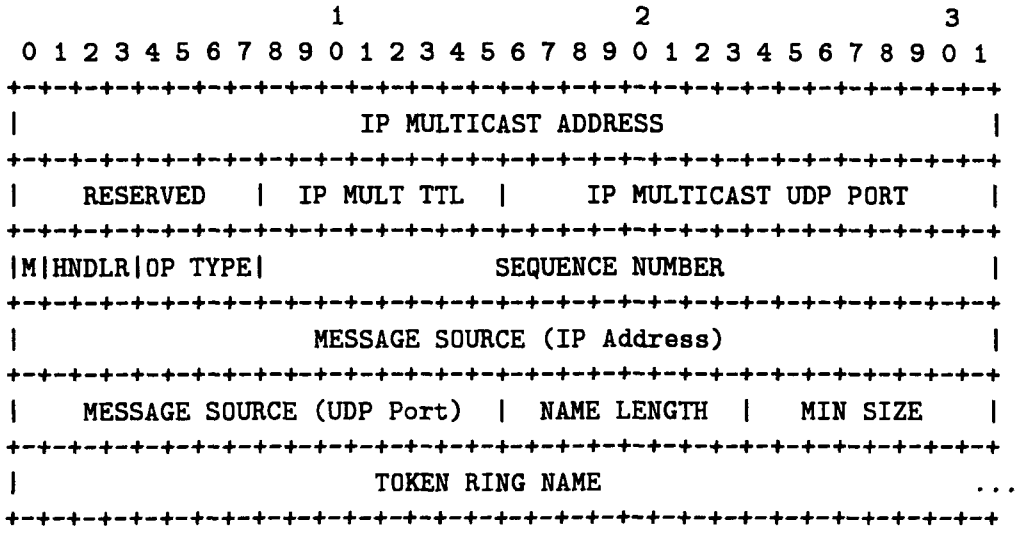
LOCK NUMBER: Specifies the lock number. Valid range is 7–4095.

RESERVD: Unused, zeroed when sent, and ignored when received.

LOCK HOLDER: Specifies the index in the token list of the member that holds the lock. The first member in the list is denoted 1.

A.3.5 List Change Request Packet

Changes to the membership view are requested by a List Change Request Packet. The format for the packet is shown below.



IP MULTICAST ADDRESS: Specifies the IP Multicast address used by the token ring.

IP MULT TTL: Specifies the IP Multicast TTL used by the token ring.

IP MULTICAST UDP PORT: Specifies the IP Multicast UDP Port used by the token ring.

M(Multicast Capable): Specifies whether the process in the List Change Request is multicast capable or not.

HNDLR(Handler): Specifies the handler value associated with the List Change Request packet. This is used only for OP TYPE of values 3–4.

OP TYPE(Operation Type): Specifies the operation type desired by the List Change Request packet. The valid values for this field are:

Value	*Operation Type Description*
0	*Reserved*
1	Add Member to Token List
2	Remove Member from Token List
3	Request Handler Lock
4	Release Handler Lock
5	Time-To-Live of Token List ID Expired (Null OP)
6–15	*Reserved*

MESSAGE SOURCE: Specifies the RMP Process ID for the process sending the List Change Request packet.

SEQUENCE NUMBER: Specifies the sequence number of the List Change Request packet.

NAME LENGTH: Specifies the length of the TOKEN RING NAME field. This is done exactly as it is done in New List packets.

MIN SIZE: The RMP process' vote for the minimum partition size of the token ring.

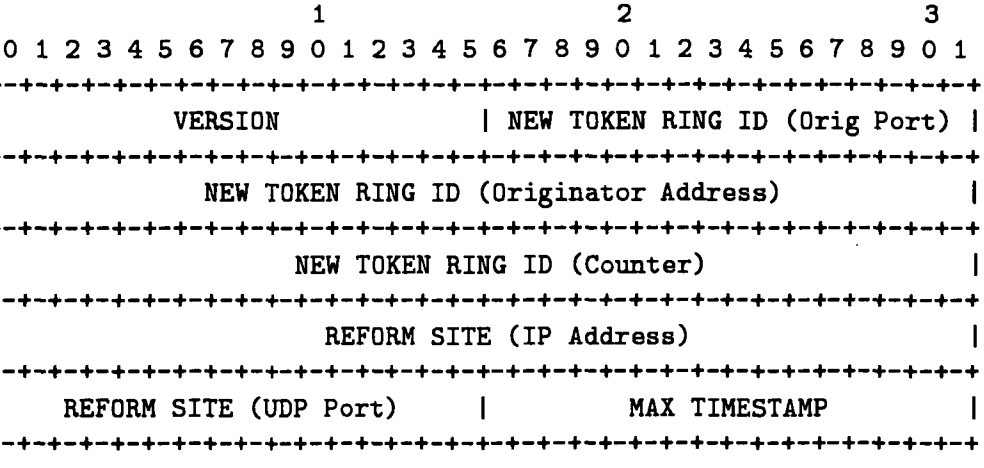
TOKEN RING NAME: Specifies the null-terminated ASCII name for the token ring. It is handled exactly as is done in New List packets.

A.4 Failure Recovery Packets

Failure Recovery Packets contain information relevant to the Reformation Extension of RMP operation. Each Failure Recovery packet type contains a fixed header. The Token List ID for the fixed header is the last known Token List ID before the failure was detected. This ID does not change until the New List is committed after reformation has finished.

A.4.1 Recovery Start Packet

The Recovery Start Packet is sent by the failure detecting site to the members of the token ring.



VERSION: Specifies the token ring version of this current reformation. The valid range of VERSION is 0 to $2^{16} - 1$. And all arithmetic operations and comparisons must be modulo 2^{16}.

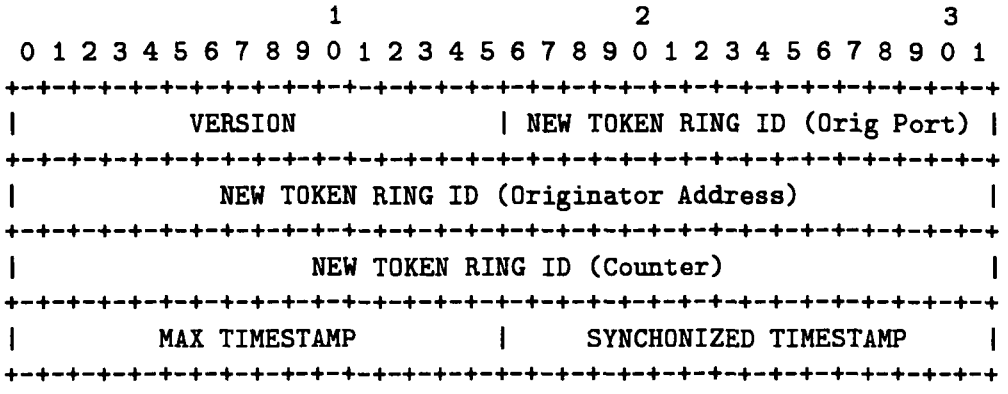
NEW TOKEN RING ID: Specifies the Token List ID *if* the reformation succeeds.

REFORM SITE: Specifies the RMP Process ID for the site that is initiating and controlling the reformation, the reform site.

MAX TIMESTAMP: Specifies the SynchTSP, from Section 3.4.4, for the reformation.

A.4.2 Recovery Vote Packet

Upon receiving a Recovery Start Packet from a member of the token ring, the other sites in the token ring send Recovery Vote Packets to the reform site.



VERSION: Specifies the token ring version of the current reformation. This has the same semantics of the Recovery Vote packet field.

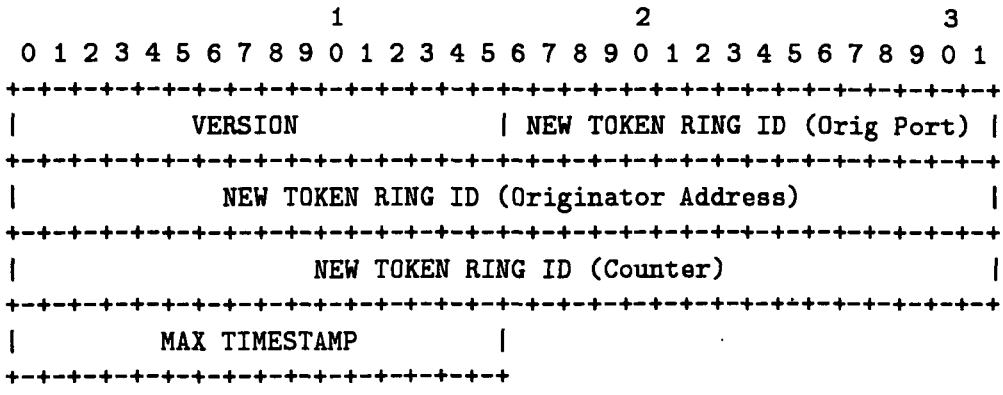
TOKEN RING ID: Specifies the Token List ID for the ring *if* the reformation succeeds.

MAX TIMESTAMP: Specifies the SynchTSP, from Section 3.4.4, for the current reformation. If the process sending the Recovery Vote packet has a higher timestamped packet than the Recover Start packet field, then the process sends this new value.

SYNCHRONIZED TIMESTAMP: Specifies up to what timestamp the process sending the Recovery Vote packet is synchronized up to.

A.4.3 Recovery ACK New List Packet

The Recovery ACK New List Packet is sent to the reform site to signify that the sending site received a New List Packet.



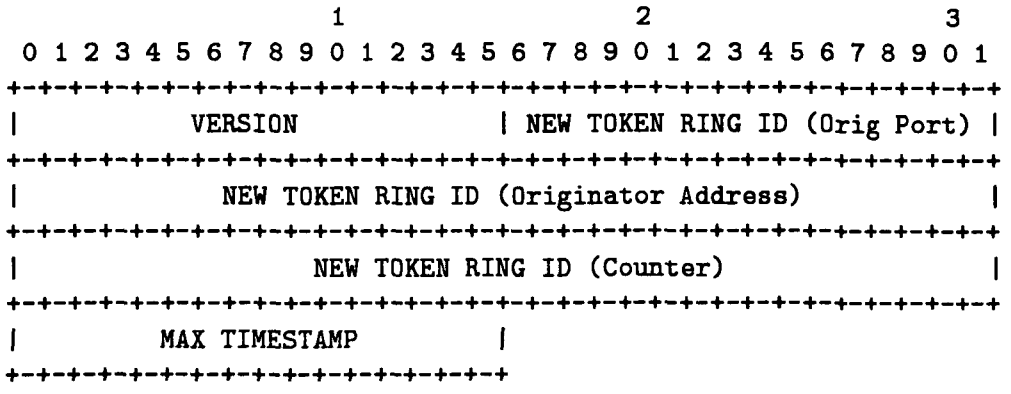
VERSION: Specifies the token ring version for the current reformation. This has the same semantics as the Recovery Start packet field.

NEW TOKEN RING ID: Specifies the Token List ID for the ring *if* the current reformation succeeds.

MAX TIMESTAMP: Specifies the SynchTSP, from Section 3.4.4, for the current reformation. This value must be one less than the timestamp of the New List packet that this Recovery ACK New List packet is in response to.

A.4.4 Recovery Abort Packet

Upon detection of a failure during reformation, a Recovery Abort Packet is sent to signify that the current reformation is to be aborted and a new reformation initiated.



VERSION: Specifies the token ring version for the current reformation. This has the same semantics as the Recovery Start packet field.

NEW TOKEN RING ID: Specifies the Token List ID of the failed reformation.

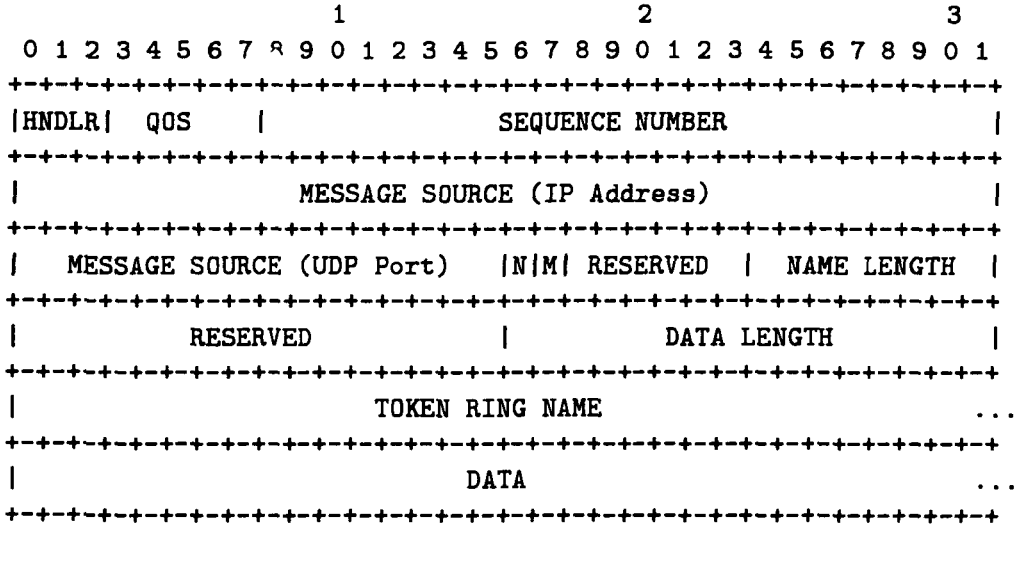
MAX TIMESTAMP: Specifies the SynchTSP, from Section 3.4.4, for the failed reformation.

A.5 Non-Member Packets

Non Member Packets are sent from a Non-Member of the token ring to the token ring to be ordered and optionally replied to by a member of the ring. Each Non-Member Packet contains a fixed header. The Token Ring ID for the fixed header is set to zero unless the Non-Member happens to know the current Token Ring ID.

A.5.1 Non-Member Data Packet

The Non-Member Data Packet is analogous to the Data Packet but is used exclusively by Non-Members of the token ring.



HNDLR(Handler): Specifies the handler value, if any, for the Non Member Data packet. This has the same semantics as the Data packet HNDLR field.

QOS: Specifies the desired QoS for the Non Member Data packet. This has the same semantics as the Data packet QoS of the same value.

SEQUENCE NUMBER: Specifies the sequence number of the Non Member Data packet. This has the same semantics as the Data packet sequence numbers.

MESSAGE SOURCE: Specifies the RMP Process ID for the process sending the Non Member Data packet.

N(No ACK): Specifies whether a Non Member ACK should be sent for Non Member Data packet or not.

M(Multiple Copies): Specifies whether multiple copies of the Non Member Data packet should be delivered to the members of the token ring or not.

RESERVED: Unused, zeroed when sent, and ignored when received.

NAME LENGTH: Specifies the length of the TOKEN RING NAME field. This has the same semantics as the New List packet field.

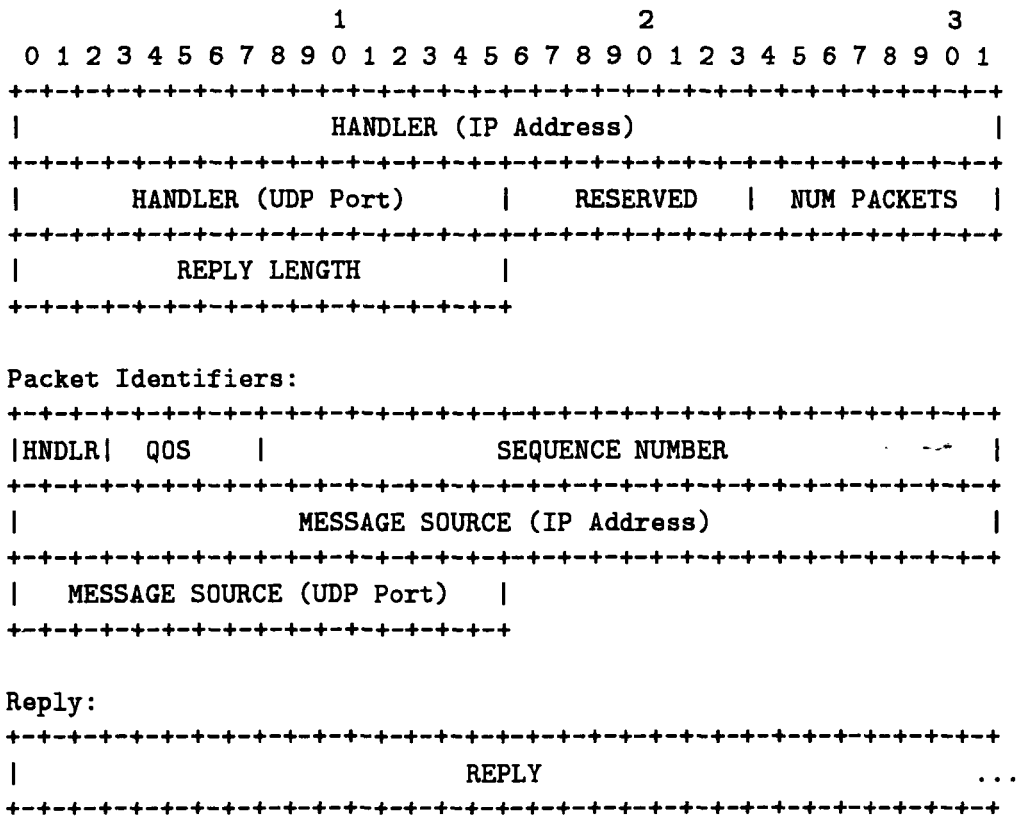
DATA LENGTH: Specifies the size of the data field in octets.

TOKEN RING NAME: Specifies the null-terminated ASCII name for the token ring. This has the same semantics as the New List packet field.

DATA: The data to be delivered.

A.5.2 Non-Member ACK Packet

Non-Member ACK Packets are sent from a member of the token ring to the Non-Member as a means of notification that a corresponding set of Non-Member Data Packets have been received and ordered. Optionally, the sender may attach reply information at the end of the Non-Member ACK Packet.



HANDLER: Specifies the RMP Process ID for the member of the token ring that generated the Non Member ACK packet.

NUM PACKETS: Specifies how many Non Member Data packets this Non Member ACK packet provides acknowledgments for.

REPLY LENGTH: Specifies the length of the REPLY field in octets.

Packet Identifiers: Each packet identifier represents a Non Member Data packet that is acknowledged by this Non Member ACK packet. The fields of the packet identifier correspond with the Non Member Data packet fields.

REPLY: The data of the reply. (Optional).

Appendix B

Complete State Tables

This appendix contains the complete state tables for RMP Normal Operation, Multi-RPC Extensions, Membership Change Extensions, and Reformation Extensions. First the event types are presented in Table B.1. Each RMP operation state, normal and extended, is presented individually and its events shown. The Reformation Extension states are duplicated here for consistency.

Event	*Description*
Data	Reception of a Data Packet
ACK	Reception of an ACK Packet
NACK	Reception of a NACK Packet
Confirm	Reception of a Confirm Packet
Non-Member Data	Reception of a Non-Member Data Packet
Non-Member ACK	Reception of a Non-Member ACK Packet
New List	Reception of a New List Packet
List Change Request	Reception of a List Change Request Packet
Recovery Start	Reception of a Recovery Start Packet
Recovery Vote	Reception of a Recovery Vote Packet
Recovery ACK New List	Reception of a Recovery ACK New List Packet
Recovery Abort	Reception of a Recovery Abort Packet
Transmission Failure	A Packet requiring positive acknowledgement has been retransmitted X number of times without receiving that positive acknowledgement
Token Pass Alarm	Expired Timer for mandatory Token Pass
Confirm Token Pass Alarm	Expired Timer for Confirm notification
Check Non-Members Alarm	Expired Timer for flushing Non Token Ring Mmebers from Membership List (local change)
Random Timeout Alarm	Expired Random Timeout Timer
Mandatory Leave Alarm	Expired Mandatory leave timer
Commit New List	A New List Packet is committed
Join Request	Application requests to join a Token Ring

Table B.1: Event Descriptions

Event	*Conditions(s)*	*Next State*	*Action(s)*
Data	Token Passed	Passing Token	place Packet in DataQ Pass-Token
Data	Token *not* Passed	Token Site	place Packet in DataQ Pass-Token
Non-Member Data	Sequence Number $\geq$ *Expected* from source Token Passed	Passing Token	place Packet in DataQ Pass-Token
Non-Member Data	Sequence Number $\geq$ *Expected* from source Token *not* Passed	Token Site	place Packet in DataQ Pass-Token
Non-Member Data	Sequence Number $<$ *Delivered* from source	Token Site	Unicast Non-Member ACK to source
List Change Request	Token Passed	Passing Token	place Packet in DataQ Pass-Token
List Change Request	Token *not* Passed	Token Site	place Packet in DataQ Pass-Token
ACK	Site named Token Site	Token Site	Unicast Confirm to Packet source
New List	Site named Token Site	Token Site	Unicast Confirm to Packet source
NACK	(none)	Token Site	Send any packets that were requested and present
Transmission Failure	(none)	Start Recovery	Send Recovery Start Packet
Recovery Start	(none)	Sent Vote	Unicast Recovery Vote Packet to Reformation Site
Token Pass Alarm	(none)	Passing Token	Generate Null ACK Send ACK Packet
Confirm Token Pass Alarm	(none)	Token Site	Unicast Confirm to last Token Site
Check Non Members Alarm	(none)	Token Site)	Remove all "Clients" that have "timed out"

Table B.2: Token Site State

Event	*Conditions(s)*	*Next State*	*Action(s)*
Data	(none)	Passing Token	place Packet in DataQ Update-OrderingQ
Non-Member Data	(none)	Passing Token	place Packet in DataQ Update-OrderingQ
List Change Request	(none)	Passing Token	place Packet in DataQ Update-OrderingQ
ACK	ACK Timestamp $\geq$ Last Token Pass Timestamp	Not Token Site	Pass ACK to Next State
New List	New List Timestamp $\geq$ Last Token Pass Timestamp	Not Token Site	Pass New List to Next State
Confirm	Timestamp $\geq$ Last Token Pass Timestamp	Not Token Site	(none)
NACK	(none)	Passing Token	Send any packets that were requested and present
Transmission Failure	(none)	Start Recovery	Send Recovery Start Packet
Recovery Start	(none)	Sent Vote	Unicast Recovery Vote Packet to Reformation Site
Check Non Members Alarm	(none)	Passing Token	Remove all "Clients" that have "timed out"

Table B.3: Passing Token State

Event	*Conditions(s)*	*Next State*	*Action(s)*
Data	(none)	Not Token Site	place Packet in DataQ Update-OrderingQ
Non-Member Data	(none)	Not Token Site	place Packet in DataQ Update-OrderingQ
List Change Request	(none)	Not Token Site	place Packet in DataQ Update-OrderingQ
New List	Site *not* named Token Site	Not Token Site	Add-New-List Update-OrderingQ
New List	Site named Token Site OrderingQ consistent	Token Site	Add-New-List Update-OrderingQ
New List	Site named Token Site OrderingQ *not* consistent	Getting Packets	Add-New-List Update-OrderingQ
ACK	Site *not* named Token Site	Not Token Site	Add-ACK Update-OrderingQ
ACK	Site named Token Site OrderingQ consistent	Token Site	Add-ACK Update-OrderingQ
ACK	Site named Token Site OrderingQ *not* consistent	Getting Packets	Add-ACK Update-OrderingQ
NACK	(none)	Not Token Site	Send any packets that were requested and present
Transmission Failure	(none)	Start Recovery	Send Recovery Start Packet
Recovery Start	(none)	Sent Vote	Unicast Recovery Vote Packet to Reformation Site
Check Non Members Alarm	(none)	Not Token Site	Remove all "Clients" that have "timed out"
Commit New List	New List does not contain site	Leaving Ring	(none)

Table B.4: Not Token Site State

Event	*Conditions(s)*	*Next State*	*Action(s)*
Data	OrderingQ consistent	Token Site	place Packet in DataQ Update-OrderingQ
Data	OrderingQ *not* consistent	Getting Packets	place Packet in DataQ Update-OrderingQ
Non-Member Data	OrderingQ consistent	Token Site	place Packet in DataQ Update-OrderingQ
Non-Member Data	OrderingQ *not* consistent	Getting Packets	place Packet in DataQ Update-OrderingQ
New List	OrderingQ consistent	Token Site	Add-New-List Update-OrderingQ
New List	OrderingQ *not* consistent	Getting Packets	Add-New-List Update-OrderingQ
ACK	OrderingQ consistent	Token Site	Add-ACK Update-OrderingQ
ACK	OrderingQ *not* consistent	Getting Packets	Add-ACK Update-OrderingQ
NACK	(none)	Getting Packets	Send any packets that were requested and present
Transmission Failure	(none)	Start Recovery	Send Recovery Start Packet
Recovery Start	(none)	Sent Vote	Unicast Recovery Vote Packet to Reformation Site
Check Non Members Alarm	(none)	Getting Packets	Remove all "Clients" that have "timed out"

Table B.5: Getting Packets State

Event	*Conditions(s)*	*Next State*	*Action(s)*
Join Request	(none)	Joining Ring	Send a List Change Request to join Token Ring
Non-Member ACK	ACK does *not* hold a reply	Not In Ring	(none)
Non-Member ACK	ACK does hold reply	Not In Ring	Delivery reply to application

Table B.6: Not In Ring State

Event	*Conditions(s)*	*Next State*	*Action(s)*
New List	Site is named Token Site	Token Site	Add-New-List Update-OrderingQ Commit New List Packet
Transmission Failure	(none)	Token Site	Form own Token Ring

Table B.7: Joining Ring State

Event	*Conditions(s)*	*Next State*	*Action(s)*
New List	Timestamp $\geq$ New List that removed site $+N$	Not In Ring	(none)
New List	Timestamp $<$ New List that removed site $+N$	Leaving Ring	(none)
ACK	Timestamp $\geq$ New List that removed site $+N$	Not In Ring	(none)
ACK	Timestamp $<$ New List that removed site $+N$	Leaving Ring	(none)
NACK	(none)	Leaving Ring	Send any packets that were requested and present
Mandatory Leave Alarm	(none)	Not In Ring	(none)

Table B.8: Leaving Ring State

Event	*Condition(s)*	*Next State*	*Action(s)*
Data	(none)	Start Recovery	place Packet in DataQ Update-OrderingQ Update SynchTSP
Non-Member Data	(none)	Start Recovery	place Packet in DataQ Update-OrderingQ Update SynchTSP
ACK	Timestamp $\leq$ SynchTSP	Start Recovery	Add-ACK Update-OrderingQ Update SynchTSP
ACK	Timestamp > SynchTSP	Start Recovery	Add-ACK Update-OrderingQ Update SynchTSP to Packet Timestamp
New List	Timestamp $\leq$ SynchTSP	Start Recovery	Add-New-List Update-OrderingQ Update SynchTSP
New List	Timestamp > SynchTSP	Start Recovery	Add-New-List Update-OrderingQ Update SynchTSP to Packet Timestamp
Transmission Failure	Packet type was Rec. Start	Created New List	Create New List Send New List
Recovery Vote	Veriosn is incorrect	Abort Recovery	Send Recovery Abort
Recovery Vote	Source in old List Vote MaxTSP > SynchTSP	Start Recovery	Update SynchTSP to Vote MaxTSP
Recovery Vote	Source in old List Vote MaxTSP $\leq$ SynchTSP	Start Recovery	Update Site Vote
Recovery Vote	Source in old List OrderingQ consistent Have Vote for each site Vote MaxTSPs = SynchTSP	Created New List	Create New List Send New List
Recovery Abort	(none)	Abort Recovery	(none)
Recovery Start	Source is *not* Reform Site	Abort Recovery	Send Recovery Abort

Table B.9: Start Recovery State

Event	*Condition(s)*	*Next State*	*Action(s)*
Recovery ACK New List	Missing ACKs from 1 or more Sites	Created New List	Mark Source as ACK sent
Recovery ACK New List	Have ACKs from All Sites List is valid	Passing Token	Add-New-List Commit New List Send Null ACK
Recovery ACK New List	Have ACKs from All Sites List is *invalid*	Not In Ring	Add-New-List Commit New List
Transmission Failure	Packet was New List	Abort Recovery	Send Recovery Abort
Recovery Abort	(none)	Abort Recovery	(none)
Recovery Start	Source is *not* Reform Site	Abort Recovery	Send Recovery Abort

Table B.10: Created New List State

Event	*Condition(s)*	*Next State*	*Action(s)*
Recovery Start	Source is Reform Site	Sent Vote	Unicast Recovery Vote to Reform Site
New List	Source is Reform Site Timestamp = SynchTSP + 1 Version is correct	ACK New List	Unicast Recovery ACK New List to Reform Site
New List	Source is Reform Site Timestamp = SynchTSP + 1 Version is correct List is *invalid*	Not In Ring	Unicast Recovery ACK New List to Reform Site
New List	Verison is incorrect	Abort Recovery	Send Recovery Abort
Data	(none)	Sent Vote	place Packet in DataQ Update-OrderingQ Update Recovery Vote
Non-Member Data	(none)	Sent Vote	place Packet in DataQ Update-OrderingQ Update Recovery Vote
ACK	Timestamp $\leq$ SynchTSP	Sent Vote	Add-ACK Update-OrderingQ Update Recovery Vote
ACK	Timestamp > SynchTSP	Sent Vote	Add-ACK Update-OrderingQ Update Recovery Vote
New List	Timestamp $\leq$ SynchTSP	Sent Vote	Update Recovery Vote
New List	Timestamp > SynchTSP	Sent Vote	Update Recovery Vote
Transmission Failure	Packet was Recovery Vote	Abort Recovery	Send Recovery Abort
Recovery Abort	(none)	Abort Recovery	(none)

Table B.11: Sent Vote State

Event	*Condition(s)*	*Next State*	*Action(s)*
New List	Source is Reform Site Version is *correct*	ACK New List	Unicast Recovery ACK New List to Reform Site
New List	Version is incorrect	Abort Recovery	Send Recovery Abort
ACK	Source is Reform Site Site is *not* named Token Site	Not Token Site	Add-New-List Commit New List Add-ACK Update-OrderingQ
ACK	Source is Reform Site Site is named Token Site OrderingQ consistent	Token Site	Add-New-List Commit New List Add-ACK Update-OrderingQ
ACK	Source is Reform Site Site is named Token Site OrderingQ *not* consistent	Getting Packets	Add-New-List Commit New List Add-ACK Update-OrderingQ
Recovery Abort	(none)	Abort Recovery	(none)

Table B.12: ACK New List State

Event	*Condition(s)*	*Next State*	*Action(s)*
Random Timeout Alarm	(none)	Start Recovery	Send Recovery Start
Recovery Start	Version is correct	Sent Vote	Unicast Recovery Vote to Reformation Site
Recovery Start	Version is incorrect	Abort Recovery	Send Recovery Abort
Recovery Vote	Version is incorrect	Abort Recovery	Send Recovery Abort
New List	Version is incorrect	Abort Recovery	Send Recovery Abort
Recovery ACK New List	Version is incorrect	Abort Recovery	Send Recovery Abort

Table B.13: Abort Recovery State

Abstract

This document describes the Reliable Multicast Protocol (RMP) design, first implementation, and formal verification. RMP provides a totally ordered, reliable, atomic multicast service on top of an unreliable multicast datagram service. RMP is fully and symmetrically distributed so that no site bears an undue portion of the communications load. RMP provides a wide range of guarantees, from unreliable delivery to totally ordered delivery, to K-resilient, majority resilient, and totally resilient atomic delivery. These guarantees are selectable on a per message basis. RMP provides many communication options, including virtual synchrony, a publisher/subscriber model of message delivery, a client/server model of delivery, mutually exclusive handlers for messages, and mutually exclusive locks.

It has been commonly believed that total ordering of messages can only be achieved at great performance expense. RMP discounts this. The first implementation of RMP has been shown to provide high throughput performance on Local Area Networks (LAN). For two or more destinations a single LAN, RMP provides higher throughput than any other protocol that does not use multicast or broadcast technology.

The design, implementation, and verification activities of RMP have occurred concurrently. This has allowed the verification to maintain a high fidelity between design model, implementation model, and the verification model. The restrictions of implementation have influenced the design earlier than in normal sequential approaches. The protocol as a whole has matured smoother by the inclusion of several different

perspectives into the product development.

Curriculum Vitae

Todd Montgomery was born on [redacted] in [redacted]. In June of 1988, he graduated from Hampshire High School, Hampshire County, West Virginia. Todd received a Bachelors of Science in Electrical Engineering and a Bachelors of Scienece in Computer Engineering from West Virginia University in December of 1992. During his undergraduate studies, he was employed at the West Virginia University Student Union, the MountainLair, as an audio/visual technician and at the Concurrent Engineering Research Center as a Bulletin Board System administrator.

After graduation he entered the graduate program at West Virginia University in the Spring of 1993, where he studied software engineering, integrated circuit testing, advanced logic design, and programming technigues. In May of 1993, he accepted a research assistant position with NASA Grant NAG 5-2129 at West Virginia University. During this employment, he developed and honed skills related to software engineering and the software life-cycle, as well as, developing skills related to verification and validation of systems. In May of 1994, Todd accepted a position as a research assistant with NASA Cooperative Research Agreement NCCW-0040 at West Virginia University, where he continued to research software engineering, verification and validation technigues, and network protocols.

Mr. Montgomery's major career interests include: software engineering, verification and validation technigues, network protocols, and distributed application development. Upon completion of his Masters of Science in Electrical Engineering, the author

entered the graduate program at West Virginia University as a Ph.D. candidate for the Computer Science Department.

Approval of Examining Committee

John R. Callahan, Ph.D.

Powsiri Klinkhachorn, Ph.D.

Afzel Noore, Ph.D., Chair

Date

304 367-8348 ❑ FAX 304 367-8211 ❑ 100 University Drive ❑ Fairmont WV 26554