A HIGH PERFORMANCE TOTALLY ORDERED MULTICAST PROTOCOL

Brian Whetten, University of California at Berkeley (whetten@cs.berkeley.edu)
Todd Montgomery, West Virginia University (tmont@cerc.wvu.edu)
Simon Kaplan, University of Illinois at Champaign-Urbana (kaplan@cs.uiuc.edu)

Abstract

This paper presents the Reliable Multicast Protocol (RMP). RMP provides a totally ordered, reliable, atomic multicast service on top of an unreliable multicast datagram service such as IP Multicasting. RMP is fully and symmetrically distributed so that no site bears an undue portion of the communication 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 QoS guarantees are selectable on a per packet basis. RMP provides many communication options, including virtual synchrony, a publisher/subscriber model of message delivery, a client/server model of delivery, an implicit naming service, mutually exclusive handlers for messages, and mutually exclusive locks.

It has commonly been held that a large performance penalty must be paid in order to implement total ordering--RMP discounts this. On SparcStation5's on a 1250 KB/sec Ethernet, RMP provides totally ordered packet delivery to one destination at 1070 KB/sec throughput and with 4.0 ms packet latency. The performance stays roughly constant independent of the number of destinations. For two or more destinations on a LAN, RMP provides higher throughput than any protocol that does not use multicast or broadcast.

Keywords: Congestion control, internetworking, distributed network algorithms, reliable multicast

1 Introduction

Totally ordered, reliable broadcast and multicast protocols have existed for quite some time [ChMa84], and provide a powerful tool for programming distributed systems and distributed databases [Chang84]. New applications such as Computer Supported Cooperative Work (CSCW) programs, groupware systems and shared tools can also benefit greatly from this service. In the past, these protocols have had problems with performance, efficiency, and/or scalability. It has become a widespread belief that these are inherent problems with a totally ordered reliable multicast protocol [RaLi93]. In part, this concept resulted from the fact that in the past multicasts had to be implemented as a series of unicasts to each destination. Recent developments such as the IP Multicasting standard [Deering89] now allow a multicast datagram to be sent to multiple destinations over an internetwork. In the case where all destinations are on the same LAN, one multicast packet to all of them costs the same as a unicast packet to just one.

This paper presents the architecture of the Reliable Multicast Protocol (RMP). RMP provides a reliable multicast service on top of unreliable datagram services such as IP and IP Multicasting. RMP is based on the token ring technique originally proposed by J. M. Chang and N. F. Maxemchuk in [ChMa83] and [ChMa84]. This CM protocol provides totally ordered atomic broadcast to a single group of broadcast clients on a LAN.

While RMP uses an algorithm similar to this CM protocol for its basic delivery, it differs in the following ways:

1) RMP provides multiple multicast groups, as opposed to a single broadcast group.

2) RMP provides an implicit naming service that maps textual group names into communication groups.

3) Instead of only providing totally ordered, K-resilient packet delivery, RMP allows the user to select from a wide range of guarantees, including agreed and safe delivery, selectable on a per-packet basis.

4) For increased scalability, RMP allows processes that are not members of a group to send messages to it, and receive replys to messages, through its multi-RPC mechanism.

5) The CM membership algorithm handles all cases, and takes an extended period of time to run. RMP optimizes for the common membership change case with the addition of a new membership algorithm that allows non-failure membership changes to be made at the cost of only a single group message.

6) RMP uses a new membership algorithm that handles faults. It is faster, less likely to mistakenly remove processes from a group, and supports virtual synchrony and extended virtual synchrony.

7) To facilitate replicated services, RMP provides a set of mutually exclusive handlers for messages. A message can request that it be handled, and at most one process will reply to the message.

8) RMP provides a windowed flow and congestion control mechanism that allows RMP to provide high performance over both LANs and WANs, even in the face of congestion.

9) RMP shows how to seamlessly extend reliable multicast to hosts that do not support multicast.

10) The implementation of RMP has demonstrated exceptional performance. For a single destination, its performance rivals or exceeds that of most TCP/IP implementations. The performance stays almost constant irregardless of the number of group members per LAN. For totally ordered messages to groups of three or more processes, RMP provides performance equal to or better than any protocol we know of.

The basic RMP protocol provides what can be thought of as N-way virtual circuits, called *groups*, between sets of processes connected by a multicast medium. It is fully distributed, so that all processes play the same role in communication. While primarily using NACKs for error detection and retransmission, RMP provides true reliability and limits the necessary buffer space by passing a token around the members of a group.

RMP provides a wide range of reliability and ordering guarantees on packet delivery, selectable on a per packet basis. In addition to unreliable and reliable but unordered quality of service (QoS) levels, RMP can provide atomic, reliably delivery of packets ordered with respect to each source. It can also efficiently provide delivery of packets in both total and causal order, using causal ordering as defined in [Lamp78]. Totally ordered delivery also provides virtual synchrony, as first defined by the ISIS project [BSS91]. Virtual synchrony guarantees that when new members join or leave a group these operations appear to be atomic, so that the sets of messages delivered before and after each membership change are consistent across all sites. Using K-resilient fault tolerance, RMP can provide total ordering and atomicity guarantees even in the face of site failures and partitions. For a set of packets with a resiliency level of K, more than K members of a group have to simultaneously partition away or fail in order to have the possibility of violating the total ordering and atomicity guarantees. By setting K to a number larger than half the members of a ring and not allowing minority partitions to continue, total ordering, atomicity, and virtual synchrony can be guaranteed in the face of any set of arbitrary partitions and failures.

The basic RMP model of communication is a publisher/subscriber model based on textual *group names.* In the absence of network partitions, any member of a group (a subscriber) will receive all packets sent (published) to the group associated with that group name. RMP also provides a client/server model of communication, where the servers are members of a group and the clients are not members, but can communicate with the servers by sending *multi-RPC* packets to the group. These packets may be simply acknowledged after being delivered to the group with the requested QoS, or they may be responded to by a single member of the group. RMP uses handlers to guarantee that at most one member will respond to a data packet. Each data packet in RMP has an optional handler number associated with it. These correspond to a set of mutually exclusive handler locks which group members may hold. The group member who holds a given handler lock will be notified upon delivery of a data packet with this handler number that it is supposed to respond to the request. Handler locks are provided in a very efficient way, and can be used for any type of application that requires mutually exclusive locks shared among a group of communicating processes.

A common belief in the research community is that totally ordered reliable multicast protocols are inherently slow. This belief has come about in large part due to the experiences researchers have had with the early versions of ISIS, which for a long time was the only system of this type available. ISIS has since become much faster

[BiCl94], but the misconception remains. Experience with RMP belies this concept. RMP was tested on 8 SparcStation5's on a 10 Mb/sec (1250 KB/sec) Ethernet. In this environment, the throughput to a single destination is 1070 KB/sec, or 86% of the network capacity. For group communication to any group of two or more destinations on a LAN, RMP exceeds not only the maximum throughput of TCP/IP, but any other possible non-multicast and non-broadcast algorithm. This is because both the packet latency and throughput of RMP stay roughly constant as the number of destinations increase, whereas the performance of other algorithms decreases linearly. For a group with 8 destinations, RMP has a 7.4 MB/sec aggregate throughput, which is 5.9 times the bandwidth of the supporting Ethernet. The throughput for RMP does not significantly change as a factor of the ordering guarantees, but the per packet latency does. A totally ordered packet will on average have a latency approximately twice that of an unordered or source ordered packet, and this increases for K-resilient packets. This QoS for latency tradeoff is fundamental to distributed protocols, which is why RMP allows this tradeoff to be made on a per packet basis. Despite this moderate latency penalty for its total ordering, RMP latency to two or more destinations is still lower than most other protocols. Therefore RMP demonstrates that a fault tolerant, reliable, atomic, fully distributed, totally ordered multicast protocol can actually achieve much better performance in group communication than systems that don't provide these features.

Section 2 compares RMP to previous work. Section 3 describes the RMP algorithms. Section 4 analyzes the performance of RMP, and section 5 compares RMP to previous work. Finally, section 6 draws our conclusions and outlines future work and section 7 acknowledges all the support we have had for RMP.

2 Design decisions and comparison to previous work

The biggest decision in building a reliable multicast protocol is how to guarantee the reliability and stability of messages without sacrificing throughput or latency. Latency is defined as the time between when a site has a packet to send and when it is delivered to the destination. A message is defined as going stable when the sender knows all destinations have received it. This is the point at which it no longer needs to be held for possible retransmissions. In a reliable multicast protocol, perhaps the most important factor influencing throughput is the number of ACKs sent per packet, so it is important to minimize this. In order to provide guarantees of total ordering and atomic delivery in the face of failures, a reliable multicast protocol will often delay delivering a packet until after it has received one or more acknowledgements of delivery. This latency for gurantees tradeoff is fundamental to this class of protocols, which is why RMP allows this tradeoff to be made on a per packet basis.

Traditional protocols use positive acknowledgments (ACKs) from the destination to acknowledge successful receipt of a packet. While quickly providing stability of messages, this approach does not scale well to a multicast system, because each destination has to send an ACK for each packet or set of packets. This largely defeats the advantage of using multicast packets, because it decreases both the efficiency and the performance of the protocol. Even though these acknowledgments are small, because they all are sent at the same time they can cause network congestion. In addition, having to process an ACK from each destination increases the load on the sender and decreases the performance of the protocol. One optimization is to not acknowledge every packet. In general, as the number of packets per ACK increases, the length of time for a message to go stable increases, but the lower the load is. As another approach, many systems use negative acknowledgments (NACKs). Negative acknowledgments shift the burden of error detection from the source to the destinations. Packets are stamped with sequential sequence numbers which destinations use to provide reliable delivery by detecting gaps in the sequence numbers and requesting retransmission of the packets corresponding to the gaps. Because the information that a packet has been received is never propagated back to the sender, the senders in these protocols do not ever know for certain that a destination has received a packet. Because of this, senders have to indefinitely keep a copy of each packet sent if the protocol is to be considered truly reliable. In addition, a lost packet will not be detected until another packet is received successfully, which may take a long time if the packet is the last to be sent to the ring for a while. Because of these problems, the RMP algorithm uses a combination of these two approaches. The basic algorithm is based on the ideas of the protocol originally done by Chang and Maxemchuk[ChMa84], [ChMa83].

The MBusI [Carroll93] was the original motivation for RMP. It provides a central server through which clients connect with TCP/IP streams, and an easy to use interface designed to ease the implementation of CSCW applications. It provides both total ordering of messages and reliable multicast, but has very limited scalability, since all packets must be routed through a central point, and duplicate copies sent to each destination.

The Totem protocol [AMSM92] is perhaps closest to RMP in its approach, and has reported similar throughput levels to RMP under heavy load. It also uses a rotating token ring approach, but only provides for a single ring for each broadcast domain. Totem avoids using any ACKs by only allowing the current token holder to send data. This provides high throughput under high load over a low latency network, but provides lower throughput and longer latency under low and asymmetrical loads. In addition, because it only allows a single sender to transmit at a time it will provide lower throughput over longer latency networks. To alleviate this problem they have proposed, but not implemented, gateways to link multiple broadcast domains together.

The ISIS system [BSS91], [Birman93] is one of the pioneering protocols in this field. It provides causal ordering and, if desired, total ordering of messages on top of a reliable multicast protocol. The reliable multicast protocol requires separate acknowledgments from each destination, which limits performance. A new system that provides causal ordering on top of IP Multicasting has been implemented which is much more efficient than the old system [Clark94], and we hope to compare RMP and this new protocol soon.

The Psync protocol [PBS89] is an ingenious protocol that uses piggybacked ACKs to provide causal ordering of messages and detection of dropped packets. However, both it and the similar Trans [MSMA90] and Lansis [ADKM93] protocols require that all of the members of the group regularly transmit messages. The Trans protocol and the ToTo [DKM] protocol implemented on top of Lansis both provide total ordering of messages. These algorithms require that at least a majority of the group members be heard from before a message can be delivered, which causes latency to increase by at least an order of magnitude. For example, for the ToTo protocol to send to a group of 8 destinations under heavy, periodic load from all sources (the best case), the latency is 23.8 ms. This increases to 114.1 ms for lightly loaded poisson sources.

The Multicast Transport Protocol (MTP) [AFM92] is an example of an asymmetric reliable multicast protocol. One site is the communication master which grants "tokens" to group members to allow them to send data. These tokens provide both flow control and total ordering of messages. This causes over dependency on the master, which limits both reliability and performance. MTP also relies exclusively on NACKs for error recovery, which limits reliability and requires extreme amounts of buffer space.

The protocol by Crowcroft and Paliwoda [CrPa88] is one of the first protocols to propose reliable multicast over an internetwork which supports hardware multicast. The protocol provides different levels of reliability guarantees, and uses positive acknowledgments from all destinations for reliability. The paper analyzes the flooding problems that occur with simultaneous ACKs from many destinations and proposes a windowed flow control system, in some ways similar to that used in RMP, to alleviate these problems.

The protocol by Navaratnam, Chanson, and Neufeld [NCN88] is a centralized multicast protocol that uses a single token site to provide total ordering and reliability. It requires that each site send back a positive acknowledgment before the next packet can be sent. An implementation on top of the V-system takes 24.8 ms to send a multicast to four destinations. This protocol also is limited in reliability and scalability by the central server. The xAmp protocol [RoVe92] is distributed but also waits for ACKs from all destinations, and so will exhibit performance similar to NCN and ISIS.

The broadcast protocol proposed by Kaashoek et. al. [KTHB88] uses a central token site to serialize messages and NACKs for retransmissions. It piggybacks ACKs onto sent messages and has the token site regularly contact silent sites in order to limit buffer space. This protocol has reported very good latency (as low as 1.3 ms for a NULL packet) because it has been implemented on top of bare hardware. However, because each message must be transmitted twice it will fundamentally achieve lower throughput than RMP -- 600 KB/sec is a rough upper bound for a 1250 KB/sec Ethernet, as compared to 842 KB/sec for RMP. This will also limit the latency for larger messages; as a 8KB packet in their protocol will spend a minimum of 13.1 ms on the Ethernet, as opposed to 6.7 ms for the message and ACK of RMP.

3 System usage and architecture

RMP is a transport level protocol that provides reliable datagram delivery on top of an unicast or multicast asynchronous, unreliable datagram service. It allows multiple groups of processes to communicate with selectable levels of reliability and ordering through the use of a QoS field in each data packet. RMP supports both implicit and explicit naming as well as publisher/subscriber and client/server models of communication. In addition, RMP allows processes that are not multicast capable to participate in the group communication through the use of forwarding and non-multicast capable flags.

3.1 RMP entities

RMP is organized around *RMP processes, groups, and membership views.* The basic entity that uses RMP to communicate is called a RMP process. A RMP process typically corresponds to a single UNIX process, application, or "software bus". There may be multiple RMP processes on the same host. Each RMP process is

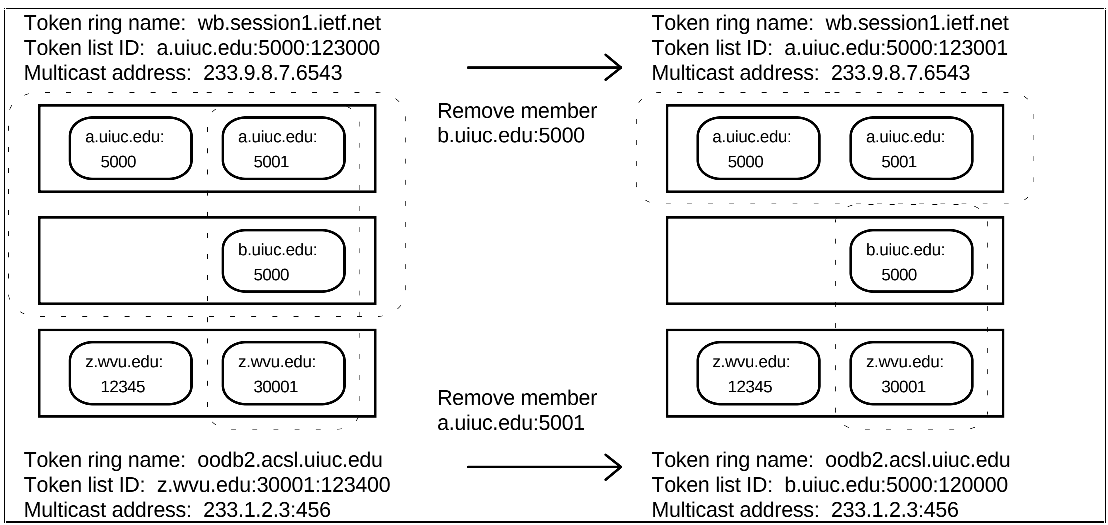

Figure 3.1: RMP entities

uniquely identified by the IP address of its host concatenated with a UDP port number that may not be reused between processes on a host and is constant over the life of that RMP process. This ID, called a *RMP process ID*, is unique across all RMP processes in the internetwork.

A group is the basic unit of group communication and message ordering in RMP, and consists of a set of RMP processes that are receiving packets sent to a given IP Multicast address and port. Each RMP process may be a member of multiple groups, and non-members can reliably send to a group and get optional replies from members of a group using the multi-RPC mechanism, described below. Each group has a *group name*, which is a text string similar to those used in current Internet host names. Unlike host names, a group name identifies a group of members instead of a single host. A group identified by a group name is not guaranteed to be unique across an entire internetwork, as two groups of RMP processes that join a given IP Multicast address and port with a given group name may not overlap due to network partitions or due to non-global multicast domains.

The mapping of group names to IP Multicast {address, port, TTL} tuples may either be handled by an external multicast address allocation authority such as [PEA94], or it may be handled by RMP. The default mapping policy uses a hash function to turn the text string into one of a range of multicast addresses (a subset of the class D IP address space, currently 24 bits) and a 15 bits of a port number. The high bit of a hashed RMP port is always high, to keep it from conflicing with statically assigned port numbers. Each group member specifies its desired TTL when joining a group. The TTL for packets sent to a group is set to the maximum of all the group member's TTL requests. Hash collisions to the same port and address are handled by using guaranteed unique membership view IDs, as described below.

The membership of a group will usually change over time. A given list of the members of a group is called a *membership view*. A membership view is always created by a single RMP process, and is identified by a *membership view ID.* Similarly to Grapevine[BLNS82], a membership view ID consists of an ID that is unique across the breadth of the internetwork concatenated with a counter that is unique across the maximum TTL for that ID. The first half of a membership view ID consists of the RMP process ID of the process that created the membership view, and the second half is based on a counter. One counter is maintained for each RMP process. When an RMP process starts, it initializes its counter to the current time in milliseconds, and it increments this counter every time it creates a new membership view. By not allowing an RMP process to generate more than one membership view per millisecond the process has been in existence and by limiting the lifetime of a membership view to 2^{31} milliseconds (just under 25 days), we guarantee that a membership view ID is unique as long as the clocks of the generating machines are stable. This guarantee eliminates the need to keep the IP Multicast address and port for a group unique across different groups, although it is desirable to avoid these address collisions. In the case of collisions to the same address and port, RMP will use the membership view IDs to filter the packets at each RMP process.

Figure 3.1 shows an example of these entities for a sample scenario with two groups and five RMP processes spread over three hosts. A new membership view ID is generated for both groups after a member is removed from each. Notice that the new ID does not have to be created by the same RMP process that created the last one for that ring.

3.2 Atomicity, reliability and ordering guarantees

Different multicast applications require many different levels of reliability and ordering guarantees in the face of transient network failures such as dropped packets. These applications also require different atomicity guarantees in the face of site failures or partitions. For example, a CSCW application may need packets to be reliably delivered at all sites, with the packets from the same source delivered in the order they were sent. This application may be able to continue, even if some sites fail away or the group partitions in two. On the other hand, a distributed database may require that all packets be delivered in the same total order at all sites, even if some of the sites partition away. RMP supports a wide range of guarantees on packets by allowing different QoS levels to be specified for packets being sent to a group, and by allowing applications to specify the minimum size of a partition that can continue to function in the face of failures. The selectable QoS levels are listed in figure 3.2. All of the QoS levels build upon previous levels, providing any guarantees that a smaller level provides.

The basic RMP QoS levels are unreliable, reliable, source ordered, and totally ordered. They are provided by differing the time at when packets are delivered and enabling or disabling the duplicate detection, NACK, and ACK policies. While throughput remains similar for the different QoS levels, higher QoS levels increase the latency of packet delivery. For example, in the common case of few dropped packets, source ordered packets have about the same latency as unordered packets, and totally ordered packets have about twice the latency of either.

The unreliable QoS is most similar to UDP traffic. An unreliable packet will be delivered 0, 1, or more times to a destination and there are no ordering guarantees on delivery. A reliable packet will be delivered 1 or more times to each destination. The source ordered QoS provides the equivalent guarantees of running a TCP socket from each source to each destination. Packets arrive exactly once at each destination in the same order as they were sent from the sender. Source order does not provide any guarantees on the ordering of packets from multiple senders in the group. Totally ordered delivery serializes all of the packets to a group, delivering all of the packets in the same order at all members of the group. Without globally synchronized clocks, it is not possible to tell which message was "really" sent first, but total ordering guarantees that some order will be imposed over all messages sent to a group, and that messages will be delivered in this order at all sites. This QoS is equivalent to running a TCP socket from each source into a central bus which serializes the packets and then sends them out through a separate TCP socket to each destination. Totally ordered packets are also causally ordered, as per Lamport's definition[Lamp78].

ISIS first defined the notion of *virtual synchrony*[BSS91], [Birman93]. Virtual synchrony often allows a distributed application to execute as if its communication was synchronous, when it is actually asynchronous. The key requirement for virtual synchrony is that all sites see the same set of messages before and after a group membership change. In other words, for a given set of packets delivered to a group, a membership change operation will partition these packets into the same two sets at all sites, and all packets in the first set will be delivered at all sites before any packets are delivered in the second set. RMP provides virtual synchrony for packets that have a QoS of at least totally ordered. This is done by implementing each membership change as a packet with a totally ordered QoS.

A critical question in group protocols is what happens to delivery guarantees in the face of failures or partitions. To solve this problem, RMP offers four levels of fault-tolerant guarantees: atomic delivery within paritions, K-resilient atomic between partitions, agreed delivery between partitions, and safe delivery between partitions. The exact semantics of agreed and safe delivery are defined in [DKM93]. All of these guarantees rely on a method of failure detection based on timeouts. If communication to one or more group members fails for an extended period of time (say 15-30 seconds), the RMP failure membership algorithm will remove them from the group. If this is due to a temporary parition, they can later rejoin the group, but as new members.

Because a member can not join back in to a ring as an old member once it has been removed, it is not possible for a group to partition into two halves and then rejoin. This is the key to atomic delivery within paritions. RMP guarantees that for totally ordered packets, if any member in a partition delivers a packet, all of the other members of that partition will deliver that packet if they were in the group membership view when the packet was sent and if they remain in the group for a sufficient period of time. Since no totally ordered packet will be discarded until it has become stable within a partition, and no packet can become stable within a partition until it has been received (but not necessarily delivered) by all of the members of the partition, if any site delivers a packet, all of the other sites in the partition will receive it before it is discarded by that site. Once a site has a packet, the only way it will not deliver it is if it crashes, upon which case it will be detected and removed from the group.

This level of atomicity does not provide any guarantees about delivery or ordering of packets between partitions. K-resilient atomicity between partitions is the first level of guarantee that addresses this. K is the minimum number of sites that must fail or partition away from a group over a short period of time, in order to violate atomicity guarantees. This is provided by having each member M verify that at least K other members have received a packet before M can deliver it. In this case, each partition will always have at least one member which has received all of the packets that have been delivered at any site.

The next level of atomicity is called agreed ordering, or majority resilience. Agreed ordering guarantees that no matter how many partitions or failures occur, any members of a group that deliver any two messages will agree on the same ordering of the messages. This level guarantees total ordering across partitions, but not atomicity. This level of atomicity is achieved by not allowing minimum partitions to continue and by making sure the majority of the members of a group have a message before any member delivers it. Both of these tests require a possibly conservative calculation of how many members are in the group, which we call MaxN. MaxN is equal to the maximum number of members that are in any membership view for any packet which is not yet stable.

The final level of fault tolerance is safe delivery, also called total resilient delivery. This level requires that a packet be stable before it can be delivered. This occurs after the token has been passed once around the ring after a packet was received. At this point, the member knows that all other members have received it, but they may not all deliver it. It is still possible that one or more of the sites could fail before delivering the packet. This is the highest level of atomicity that any system such as RMP can reasonably provide.

Ordering guarantees between packets of different QoS levels are determined by the lowest QoS of the packets in question. For example, for a set of packets S1 with source ordered QoS and a set of packets S2 with totally ordered QoS, the best guarantee that is provided over the union of the two sets is source ordering.

3.3 Communication model

The two main options in current communication addressing are explicit and implicit addressing. RMP supports both of these addressing models, and supports both the peer group model of communication and the the client-server model of communication. Protocols such as TCP and UDP require explicit naming of the destinations of communication, while systems such as Grapevine[BLNS82] and the MessageBus[Carroll93] allow implicit naming through a publisher/subscriber model of communication. With implicit naming, RMP processes join or "subscribe" to a group by specifying the name of a group to join, and other processes "publish" or send messages to this group by using the group name or a membership view ID associated with the name. When this model is used, messages sent to the group name are delivered automatically to all RMP processes, if any, that are members of that group, so no explicit knowledge of the membership of a group is needed. As explained above, RMP does this by mapping group names into {multicast address, port, TTL} tuples that are used to send to other members of the group.

Instead of specifying a group by its name, RMP processes may explicitly name another RMP process (specified by an IP address and a UDP port) that is a member of the group and request that this member forward packets on behalf of it. In addition to allowing members to join a group based on a process ID, rather than a name, this can also be used by a non-member or non-multicast capable member to send packets to a group. When coupled with the notification of members of the current membership of the group at whenever the membership changes, this allows processes to exert explicit control over group naming and membership when desired.

The default communication model for RMP is that of peer groups. In a peer group, each member receives all of the messages sent to the group, and a process has to be a member in order to send to the group. In addition to

QoS	Name and Service Guarantees	Delivery Time
Unreliable	Packets are delivered 0, 1 or more times, in any order.	Immediately upon receipt of a data packet
Unordered	Packets are delivered at least once, in any order.	Immediately upon receipt of a data packet, with missing packets detected and rerequested
Source ordered	Packets are delivered exactly once, in the order they were sent from each source.	After all of the data packets from the same source and with smaller sequence numbers have been delivered
Totally ordered	Source ordered, plus all totally ordered packets are delivered in the same order at all sites.	After all of the data packets with smaller timestamps have been delivered
K resilient	Totally ordered, plus delivery is atomic at all sites that do not fail or partition, provided that no more than K sites fail or partition at once.	After all of the data packets with smaller timestamps have been delivered and the token has been transferred K-1 times
Majority resilient	K resilient, with K set to (MaxN+1)/2, where MaxN is the highest number of sites in the group for any membership view in the OrderingQ	Same as K-Resilient, but also requires that only a majority partition can continue functioning.
Totally resilient	K resilient, with K set to N.	Same as Majority Resilient

Figure 3.2: RMP QoS levels

a peer group model of communication, RMP also supports the client-server model of communication through the multi-RPC mechanism. This mechanism allows non-members to send to a ring (using either implicit or explicit addressing) and to optionally get replies from a group member. A multi-RPC packet can be delivered with all of the QoS levels available to a member. When the packet is delivered, the current token site normally sends back an acknowledgement to the non-member, although this feature can be disabled. Multi-RPC packets are asynchronous and have integrated flow and congestion control.

Both multi-RPC and normal data packets can receive replies through the use of *handlers*. RMP provides a set of mutually exclusive locks to each group, that members can request and release very efficiently. The first 6 of these are designated *handler locks*. Each data or multi-RPC packet can specify a handler for the packet. The current member who holds the corresponding handler lock is expected to reply to this message. If a request is from a group member, the reply is also sent to the group. If it is from a non-member, it is sent to only that member. Together, the multi-RPC and handler mechanisms allow very efficient replicated servers to be implemented.

3.4 Non multicast capable processes

For efficiency, RMP should be run on top of an unreliable multicast service. However, for flexibility, it also supports the use of hosts that are not multicast capable. This is done through the use of forwarding and multicast capable flags. Each RMP process has to have a UDP/IP port open for sending and receiving packets in addition to any IP Multicast addresses it is using. Any packet that is sent to the UDP/IP port for an RMP process can have a forwarding flag turned on. This flag directs the receiving process to copy it to the IP Multicast address for that group, with local loopback disabled.

Some of the packets sent in RMP are unicast to their destinations. As each RMP process is identified by its unicast address, these addresses are already stored in the membership view for a group. Each of these RMP process IDs in the membership view contains an additional multicast capable flag denoting whether or not they can receive IP Multicast packets. When a multicast is sent to a group, if any members are not multicast capable, the sender must also send a unicast to each of these destinations. In the rest of the paper, wherever we mention a multicast from a group member, we are referring to this extended notion of multicast. Because the case of having all sites be multicast capable can easily be stored as a flag at each site, this common case will pay hardly any penalty for the extra flexibility this service provides.

4 Algorithms

RMP provides all of these services through a set of five main algorithms. The basic delivery algorithm handles the delivery of packets to the members of a membership view. When a membership change request or a handler lock request occurs on a group, the membership view change algorithm creates a new membership view and updates it at each member of the ring. When failures occur in a group, the reformation algorithm polls the current members of a group, synchronizes them to the same point, creates a new membership view for that ring, and commits it at each member. Non members can participate in a group using the multi-RPC algorithm. Finally, all of the senders in RMP use a flow and congestion control algorithm based on the Van Jacobson TCP congestion control algorithms. This section describes these five algorithms, although no proof is offered here because of lack of space.

4.1 Packet types

RMP uses ten packet types in its communication. These are listed in figure 4.1. The group algorithm uses Data packets, ACKs, Confirm packets, and NACKs. The membership view change algorithm uses New List and List Change Request packets. The recovery algorithm uses four recovery packet types as well as New List packets. The multi-RPC algorithm uses Non Member Data packets and Non Member ACK packets. Flow control and congestion control is based on ACK packets and NACK packets.

4.2 Basic delivery algorithm

The basic delivery algorithm is very similar to that first proposed in [MaCh84], although RMP allows multiple data packets to be timestamped with a single ACK. The basic algorithm primarily uses NACKs for reliable delivery, and uses a single ACK per packet or set of packets to provide total ordering and stability of packets. In RMP, all data packets are stamped with a tuple {RMP process ID, sequence number for that process, QoS level for the packet} which uniquely identifies each data packet. Data packets are multicast to the members of the group and are handled by a primary receiver called the token site. When the token site receives one or more data packets, it multicasts a positive ACK out to the members of the group. In certain cases, a NULL ACK may be sent out that doesn't acknowledge any data packets (see below). Each ACK contains zero or more identifying tuples for Data packets, along with a global sequence number, called a timestamp, which serializes all of the ACK, Data, and New List packets in a group. For a given ACK, the ACK is given the value of the timestamp it contains, and each data packet ordered by the ACK is given a consecutive timestamp. For example, an ACK that orders two data packets might have a timestamp of 8. In this case, the first data packet would receive timestamp 9 and the second would be numbered 10.

Each ACK performs a number of functions:

- It lets the sender know that the current token site has received the packet. In this way it functions as a traditional positive acknowledgment to the sender.
- The timestamps in the ACKs provide a total and causal ordering on messages.
- The timestamps also provide a global basis for the detection of dropped packets. The receivers can detect any missed packets, both ACKs, Data, and New List packets, through these global sequence numbers. With multiple simultaneous senders, this provides faster detection of lost packets than does detection based on sequence

Packet Type	**Description**	**Multicast?**
Data Packet	Contains data to the group from a group member	Yes
	Control Packets	
ACK	Provides positive acknowledgment and total ordering for one or more data packets and/or non-member data packets, as well as passing the token and confirming that the token has been accepted by the site that sent the ACK	Yes
Confirm	Provides positive acknowledgment to the last token site that the new token site has accepted the token. This function is usually performed as part of an ACK.	No
NACK	Requests retransmission of one or more packets	Usually
New List	Contains a new membership view and its own ACK. It is also used during failure recovery.	Yes
List Change Request	Requests a change to the current membership view	Yes
	Failure Recovery Packets	
Recovery Start	Sent out when a failure is detected to start the recovery process. Is sent repeatedly by the initiator until the sites in new group are synced to the same point.	Yes
Recovery Vote	The response to a recovery start packet, notifying the initiator of the sync point for this member of the new list	No
Recovery ACK New List	Acknowledges receipt of the New List packet for the new membership view	No
Recovery Abort	Provides notification that an error in the reformation protocol occurred	Yes
	Non Member Packets	
Non Member Data	A data packet from a process which is not a member of the group	Yes
Non Member ACK	An ACK or response to a non-member data packet.	No

numbers from each sender.

As with other NACK based solutions, this single ACK does not solve the problem of detecting when a message becomes *stable*. A stable message has been received by all members of the group. To solve this problem, each ACK also passes the token site to the next process in the group. Before it can accept the token, each member is required to have all of the packets with timestamps less than that of the ACK naming it the token site. If it does not have some packets, it requests them before accepting the token. This guarantees that given N members of a ring, once the token has been rotated N times, the token site knows that all messages with a timestamp at least N smaller than the current timestamp have been received at all destinations. This allows each site to detect when a packet has become stable, and bounds the number of packets the each site needs to hold for retransmission.

Ordering of packets, detection of missing packets, and buffering of packets for retransmission is all handled with the use of two lists, the DataList and the OrderingQ. The DataList contains Data packets and List Change Request packets (described below) that have not yet been ordered. The OrderingQ contains *slots*, each of which holds a pointer to a packet, the delivery status of the packet (Missing, Requested, Received, or Delivered) the {source, sequence number, QoS} identifying tuple for the packet, and the timestamp for the packet. The fields in a slot are not all used at all times. The slots in the OrderingQ always have monotonically increasing timestamps.

When a Data or List Change Request packet is received, it is placed into the DataList. When an ACK or a New List packet (also described below) is received, it is placed in the OrderingQ, creating one or more slots on the end of the queue if necessary. Each packet occupies exactly one slot in the OrderingQ. When an ACK is placed in a slot, the tuples it contains identifying data packets are copied to the slots immediately succeeding it, creating new slots if necessary.

Whenever a Data or ACK packet is received, the OrderingQ is scanned through once to match up Data packets in the DataList with empty slots that have been created by an ACK. When a slot is found that has the same identifying tuple as a Data packet in the DataList, the packet is moved from the DataList to that slot. When holes occur in the OrderingQ, NACK packets are sent out, requesting retransmission of these packets. The exact policy for determining the destination of the NACKs and whether or not the retransmissions should be unicast or multicast is a topic for continuing research. The current policy is to multicast NACKs to the group and name the last known token site as the site to handle them. If there is no response to a NACK within the specified timeout, a site will resend the NACK to a different site. This will continue up to a constant threshold, after which point the NACK will be multicast to the entire ring for any site to respond to. By multicasting each NACK but requesting only a single site to respond to them, the NACK explosion problem is limited.

A site is not allowed to accept the token until there are no empty slots in the OrderingQ up to the last data packet ordered by the ACK naming this site as the new token site. Because of this, the OrderingQ does not need to contain any more than N ACKs and New List packets, where N is the current number of sites in the group. When there are more than N ACKs and New List packets in the OrderingQ, the slots at the front of the queue and their corresponding packets are dequeued and freed until this condition is met.

In addition to bounding the buffer space needed in a ring, passing the token guarantees that site failures and dropped messages are detected within N messages. In order to bound the amount of time before a lost packet or a failed site is detected, RMP sends NULL ACKs, which pass the token but do not order any packets, if the token site does not receive a message within a given period of time. With RMP, this is currently on the order of 1 second. When a ring goes quiescent for an extended period of time, the token is passed all the way around the ring once and then stops. At this point, all of the sites are guaranteed to have all of the messages.

If a site repeatedly fails to receive the proper response to one of the actions that requires a positive acknowledgment, it declares the site dead and runs the reformation protocol, described below in section 3.4.

4.3 Membership view changes

A new membership view is created whenever a RMP process joins a group, leaves a group, is granted a handler lock for a group, or releases a lock for a group. Members that fail or partition away are handled by the fault recovery algorithm, described below. For these membership view changes, a site requests the change by multicasting an unreliable List Change Request packet to the group. List Change Request packets require positive acknowledgment, and are resent periodically until this is received or a fault in the ring is declared. The current token site serializes these change requests and sends out New List packets in response.

List Change Request packets are handled similarly to Data packets, except that when the token site would normally send an ACK packet to order and acknowledge a Data packet, it instead generates a New List packet. When a token site creates a New List packet, it makes the requested change to the current membership view, if possible, and puts this list into the second half of the New List packet. It fills the first half of the New List packet with the fields of an ACK, including a timestamp ordering the New List packet and the ID of the next token site. After the New List packet is sent out, the List Change packet is dropped, instead of being put on the OrderingQ. The New List packet acts as both a totally ordered Data packet whose destination is RMP instead of an application, and as its own ACK. When a New List packet is received, it is put on the OrderingQ, and then broken down into its two halves. The ACK half is processed first, which will put the New List packet into the OrderingQ. From this point, it is handled like a data packet. When the data half of the New List packet would normally be delivered to the RMP process, it is instead committed and a notification of the membership view change is delivered to the application.

When a RMP process is joining a group, it will repeatedly send the List Change Request packet until it receives a New List packet naming it as the new token site. Because the joining process may not yet know the IP Multicast address for the group, this New List packet is also unicast to the joining process's UDP address and port, and it contains the multicast address and port for the group. A new member is always added in to the membership view directly after the current token site. This forces the new member to take an immediate role in the group, providing positive acknowledgment that it has joined the ring and started processing messages at the correct time. If no New List packet is received after a certain number of retries to a group, the new RMP process creates a new group with only itself in it. If a ring does actually exist for that group name and multicast address, but is currently unreachable due to a network partition, a second ring may form. Because group IDs are unique, even if this partition heals, the rings will never overlap or merge.

When a site removes itself from a ring, it must remain a member of the group until after it has seen and committed the new membership view removing it from the ring. After leaving the ring, it must continue to process NACK requests and keep track of token passes until it no longer has any packets from the old list that it must hold for retransmission. A packet must be held for retransmission until the site knows it is stable.

List Change Request packets requesting a handler lock will not be granted if another process already holds that handler lock when the request is processed. If a request is denied, a New List packet is still generated, but the membership view it contains will be the same one the group was using before.

4.4 Fault recovery

In the normal mode of RMP, there are four types of packets which require positive acknowledgment: Data packets, ACKs, List Change Requests, and NACKs. If, at any time, a RMP process repeatedly times out and resends one of these packets more than a set number of times (ten in the current implementation) it decides that a failure has occurred and starts the fault recovery protocol. The initiator of the fault recovery protocol repeatedly polls the sites in the group to determine who is still active and reachable, generates a new membership view, makes sure that all sites in the new membership view have the same set of packets in their OrderingQ, and then commits this list at all sites in the new list using a two phase commit protocol. Because token passes are one of the actions that require positive acknowledgment, a failure will be detected within N token passes.

The fault recovery protocol is broken into two halves--creating and synchronizing the list, and committing the list. In the first half, the initiator repeatedly queries the other sites in the old membership view to see if they are up, to see what the highest version of membership view they have seen is, and to find out what their current *sync point* is. The sync point for a RMP process is defined as the highest consecutive timestamp it has in its OrderingQ plus the highest consecutive sequence number it has received from each site in the old list. This information corresponds to the timestamp of the highest packet that a RMP process has delivered with a QoS of at least totally ordered concatenated with the highest sequence number of a packet delivered with a QoS of at least source ordered from each RMP process in the ring. By responding to a query, a site provides the requested information and confirms that it has joined the new membership view the initiator is creating. A process is only allowed to join a membership view with a larger version number than any it has seen before, and it is only allowed to join one list at a time. If any of these conditions are not met, each process detecting the error multicasts an abort reformation packet out, aborts its own reformation, and waits for a random timeout period or another reformation start packet from another site before restarting.

The goal of the first phase is to have as many of the old sites as possible join the same new membership view and reach the same sync point. If some of the sites are missing packets, this may be an iterative process. If the initiator receives a higher sync point from another process than the one it has, it stores this as the new sync point for the list and forwards this sync point to the other processes. All sites that are missing packets request those packets and send higher sync points as they receive them. After either all but one of the old sites in the membership view have responded and been brought to the same sync point, or after no further progress has been made towards the shared sync point after a set number of retries, the initiator creates a new list with all of the members of the old list that have reached the same sync point. If the initiator receives a packet from another initiator with a smaller or equal version number, it sends back a packet notifying that initiator that it should abort its reformation. If the initiator receives a packet from another initiator with a larger version number or if it is not itself able to reach the shared sync point, it sends out an abort packet to the members of the list and aborts its own reformation process, setting a random timeout before it starts the reformation again.

Once the initiator has created a new list, it must check that this new list has at least the minimum number of sites specified in the old version of the membership view. When joining a membership view, each site specifies the minimum number of sites that must remain in a partition in the case of a failure. The minimum size for the new group is the maximum of these values for each member in the old ring. This value can be either a constant, a symbolic constant specifying the majority of the members of the old list, or a constant for all of the members in the old list. In the latter case, any failure will always cause the ring to stop operation.

If this test is passed, the initiator creates the new list and must commit it at all of the sites. First, it multicasts a New List packet to all of the members of the new list and requests a reply from each of them. Once it has received these acknowledgments from each member, it commits the new list itself and makes itself the new token site. If it has any Data packets waiting to be acknowledged, it sends out an ACK on these packets. Otherwise, it sends a NULL ACK to pass the token. After each of the members of the new list receive this or any other ACK to the new list, they also commit the new list and start processing packets as normal.

4.5 Multi-RPC delivery

The client server model of communication has become widely accepted as a powerful way of providing services to users. While RMP could support this model simply by having all clients and servers join a group, this is often inefficient and will limit the scalability of client/server groups. As an alternative, RMP provides facilities for RMP processes that are not members of a group to use a multi-RPC algorithm to send data to a ring and to receive acknowledgments of successful delivery and/or responses from a member of the group. This is a powerful feature, for it allows multiple servers to exist in a group, and all of them can get messages from clients. These messages can be automatically acknowledged, or a single member of the group can be selected to handle the request and reply to it. These Non Member Data packets can be delivered with all of the QoS levels of a Data packet sent from a group member.

The two main changes between a Data packet and a Non Member Data packet are that a Non Member Data packet can include a group name instead of a membership view ID, and it includes two flags that aid in getting acknowledgments or responses. Since a non member will often not know the current membership view ID for a group, it can instead specify the textual representation for the ring. When it gets a reply back from the group, it can cache the membership view ID included in this reply and use this in subsequent Non Member Data packets. The first flag specifies whether or not an acknowledgment should be sent in response to the Non Member Data packet. If so, then when the current token site delivers the Non Member Data packet to its application, it unicasts a Non Member ACK to the sender. Because the sender has the responsibility for making sure the packet is delivered reliably, this acknowledgement may need to be sent to the sender multiple times. This is done by sending another copy of the Non Member ACK each time a duplicate of a Non Member Data packet that has already been delivered is received. The second flag provides the same repeat reply functionality, but for replies instead of acknowledgements. When this flag is turned on, the Non Member Data packet will be delivered multiple times to the application. This is usually used in conjunction with a handler number for the packet. If a handler number is specified, the member of the group that holds that handler lock will be responsible for replying to the Non Member Data packet. This reply will usually be used as the ACK for this multi-RPC call, and must be sent as many times as necessary until the non member receives it.

In order to provide source ordered (and higher) delivery guarantees on non member packets, the members of a group have to keep track of the highest delivered sequence number from each non-member. These are stored in the membership view along with the sequence numbers for members, but are marked with a non-member flag. Because of this, they are sent out as part of each New List packet, and each joining member will receive a copy of these sequence numbers. It is the responsibility of a non member that is sending source ordered packets to set a flag on the first packet sent to a group, notifying the destinations that it is the first packet. This notifies the members that they can create a sequence number for this process starting with this packet. Optionally, these non-member sequence numbers may be flushed if no packet has been received from that site within two times the maximum TTL of a packet in the internetwork. If this policy is used, then non-member senders must keep track of when their sequence numbers may have been flushed out, and set the first packet flag on the next packet sent after this occurs. If a packet is recieved that has the first packet flag set and there is already a sequence number for that site, the flag is ignored. If a non-member sends out a packet and it is not received after a set number of retries, the non-member should make sure its first packet flag is set, to create a new entry in case its entry has been flushed. Handling these two cases correctly ensures that correct operation will occur without the need of global clock synchronization.

4.6 Flow and congestion control

Flow and congestion control policies for reliable multicast protocols are an open problem. Because reliable multicast protocols primarily use NACKs for error detection, there is no existing explicit feedback path with which destinations can signal losses or low buffer space to the senders. In addition, the throughput for a multicast group should be divided up between the members of the group who are trying to send, but the policy for this division is usually dynamic and not known in advance. Because of this, the flow and congestion control policies used by RMP are designed to be orthogonal to the rest of the protocol. Flow and congestion control policies can be inserted easily into the protocol, and different policies can be used in different environments. As the default, we propose a modified sliding window protocol based on the Van Jacobson algorithms used in TCP [Jacobson88].

Two of the most common schemes used today for flow control and congestion control are sliding windows and leaky buckets. Leaky bucket schemes, which enforce explicit rate controls on each sender, are classified as predictive controls. They try to predict how much bandwidth each sender can use at any given time, and then mandate that the senders do not exceed this. Calculating the values for these rates is difficult, and they must divide up the bandwidth between the senders on a relatively static basis. This decreases the flexibility and throughput attainable by these schemes. In addition, it is difficult with a leaky bucket scheme for the destinations to signal back that their buffers have been overrun if a destination process stalls for some reason. However, for the new very high speed WAN networks that are being proposed, the cost of congestion can be very high because a sender can have hundreds of packets in transit at once. For these networks, a leaky bucket or rate control scheme may be necessary.

For networks that have a lower latency bandwidth product, the drawbacks of a leaky bucket scheme may not be necessary. For RMP, we propose an adaptive flow and congestion control scheme based on a modified sliding window scheme. This algorithm treats flow control and congestion control as the same problem and solves it in part by using some of the algorithms proposed by Van Jacobson for TCP. Each sender maintains a window of how many bytes it can have in transit at once. When a packet is sent, the window size is decreased by the number of bytes sent, and when the first copy of an ACK for a packet is received the window size is increased by the same amount. This causes the flow control feedback to be rotated among members. If a site is overrun and runs out of buffers, when it gets the token site it can delay acknowledging any more packets until it can process incoming packets again. If the RMP process delays too long, however, it will be considered to be faulty, and it will be removed from the group. This is usually desirable behavior, as otherwise the other members of the ring will block indefinitely on this one site.

The bulk of the flow and congestion control is provided by controlling the maximum size of the window at each sender. The maximum size of this window grows according to the slow start algorithm proposed by Van Jacobson, and is decreased when an expired retransmission alarm or NACK occurs. The Van Jacobson algorithms were originally designed to just provide congestion control. However, because they provide such good adaptive congestion control, RMP also uses them for additional flow control by treating NACKs as another signal of congestion.

The Van Jacobson algorithms for congestion control that are used by RMP include:

(1) round-trip-time variance estimation

(2) slow start

(3) dynamic window sizing on congestion

(4) exponential retransmit timer backoff

Round-trip-time variance estimation comes from the observation that when a network path becomes congested, the variance on packet latency becomes very high compared with the average. "If the network is running at 75% of capacity...one should expect the round-trip-time to vary by a factor of 16."[Jacob88] The proposed algorithm continually estimates this variance, and eliminates most of the spurious retransmissions while still maintaining timeouts small enough to detect dropped packets quickly.

The slow start algorithm (2) is used to increase the window size from 1 packet to the maximum window size that the receiver allows that does not cause congestion, as calculated by algorithm (3). This is done by incrementing the window size by one packet each time that an ACK is received. Because the window size is constantly growing, slow-start actually increases the window size fairly quickly. It will increase from 1 to W on a network with latency L in $L\log_2 W$ time.

With the assertion that the timer algorithm almost completely avoids retransmissions that are not due to lost packets and with the observation that most lost packets are due to congestion instead of errors, it follows that most expired timers signal congestion. Algorithm (3) uses this to respond aggressively to this congestion by exponentially reducing the window size by a constant number (currently 50%) each time that a timer expires. The original protocol actually uses a two-level bound on this window in the face of congestion. It reduces the window size to one packet any time an error occurs, uses slow start to quickly build up to 50% of the level before the error, and then uses a slower linear increase to build up from there. RMP only reduces the current window size by 50% because RMP modifies the packet lengths according to the window size. The algorithm used for this causes slow start to reach a window size W (assuming W is less than two times the maximum packet size) in O(W) time instead of O(logW) time, and this makes the cost of reducing the window all the way to 1 packet too high.

Finally, the exponential retransmit timer backoff is used to double the timer each time it expires, resetting it to the value calculated by (1) when an ACK is finally received. Both this and algorithm (1) are applied to the timers for all of the packets that require positive acknowledgment. Along with further decreasing congestion, this provides an efficient detection method for failed sites. The maximum value for a timer is clamped at a certain value (currently 2 seconds). Then up to N retransmissions (currently 10) are allowed before a site is declared

```
// Avoid "Silly Window" effect
if (A < MIN_PACKET && A < W)
    Delay sending packet until an ACK is
    received
// Send up to 1/2 of the window at a time
S = min(P, W/2);
// Send at least MIN_PACKET bytes
S = max(S, MIN_PACKET);
// Can only send up to A bytes
S = min(S, A);
// Reduce effect of lost packets
S = min(S, MAX_PACKET);
```

Figure 4.2: Packet Size Algorithm

dead. With the currently implemented values this policy detects failures in nearby sites within 5 seconds, and distant sites within 15-20 seconds.

By using NACKs as signals of dropped packets, these algorithms also provide effective flow control. If a destination gets overrun by the senders, it will drop one or more packets. This will usually be detected by the destination very rapidly. When this occurs, the destination multicasts a NACK back to the group. In addition to requesting a retransmission of the packet from the current token site, this NACK also informs the original sender (who is named in the NACK) that a destination has lost a packet from that sender. This is treated the same as an expired timer due to a lost packet, and causes the sender to decrease its window size by 50%. In order to make sure that multiple NACKs on the same packet do not each decrease the window size, a cache of the sequence numbers of the last three messages sent by this site that were dropped by another site is maintained by the sender. Incoming NACKs are compared against this cache, and the window size is only modified if a NACK for this message isn't in the cache.

A problem that RMP faces with flow and congestion control is that the rotating token site introduces a higher overhead per acknowledgment than traditional protocols such as TCP. This is compounded by the protocol being more complicated than TCP and thus requiring more processing per packet. To solve this problem, RMP uses larger packet sizes than does TCP. In an error free environment, having the IP or IP Multicasting layer do the fragmentation and reassembly is more efficient than having RMP do it. If errors occur, the window size quickly drops to a single minimum size packet. The algorithm to determine the size of the packet to be sent out (S), given the current window size (W), the available space in the window (A), and the offered packet size (P), is shown in figure 4.2. The critical step in this algorithm is that up to half of the available window is sent at a time until the maximum packet size has been reached. This trades off a small amount of network utilization in the case of errors for typically higher efficiency of handling packets and higher throughput.

4.7 Implementation options

RMP is designed to be implementable in either user-space or kernel space. There are numerous pros and cons to each approach, so RMP does not enforce either approach. Traditional protocols have been implemented as monolithic entities in the kernel. This is motivated primarily out of concerns for security and performance. However, as pointed out by [JHC94], [MaBe93], and [TNML93], user-space implementations can be more easily modified and customized, are easier to debug and experiment with, are more easily ported between different platforms, and can increase the performance of protocols, especially when run on a multiprocessor. [TNML93] shows that it is possible to implement protocols in user-space that actually have better performance than kernel space implementatinos.

Numerous "software buses" have been implemented recently, which are designed to make applications which do group communication much easier to program. RMP was originally designed as the transport layer for one of these buses, the MessageBus[Carroll93], and supports these types of systems very well. Some of the features that these systems provide on top of RMP are translation of raw data into different formats and objects, filtering of packets based on another level of group or domain identifier, dynamic loading of processes that are requested to provide a service, buffering multiple small packets into a single large packet under heavy load, and a uniform communication interface independent of the actual transport used. Other instances of these software buses are Polylith [Purtilo85] and MultiBus [CaMo94].

5 Performance

The optimal number of packets sent to a group is one multicast packet for each data packet sent. RMP also sends out ACKs and Confirm packets, but the number of these decrease with increased load. In cases of few errors and low load, RMP may require up to 3 multicasts per data packet. Under high load, this will decrease to slightly more than one multicast per data packet, because each ACK will be acknowledging more than one data packet. As the error rate increases, the number of packets sent increases, but it is always lower than that required with positive acknowledgments for groups of three or more sites including the sender [ChMa84].

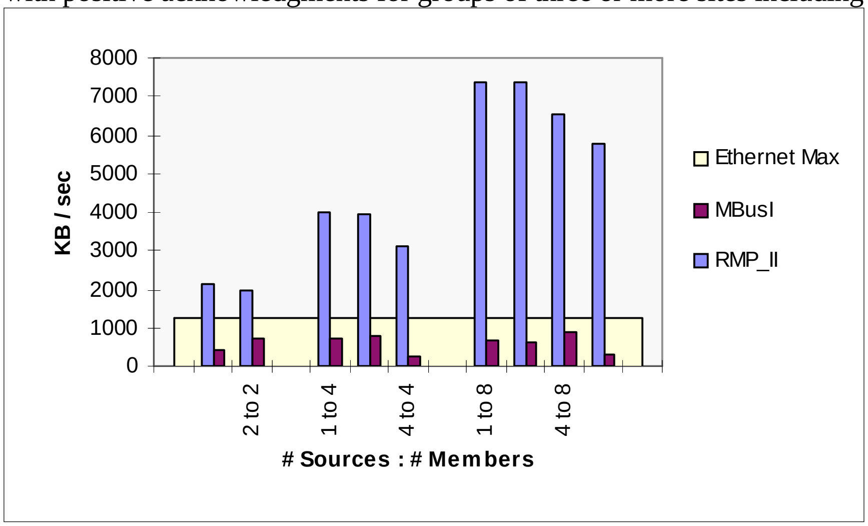

Figure 5.1: Aggregate Throughput (KB/sec)

We have tested RMP under a variety of cases. The most extensive tests were done on a set of 9 SparcStation2's and SparcStation5's on a lightly loaded 10 Mb/sec Ethernet. Unless otherwise noted, totally ordered packets were used for all tests. Throughput was measured by timing the transfer of a 5 MB data file, and so does not include packet headers. Latency was measured by "ping-ponging" a single message back and forth between the group members. Because of this, latency measures the time to get from application to application. To eliminate the start up effects of the adaptive time-outs, an initial run was made in each test and discarded. Three runs were made in each test and the results were averaged together. While we do not show the standard deviations, they were quite small for all of the cases except for 4 and 8 senders, where the Packet Starvation Effect [WhStFe94] was causing the Ethernet to drop a significant portion of the packets sent. Under this case, the network has an extremely high standard deviation in latency, so consistent results are not possible without extremely long runs.

Figure 5.1 shows the aggregate throughput of the network as a function of the number of sources and number of group members. Aggregate throughput is the throughput the user sees. It is computed by taking the amount of data sent from all of the senders and multiplying it by the number of destinations. In these tests, the sender is also a destination. This is done so that it can see its own messages totally ordered with the others. This is the way that most groups use totally ordered multicast, and is actually the worst case for the throughput of the protocol because of the increased CPU load of the senders. A single-server, TCP/IP based system, the MBusI, is shown for comparison. The MBusI accepts a TCP/IP stream from each communication client and routes packets between them. The maximum bandwidth of the Ethernet used in this study is also shown. For the case of 1 source and 8 members, RMP achieved an aggregate throughput of 7384 KB/sec. This is 5.91 times the bandwidth of the Ethernet. It is impossible for any solution that does not use either multicast or broadcast to achieve any result that breaks the Ethernet throughput boundary this way.

In graph 5.2, we see the single sender throughput plotted against the number of destinations. The single sender throughput is equal to the aggregate throughput divided by the number of destinations. To be compatible with other published figures, in these tests the source was separate from the destinations. Data from the MBusI is included for comparison. The performance of all protocols (such as UDP ISIS, Sun ToolTalk, the MBusI, and RPC) that do not use hardware broadcast or multicast drops off as a factor of 1/N. In contrast, RMP stays roughly constant regardless of the number of destinations. RMP breaks the fundamental unicast limit for two destinations, and so has higher throughput in this environment for all groups of more than one destination.

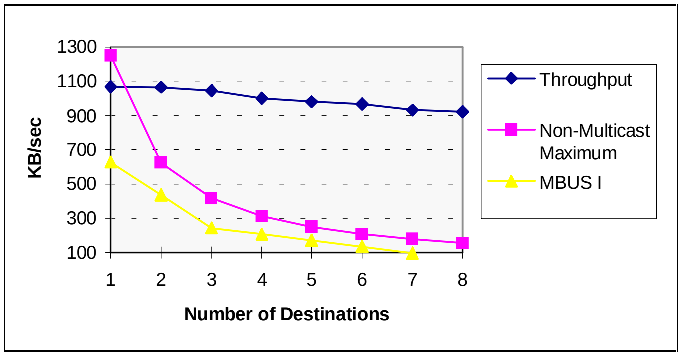

Figure 5.2: Single sender throughput

In figure 5.3, we see the same factors, but with latency as the metric instead of throughput. Here again, the performance of RMP stays almost constant. While we have not yet been able to make fair comparisons to other protocols, the data that we have shows that the latency of protocols that do not take advantage of hardware multicast also scales linearly as a function of N [BiCl94].

We have also performed some tests of RMP over internetworks. One test was over a dedicated mbone connected between West Virginia University and the University of Illinois at Champaign-Urbana, and another was over two Ethernets connected by a gateway. The single sender, single destination throughput for both cases was 98 KB/sec. This indicates that the routing daemon, which is implemented in software on a general purpose host, is limiting the speed of RMP. We hope to perform better tests as soon as we can get access to hardware routers that support IP Multicast.

6 Conclusions

In this paper we have described the basic mechanisms and algorithms of RMP, a fully distributed reliable multicast protocol with selectable ordering, atomicity, and fault tolerant guarantees. We have shown that RMP provides these features with very high performance. To our knowledge, RMP provides better performance for totally ordered delivery of packets to 2 or more destinations on an Ethernet than any other protocol. Much work has gone into providing reliable multicast services with lower ordering guarantees because it was believed that the performance of a totally ordered multicast protocol was inherently low. RMP suggests that this is not the case, and that an efficient reliable multicast service can provide total ordering of messages for only a small latency penalty. Finally, because of its use of multiple groups, an optional client/server architecture, its fully distributed nature, and its flow and congestion control algorithms, we expect RMP to scale gracefully and efficiently to large groups spread over a large internetwork. Initial results confirm this hypothesis, and we plan to further test this in the near future.

7 Acknowledgments

This work is supported in part by the NSF through grants CCR-9007195 and CCR-9108931, by NASA through cooperative research agreement NCCW-0040 and grant NAG 5-2129, by the NASA headquarters office of safety and mission assurance (OMSA), by the US Army Corps of Engineers, and by Sun Microsystems, Intel, Bull, Hewlett-Packard, DEC and Fujitsu/Open Systems Solutions. In addition, we would like to thank Alan Carroll for his work on the MBusI which inspired this whole project, Doug Bogia and Bill Tolone for their constant support, and Nick Maxemchuk and Jo-Mei Chang, whose work defined so much of RMP.

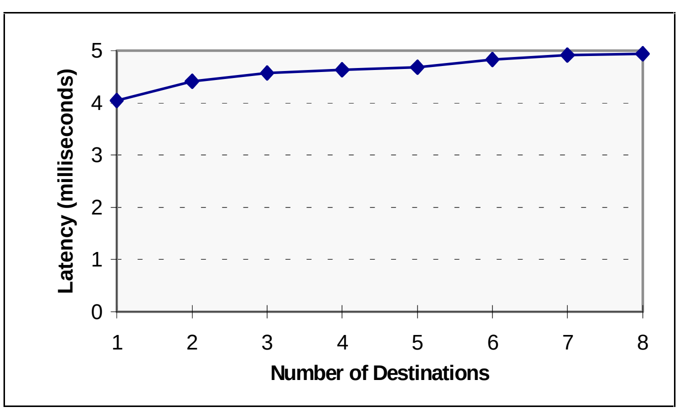

Figure 5.3: Single sender latency

References

[ADKM91] Y. Amir, D. Dolev, S. Kramer and D. Malki. "Transis: A Communication Sub-system for High Availability." *Technical Report CS9113*, Hebrew University of Jerusalem, Nov. 1991.

[AFM92] S. Armstrong, A. Freier, K. Marzullo. "Multicast Transport Protocol". *RFC1301*. (February, 1992).

[AMSM92] D. A. Agarwal, P. M. Melliar-Smith, and L. E. Moser. "Totem: A protocol for message ordering in a widearea network." *Proceedings of the First ISMM International Conference on Computer Communications and Networks* (San Diego, CA, June 1992). pp. 1-5.

[ARP93] R. Aiello, E. Pagani, G. P. Rossi, "Design of a Reliable Multicast Protocol". *Proceedings of IEEE INFOCOM '93* (San Francisco, March 1993). pp. 75-81.

[APR93] R. Aiello, G. P. Rossi, E. Pagani. "Casual Ordering in Reliable Group Communications." *Proceedings of ACM SIGCOMM '93* (San Francisco, Sept. 1993). pp. 106-115.

[BLNS82] A. Birrell, R. Levin, R. Needham, M. Schroeder. "Grapevine: An exercise in distributed computing." *Communications of the ACM*, 25, 4, April 1982. pp. 260-274.

[BSS91] K. Birman, A. Schiper, P. Stephenson. "Lightweight Causal and Atomic Group Multicast." *ACM Transactions on Computer Systems.* 9, 3 (Aug. 1991). pp. 272-314.

[BiCl94] K. Birman, T. Clark. "Performance of the Isis Distributed Computing Toolkit." *Technical Report TR-94-1432, Dept. of Computer Science, Cornell University.*

[Birman93] K. Birman. "The Process Group Approach to Reliable Distributed Computing." *Communications of the ACM.* December, 1993, 36,12. pp. 37-53.

[CaMo94] J. Callahan, T. Montgomery. "A Decentralized Software Bus based on IP Multicasting." *Proceedings of Third Workshop on Enabling Technologies: Infrastructure For Collaborative Enterprises*, Morgantown, WV, April 17-19, 1994, pp. 65-69.

[Carroll93] Alan Carroll. "ConversationBuilder: A Collaborative Erector Set." *Ph.D. Thesis, Department of Computer Science, University of Illinois*, 1993.

[ChMa83] J. M. Chang and N. F. Maxemchuk. "A Broadcast Protocol for Broadcast Networks." *Proceedings of GLOBCOM* (Dec. 1983).

[Chang84] J. M. Chang. "Simplifying Distributed Database Systems Design by Using a Broadcast Network." *Proceedings of SIGMOD* (June 1984). pp. 223-233.

[ChMa84] J. M. Chang and N. F. Maxemchuk. "Reliable Broadcast Protocols." *ACM Transactions on Computer Systems.* 2, 3 (Aug. 1984). pp. 251-273.

[ChZw85] D.R. Cheriton and W. Zwaenepoel. "Distributed Process Groups in the V-Kernel." *ACM Transactions on Computer Systems.* 3, 2 (May 1985). pp. 77-107.

[CrPa88] J. Crowcroft, K. Paliwoda. "A Multicast Transport Protocol". *Proceedings of ACM SIGCOMM '88.* pp. 247-256.

[Deering89] S. Deering. "Host Extensions for IP Multicasting". *STD 5. RFC1112.* (August 1989).

[DKM93] D. Dolev, S. Kramer, D. Malki. "Early Delivery Totally Ordered Multicast in Asynchronous Environments." *23rd Annual International Symposium on Fault-Tolerant Computing (FTCS).* (Toulouse, France, June, 1993). pp. 544-553.

[Jacob88] V. Jacobson. "Congestion Avoidance and Control." *Proceedings of ACM SIGCOMM '88 Symp* (Sept. 1988). pp. 314-329.

[JHC94] P. Jain, N. Hutchinson, and S. Chanson. "A Framework or the Non-Monolithic Implementation of Protocols in the x-kernel." *Proceedings of USENIX High Speed Networking (August 1994).* pp. 13-30.

[KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, H. E. Bal. "An Efficient Reliable Broadcast Protocol." *Operating Systems Review.* 23, 4 (Oct. 1989), pp. 5-19.

[Lamp78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM.* 21, 7

[LuGl90] S. W. Luan and V. D. Gligor. "A Fault Tolerant Protocol For Atomic Broadcast." *IEEE Transactions on Parallel and Distributed Systems.* 1, 3 (July 1990). pp. 271-285.

[MaBe93] C. Maeda and B. Bershad. "Protocol Service Decomposition for High-Performance Networking." *Proceedings of 14th ACM Symposium on Operating Systems Principles*, December 1993.

[MaCh84] N. F. Maxemchuk and J. M. Chang. "Analysis of the Messages Transmitted in a Broadcast Protocol." *Proceedings of the International Computer Conference* (Amsterdam, May 1984). pp. 1263-1267.

[MeBo76] R. M. Metcalf. and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM.* 19, 7 (July 1976). pp. 395-404.

[MMA90] P. M. Meillar-Smith, L. E. Moser, V. Agrawala. "Broadcast Protocols for Distributed Systems." *IEEE Transactions on Parallel and Distributed Systems.* 1, 1 (Jan. 1990). pp. 17-25.

[PEA94] S. Pejhan, A. Eleftheriadis, D. Anastassiou. "Distributed Multicast Address Management in the Global Internet." *Submitted to IEEE Journal on Selected Areas in Communication* on March 1, 1994.

[PBS89] L. L. Peterson, N. C. Buchholz, R.D. Schlichting. "Preserving and Using Context Information in Interprocess Communication". *ACM Transactions on Computer Systems.* 7, 3 (Aug. 1989). pp. 217-246.

[Purtilo85] J. Purtilo. "Polylith: An Environment to Support Management of Tool Interfaces." *ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, WA, June 25-28, 1985. pp. 12-18.

[RaLi93] K. Ravindran and X. T. Lin. "Structural Complexity and Execution Efficiency of Distributed Application Protocols." *Proceedings of ACM SIGCOMM '93.* (San Francisco, Sept. 1993). pp. 160-169.

[TNML93] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. "Implementing Network Protocols at User Level." *IEEE/ACM Transactions on Networking*, 1(5), 1993. pp. 554-565.

[Verissmo92] Verissmo. "xAMp: A Multi-primitive Group Communications Service. *Proceedings of the 11th Symposium on Reliable Distributed Computing.*

[WhStFe94] B. Whetten, S. Steinberg, and D. Ferrari. "The Packet Starvation Effect in CSMA/CD LANs and a Solution". *Proceedings of IEEE Local Computer Networks Conference* (Minneapolis, Minnesota, Oct. 1994).