This is to certify that the

thesis entitled

DESIGN AND IMPLEMENTATION OF A RELIABLE MULTICAST PROTOCOL

presented by

Robin F. Wright

has been accepted towards fulfillment
of the requirements for

Master's degree in Computer Science & Engineering

Major professor

Date 12/11/98

Abstract

Design and Implementation of a Reliable Multicast Protocol

By

Robin F. Wright

Multicast communication, in which a single sender communicates with two or more receivers, is becoming an important part of computer networking. Multicast communication can be beneficial to many types of applications, including computer supported cooperative work, data distribution services, distributed interactive simulation, tele-gaming, and distributed parallel processing. IP multicast provides a mechanism for best-effort multicast. However, the differing requirements of applications has resulted in many reliable multicast protocols, but little consensus on a single standard. Moreover, few reliable multicast protocols are currently provided as operating system services.

In this research, we design and implement RMC, a reliable multicast protocol suited for integration in an operating system kernel. RMC takes advantage of an underlying best-effort multicast delivery system, such as IP multicast, and provides end-to-end reliable delivery of a stream of data to multiple destinations. In this thesis, we use the RMC protocol to study three key issues in reliable multicast protocols: reliability, group management, and flow control. We focus on NAK-based reliability because it scales better than ACK-based reliability, and we limit work at the sender by

supporting anonymous group membership. While NAK-based protocols typically use rate-based flow control, however, kernel-level protocols typically use window-based flow control, due to limited buffer space. We explore the combination of rate-based and window-based flow control to allow for the use of NAK-based reliability in a kernel-level implementation. Our contributions can be summarized in the following three areas.

First, we have designed the RMC reliable multicast protocol. RMC uses negative acknowledgments for reliability and a combination of window-based and rate-based flow control. While typical window-based protocols use positive acknowledgments to slide the window forward, RMC uses purely negative information, and a timer that is based on the round-trip time to the most distant receiver, to prevent the window from sliding forward too quickly. Further, RMC uses anonymous group membership to further reduce the work that is required of the sending host.

Second, we have implemented the RMC protocol in the Linux operating system kernel. RMC is implemented atop IP, enabling its use across wide area networks, and beneath the BSD socket interface, enabling applications to access it via standard system calls. Moreover, the implementation required us to address a number of important practical problems that might not arise if we had limited the study to analysis and/or simulation; examples include buffer management, interrupt handling, and data locking.

Third, we have evaluated the performance of the RMC protocol through both experimentation and simulation. We first evaluate the performance of RMC as implemented in the Linux kernel on machines in a local area network. In this environment, we are able to observe the behavior of RMC in a working environment. The experimental tests reveal that RMC performs very well in a local area network, where packet loss and retransmissions are minimal. We further evaluate the performance of

RMC in a simulated network, where we can set network parameters such as network delay and loss rates to simulate wide area networks. Results of the simulation tests show that RMC is scalable up to 100 receivers in a local area network environment, where loss rates and network delay are low. Although throughputs are lower in a wide area network environment, due to higher loss rates, the RMC protocol still provides good performance and is able to accommodate a large number of NAKs, even without the use of global NAK suppression. Moreover, the efficiency that is gained by using reliable multicast over reliable unicast is very large.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer networking has become an essential part of the computing field, both in local area and wide area environments. In local area environments, computer networks are being used to create distributed computing environments and parallel servers. In addition, as networking technology has become more advanced, wide area networks have grown both in size and utilization, to reach not only research institutions and large corporations, but also homes and small businesses.

Many of the applications that execute in local area and wide area networks involve multicast communication, in which a single source host transmits information to a limited group of receiving hosts. Classes of applications that utilize multicast communication include collaborative applications such as shared whiteboards, video conferencing, and distance education; applications for distributing data such as software and databases; distributed interactive simulation applications; and applications in the entertainment industry including video and audio broadcasts.

Currently, Ethernet multicast and IP multicast and provide multicast services in a local area and wide area networks, respectively. Both of these services are unreliable, which means that the data is not guaranteed to arrive safely and in order at the

destinations. However, many of the applications that use multicast communication require the communication to be reliable. Because there is no current standard for reliable multicast, these applications typically manage their own reliability at user level.

In light of the growing need for reliable multicast, much work has been done on developing reliable multicast protocols. Most reliable multicast protocol implementations must address three issues: reliability, flow control, and group management.

Reliability is the basis for reliable multicast protocols, and ensures that all members of the multicast group receive complete and correct data. Usually, reliability is provided by either a sender-initiated or a receiver-initiated method. In sender-initiated, or *ACK-based*, protocols, each receiver in the multicast group confirms safe data arrival by sending a positive acknowledgment (ACK) for each packet that is correctly received. The sender is responsible for detecting data loss and subsequently retransmitting the lost data. Because the nature of ACK-based protocols requires a great deal of receiver feedback, ACK-based protocols suffer from the well-known *ACK-implosion problem* [2], in which the sender becomes overwhelmed with feedback messages as the number of receivers grows large. In receiver-initiated, or *NAK-based*, protocols, each receiver sends a negative acknowledgment (NAK) to the sender only when data loss is detected at that receiver. NAK-based protocols lessen the need for receiver feedback, but may still suffer from implosion in the event of a highly correlated data loss.

Group management concerns how, and to what extent, the sender is required to maintain the state of each member in the multicast group. The extent that a protocol maintains group membership information varies widely under different reliable multicast protocols. Under a tight group management policy, for example, the sender may maintain complete state information on each member of the multicast group,

including the state of each member's receive window. In contrast, under a loose group management policy, the sender may maintain minimal information about the multicast group; for example, the sender may only maintain the number of receivers in the group. The type of reliability that the reliable multicast protocol uses often determines the type of group management that is needed. Protocols that use ACK-based reliability often require careful management of the group, while protocols that use NAK-based reliability are less reliant on the composition of the group.

The purpose of flow control in a reliable multicast protocol is to prevent the sender from causing packet loss at a given receiver by causing the receiver's buffer to overflow. Flow control for reliable multicast protocols may be rate-based or a window-based. In rate-based flow control, timers are used to limit the sender to transmitting data at or below a given rate. Data loss is used as an indication of congestion, and sending rates are adjusted accordingly. In window-based flow control, the sender is limited to sending data that falls within its current send window. As receivers confirm that data has been correctly received, the send window slides forward, allowing the sender to transmit additional data.

In this thesis, we study the issues of reliability, group management, and flow control as they apply to a reliable multicast protocol that is designed to be implemented in the operating system kernel. We focus on NAK-based reliability because it scales better than ACK-based reliability and we limit work at the sender by supporting anonymous group membership. While NAK-based protocols typically use rate-based flow control, however, due to limited buffer space, kernel-level protocols typically use window-based flow control. We explore the combination of rate-based and window-based flow control to allow for the use of NAK-based reliability in a kernel-level implementation.

Thesis Statement: *Using a combination of window-based and rate-based flow*

control, in conjunction with NAK-based recovery and anonymous group membership, it is possible to develop an efficient and scalable reliable multicast protocol that fits well into the operating system kernel.

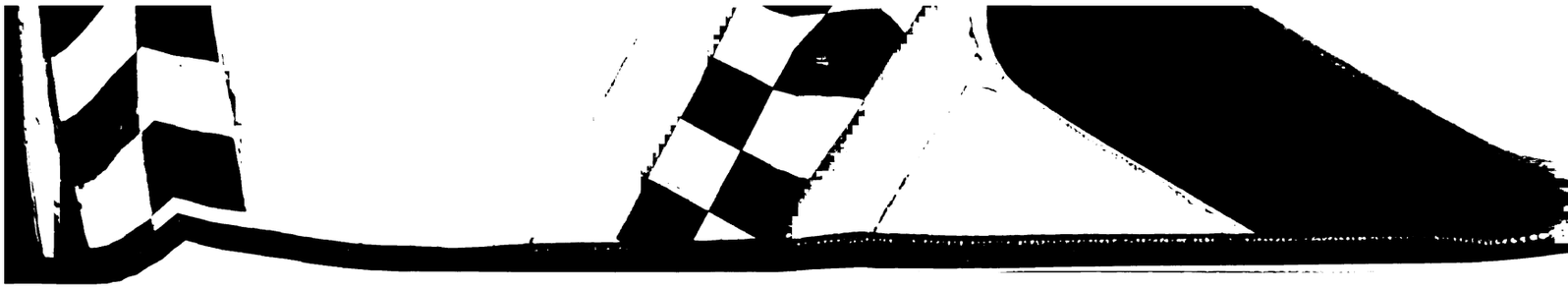
The major contributions of this work can be summarized as follows.

1. *Reliable Multicast Protocol (RMC) design.* RMC is a reliable multicast protocol that uses negative acknowledgments for reliability and a combination of window-based and rate-based flow control. While typical window-based protocols use positive acknowledgments to slide the window forward, RMC uses purely negative information, and a timer that is based on the round-trip time to the most distant receiver, to prevent the window from sliding forward too quickly. Further, RMC uses anonymous group membership to further reduce the work that is required of the sending host.

2. *RMC Implementation in the Linux kernel.* The RMC protocol has been carefully integrated in the Linux kernel, and provides an efficient and easily accessible solution to the need for reliable multicast. The implementation required us to address a number of practical problems, including buffer management, interrupt handling, and data locking.

3. *Performance evaluation through both experimentation and simulation.* We first evaluate the performance of RMC as implemented in the Linux kernel on machines in a local area network. In this environment, we are able to observe the behavior of RMC in a working environment. We further evaluate the performance of RMC in a simulated network, where we can set network parameters such as network delay and loss rates to simulate wide area networks.

The remainder of this thesis is organized as follows. In Chapter 2, we present material related to this work, including background material on multicast commu-



nication and applications that use multicast, the requirements for reliable multicast protocols, and a survey of NAK-based reliable multicast protocols. We present the design of the RMC protocol in Chapter 3, and subsequently describe the Linux implementation of the RMC protocol in Chapter 4. Chapter 5 presents performance results of the RMC protocol in two settings: the Linux kernel and a simulated network environment. Finally, in Chapter 6 we discuss conclusions and possible future directions of this work.

Chapter 2

Background and Related Work

Three basic types of communication take place on a network: unicast, broadcast, and multicast. Unicast communication is a point-to-point operation in which a single source host, the sender, transmits data to a single destination host, the receiver. In contrast, broadcast communication is a point-to-multipoint operation in which a single source host sends data to all the hosts on a network. Finally, multicast communication occurs when a single source host transmits data to a limited group of receivers.

Multicasting is implemented in different ways, depending on the network architecture. In a single-hop network, such as a local area network (LAN), data can be delivered directly to destinations. In the medium is shared, as in a traditional Ethernet network, the network interfaces of receiving nodes can be programmed to accept packets containing certain multicast addresses, as shown in Figure 2.1(a). If the medium is switched, as in an Ethernet switch, the switch can be programmed to forward multicast packets to a specified set of nodes, as shown in Figure 2.1(b).

In a multiple-hop network, multicasting is implemented by sending data over a multicast tree that is rooted at the sender and that includes all members of the

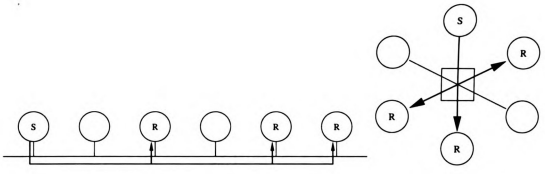


Figure 2.1: Multicasting in single-hop networks.

multicast group. The concept of a multicast tree is shown in Figure 2.2. Routers and switches receive multicast packets and forward or replicate them as necessary to reach the intended set of destinations. Whether the network is single-hop or multiple-hop, using multicast to communicate to a limited group of receivers saves a large amount of bandwidth compared to separately unicasting the data to each member of the group. Multicasting is also a better choice than broadcasting the data, which would unnecessarily interrupt processors that do not wish to participate in the multicast operation.

2.1 Applications of Multicast Communication

Applications of multicast communication are widespread in both local and wide area networks. In local area networks, networked collections of commodity workstations may be used as a platform for distributed parallel computing [3, 4]. Here, multicast supports many of the operations that are necessary to implement both distributed shared memory and message passing. In distributed shared memory, for example, multicast may be used to perform memory updates and invalidations. In message passing, multicast may be used to distribute data in parallel number algorithms such

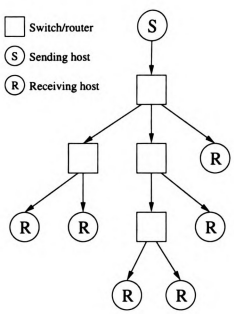


Figure 2.2: Multicasting in multiple-hop networks.

as matrix operations. Another application of multicast occurs in distributed operating systems [5]. In this case, multicast serves as an efficient method for communicating data or events among participating personal computers, workstations, and servers. Multicast is also useful in a LAN for communication between machines participating in distributed database management and parallel web servers.

There are several classes of wide area applications that benefit from multicast. Collaborative applications form one of these classes. In collaborative applications, humans use computers and networks as a tool to work together in groups or teams. For example, a group of people may use video conferencing, shared whiteboards, or interactive chat to share their ideas on a project [6]. An instructor interacting with a group of students through a distance education application is another example of a collaborative application [7, 8]. In collaborative applications, multicast is used to distribute and synchronize events and data among the participating members.

A second class of Internet applications that uses multicast is Distributed Interactive Simulation (DIS). Like collaborative applications, DIS also involves human

interaction. However, in DIS, participating players interact in a common virtual world that is created through simulations. DIS was originally developed for use in the military to simulate battlefield scenarios for training purposes. Today, DIS continues to be used to train people to function in complex environments such as airplane cockpits and space vehicles [9, 10]. DIS is also increasingly being used in interactive gaming applications. Similar to collaborative applications, in DIS, multicast is used to keep the state of the simulation synchronized at all participants on a real-time basis.

Data dissemination applications form a third class of Internet applications that benefit from multicast. Multicast communication provides this type of application with an efficient way to distribute data to a group of receivers. For example, data dissemination applications may be used to distributed live data feeds such as stock quotes or bank transactions. These applications are also commonly used to distribute bulk data such as software and database updates. Multicast communication also has great potential in the entertainment industry. Multicast may be used in the entertainment realm for computer versions of video and audio broadcasting as well as for services such as video on demand.

2.2 Multicast Communication in the Internet

Multicast is supported at two levels in the TCP/IP network hierarchy, shown in Figure 2.3. Multicast services within a single local area network are supported at the MAC, or hardware layer. At the network level protocol, which sits above the MAC layer in the network protocol stack, multicast routing protocols controls the forwarding of multicast packets through network interconnects such as a routers. Finally, the Internet Group Management Protocol (IGMP) manages multicast group membership

at the network protocol level.

2.2.1 LAN multicast

In local area Ethernet networks, Ethernet cards use the low-order bit of the high-order octet in the 48-bit hardware address to distinguish unicast addresses (0) from multicast addresses (1). By default, an Ethernet card filters only packets from the network that are addressed to either its built-in hardware address or to the broadcast address. However, multicast-capable Ethernet cards can be programmed to also accept packets that are addressed to specified multicast addresses. Thus, an Ethernet packet can be locally multicast by simply setting the Ethernet header destination field to a multicast address. Further, a host can join and leave a multicast group on the local network by requesting its network interface card to accept or reject packets for a specified multicast address [11].

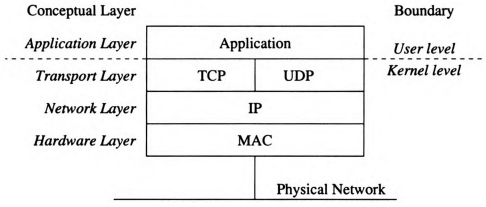


Figure 2.3: The TCP/IP network layers.

2.2.2 Multicast Routing

The Internet Protocol (IP) supports multicast services and has reserved IP addresses in the range 224.0.0.0 through 239.255.255.255 as multicast addresses. Multicast-enabled routers use multicast routing protocols to determine how to route multicast

packets from the source host to each member of the multicast group. In general, when a new multicast group is formed, multicast routing protocols construct multicast trees that connect all members of the multicast group. As multicast communication commences, routers use the trees to forward multicast packets from the source to each member of the multicast group. Depending on whether the multicast group is expected to be densely or sparsely distributed throughout a network, routing protocols may use either a *dense mode* or a *sparse mode* routing algorithm [12].

Dense mode multicast routing protocols use sender-initiated flooding techniques to propagate routing information to multicast routers. The Distance Vector Multicast Routing Protocol (DVMRP) [13], Multicast extensions to Open Shortest Path First (MOSPF) [14], and the Protocol Independent Multicast-Dense Mode(PIM-DM) [15] can be classified as dense-mode protocols. Dense mode multicast routing protocols are efficient when the multicast group is widely represented throughout a network or when senders and receivers are located close to each other in the network. However, due to the periodic flooding that these protocols use to collect routing information, dense mode routing protocols generally do not work well for situations where members of the multicast group are located sparsely throughout the network, or where bandwidth is not readily available.

Sparse mode multicast routing protocols, including Core Based Trees (CBT) [16] and Protocol Independent Multicast-Sparse Mode (PIM-SM) [17], are intended for multicast groups that are sparsely distributed over a large network. These protocols use receiver-initiated techniques through which routers explicitly join multicast trees. That is, a router only becomes involved in construction of the tree at the time a host in its subnet becomes a member of the given multicast group. In contrast to dense mode protocols, where a tree is created for each (source, multicast group) pair, sparse mode multicast routing protocols construct a single tree that is shared by all

members of the group, regardless of which host is the sender. Compared to creating a tree for each (source, multicast group) pair, using a shared tree greatly saves on the amount of state information that must be stored at each router.

It is possible that not all routers in a network will be multicast-enabled. To allow multicast communication to span parts of a network that are not connected by multicast-enabled routers, multicast tunneling may be used. In multicast tunneling, multicast packets are encapsulated in unicast packets to be transported through the portion that is not multicast-enabled. The Internet Multicast BackBone (MBone) is a virtual network within the Internet that is composed of interconnected subnetworks and routers that support IP multicast traffic. It consists of islands of network that have routing capabilities connected to other islands through multicast tunneling. The MBone has been used for the development of many distributed multimedia applications as well as for the delivery of live events [18].

2.2.3 Multicast Group Membership

Management of the multicast group membership within the IP layer is implemented by the Internet Group Management Protocol (IGMP) [19]. When a host wishes to join an IP multicast group, it issues an IGMP join request for the appropriate IP multicast address. Local multicast-capable routers that receive the join request set up the proper routing information and propagate the group membership information to other multicast-capable routers in the Internet. To maintain group membership information, multicast routers periodically poll the local network to determine the set of locally active multicast groups. If several polls for a given multicast address occur without response, the router assumes that no hosts on the local network are interested in the group, and it stops advertising its membership to other routers.

2.3 Reliable Multicast Protocols

Both hardware multicast and IP multicast provide best effort or unreliable data delivery. This means that they do not guarantee complete and correct delivery of the data. However, many of the applications that utilize multicast services require the data to be delivered reliably. For example, operations in a distributed operating system or in distributed parallel computing are assumed to be reliable. In addition, many of the applications that use multicast in the wide area networks need, or would benefit from, a reliable multicast service. Mass distribution of software, for example, would lose many of the benefits of multicasting if the software was delivered incorrectly to a number of sites and had to be redelivered anyway. Data, such as stock prices, is of little use to stock brokers if they cannot rely on the data being correct. In addition, although multimedia often must be delivered with time constraints, a certain degree of reliability will provide end users with better quality. Reliable multicasting also makes it more straightforward to implement encryption and compression on a multimedia data stream.

In light of the growing need for reliable multicast communication, a great deal of research has been devoted to this topic in the last several years. Numerous reliable multicast protocols have been designed and analyzed, and a number have been implemented. In addition to the basic transfer of data that is provided by underlying network layers, there are three main requirements for reliable multicast protocols: reliability, flow control, and connection management. Each of these is discussed in turn.

2.3.1 Reliability

In multicast communication, reliability is the property of ensuring that each receiver in the multicast group receives correct and complete data. Traditionally, reliability is achieved by retransmitting any data packets that are lost or damaged during transmission. Depending on the network loss characteristics, retransmitted data may be unicast only to those receivers that request it, or it may be multicast to the entire group. When losses are highly correlated among the receivers, it is more efficient in terms of bandwidth to multicast the retransmissions. However, when loss is uncorrelated, it is more efficient to unicast the retransmissions. In cases where both correlated and uncorrelated losses occur, protocols may choose to use an adaptive retransmission strategy. In this method, senders collect feedback information for a period of time and then either unicast or multicast the retransmission based on the degree of loss.

ACK-Based Protocols. The responsibility of ensuring that data is reliably delivered in a multicast protocol can be placed either at the sender or at each of the receivers. Sender-initiated protocols rely on the sender to ensure that all data packets are reliably delivered to all members of the multicast group. In sender-initiated protocols, each receiver in the multicast group sends positive acknowledgments (ACKs) to the sender to confirm the safe arrival of each data packet. The sender detects data loss by setting a timer while waiting for acknowledgments from all the receivers. If the timer fires before the sender receives ACKs from all of the receivers, the sender assumes that the data has been lost and retransmits the packet to the multicast group. In this thesis, we will refer to sender-initiated protocols as *ACK-based* protocols.

The Xpress Transport Protocol (XTP) [20] is one protocol that implements an ACK-based reliable multicast service. In XTP, the sender periodically requests re-

2.3.1 Reliability

In multicast communication, reliability is the property of ensuring that each receiver in the multicast group receives correct and complete data. Traditionally, reliability is achieved by retransmitting any data packets that are lost or damaged during transmission. Depending on the network loss characteristics, retransmitted data may be unicast only to those receivers that request it, or it may be multicast to the entire group. When losses are highly correlated among the receivers, it is more efficient in terms of bandwidth to multicast the retransmissions. However, when loss is uncorrelated, it is more efficient to unicast the retransmissions. In cases where both correlated and uncorrelated losses occur, protocols may choose to use an adaptive retransmission strategy. In this method, senders collect feedback information for a period of time and then either unicast or multicast the retransmission based on the degree of loss.

ACK-Based Protocols. The responsibility of ensuring that data is reliably delivered in a multicast protocol can be placed either at the sender or at each of the receivers. Sender-initiated protocols rely on the sender to ensure that all data packets are reliably delivered to all members of the multicast group. In sender-initiated protocols, each receiver in the multicast group sends positive acknowledgments (ACKs) to the sender to confirm the safe arrival of each data packet. The sender detects data loss by setting a timer while waiting for acknowledgments from all the receivers. If the timer fires before the sender receives ACKs from all of the receivers, the sender assumes that the data has been lost and retransmits the packet to the multicast group. In this thesis, we will refer to sender-initiated protocols as *ACK-based* protocols.

The Xpress Transport Protocol (XTP) [20] is one protocol that implements an ACK-based reliable multicast service. In XTP, the sender periodically requests re-

ceiver status by setting a bit in the outgoing packets. Receivers respond to this request by returning an ACK to the sender. The sender aggregates ACK information from the multicast group and retransmits data to the entire multicast group.

Single Connection Emulation (SCE) [21] is another ACK-based reliable multicast protocol. SCE is implemented by adding a network layer between the IP and TCP layers. The SCE layer uses IP multicast to transmit data to the multicast group, and receivers send ACKs back to the sender as they would in a TCP connection. At the sender, the SCE layer consolidates ACKs and other feedback information and passes it up to the TCP layer. The TCP layer, in turn, interprets the feedback as it would for a unicast connection and retransmits data accordingly.

ACK-based protocols work reasonably well when the number of receivers from which the sender must process feedback is small. However, ACK-based protocols require the sender to maintain state information for each receiver, which is expensive for large groups. More importantly, the number of ACKs grows linearly with the number of receivers. As a result, the sender can be quickly overwhelmed when the number of receivers is large. This problem is commonly known as the *ACK implosion problem* [2].

NAK-Based Protocols. In contrast to sender-initiated reliable multicast protocols, receiver-initiated protocols transfer the responsibility of reliability to each receiver in the multicast group. That is, receivers must detect missing or damaged packets in the incoming data stream, and request retransmission of the needed data. These retransmission requests may also be referred to as negative acknowledgments (NAKs). We will refer to receiver-initiated protocols as *NAK-based* protocols.

Since receivers detect data loss by gaps in the sequence numbers of incoming packets, it is possible for the last packet in a transmission to be lost without being

detected by the receivers. To address this issue, most NAK-based protocols send periodic keepalive or heartbeat packets during idle sending times. These packets typically include the sequence number of the next data octet to be sent. A receiver can detect loss at the end of the transmission be comparing the sequence number indicated by a keepalive packet to the sequence number of the next expected data at that receiver.

Researchers at Stanford University conducted some of the earlier work in NAK-based reliable multicast in their user-level implementation of the Log-Based Receiver-Reliable (LBRM) protocol [1]. LBRM makes use of a log server that logs all transmitted packets from the source. When a receiver detects a lost packet, it sends a NAK to the log server, which is then responsible for retransmitting the requested packet. The sender needs only to reliably deliver the packet to the log server (by a positive acknowledgment from the log server) and can then release the data from its memory. Like LBRM, RMC uses NAK-based reliability. Rather than using a log server, however, in RMC NAKs are sent directly to the sender and the sender is responsible for buffering sent data.

During idle periods, LBRM sends heartbeat packets at a variable rate rather than a constant one. Each sender maintains a heartbeat timer that represents the time from the last transaction to the time the next heartbeat will be sent. This timer is initialized to a minimum value after a data transmission and is doubled after each heartbeat transition until it reaches a maximum value. This method enables lost packets to be detected quickly and reduces the number of heartbeat messages compared to a fixed rate protocol. RMC also sends keepalive messages according to an exponential backoff timer during idle periods.

The Reliable Adaptive Multicast Protocol (RAMP) [22] is a NAK-based reliable multicast protocol that was designed and implemented by researchers at TASC.

RAMP is a transport layer protocol that is layered on top of IP multicast services. RAMP is intended to support collaborative-interactive applications, where the application typically is both a sender and a receiver of a single multicast group. In addition, this protocol was designed for an all-optical, circuit-switched, gigabit network developed by the ARPA-sponsored Testbed for Optical Networking (TBONE) project. In this network, packet loss mainly occurs due to buffer overflow at the receiver. As a result, packet loss is highly uncorrelated among the receivers.

In light of the given network characteristics, reliability in RAMP is provided by immediate NAKs unicast from the receivers to the sender, followed by unicast retransmissions of the requested data from the sender to the requesting receiver. Unicast retransmissions eliminate unnecessary processing of redundant packets at other receivers. Further, the use of immediate rather than delayed NAKs improves receiver performance by decreasing recovery time and, as a result, giving applications earlier access to the data so that protocol buffer space may be freed for incoming data. Like RAMP, in RMC receivers send immediate NAKs to the sender. However, in RMC loss is assumed to be correlated and, consequently, retransmissions are multicast.

Assuming that packets are successfully received more often than they are not, NAK-based protocols are more scalable than ACK-based protocols. However, NAK-based protocols may also suffer from an implosion of feedback at the sender in the case of large, highly correlated loss of a portion of the data stream. In addition, if there is a disruption in the connection between the sender and a subset of the multicast group, the sender might incorrectly interpret the lack of NAKs as an indication that data is being received successfully at all receivers. As a result, the sender may prematurely release buffered data.

Local Recovery Mechanisms. In order to reduce implosion and workload at the sender, *local recovery* may be used. In this method, rather than placing the entire responsibility of retransmission on the sender, some or all of the receivers are also allowed to retransmit requested data. Ideally, the retransmissions will come from a receiver that is close to the requesting receiver, so that loss recovery time will also be reduced. Local recovery may be used in both ACK- and NAK-based protocols. Depending on which receivers are allowed to perform the retransmissions, local recovery is referred to as either *unstructured* or *structured.*

Unstructured local recovery is used only in NAK-based protocols, and requires NAKs to be multicast to the entire receiver group. Typically, any receiver that overhears a NAK and is able to supply the missing data can retransmit the data to the multicast group. Since both NAKs and retransmissions are multicast, and only one NAK and one retransmission are necessary for each data loss, NAKs and retransmissions from multiple nodes are redundant and cause unnecessary network traffic. To address this problem, local recovery is normally used in combination with *NAK suppression.* In NAK suppression, when a receiver detects data loss, instead of immediately sending the NAK, it waits a random period of time and then multicasts the NAK to the multicast group. If a receiver "overhears" a NAK for the same data while it is waiting to send its own NAK, it cancels its NAK and allows the other NAK to report the loss. Similarly, to restrict the number of responses to a NAK, each receiver waits a random amount of time before retransmitting data, and if it hears another retransmission before doing so, it cancels the retransmission.

Unstructured local recovery with NAK suppression is used by the Scaleable Reliable Multicast (SRM) [6] protocol. Rather that multicasting NAKs and retransmissions to the entire group, SRM defines local loss neighborhoods based on the number of hops from the requesting host. When a receiver detects a loss, it multicasts the

NAK to its local neighborhood by specifying a limited number of routing hops, or time-to-live (TTL). Any receiver that hears the NAK, multicasts the retransmission with the same TTL. In this way, a receiver that overhears a NAK and cancels its own, will also overhear the retransmission. Receivers suppress both NAKs and retransmissions based on the distance from the host that originally sent the data or NAK.

Unstructured local recovery with NAK suppression reduces the feedback implosion and the workload at the sender. However, multicasting of NAKs and retransmissions not only introduces extra network traffic, it also interrupts receivers with unnecessary data. Structured local recovery, which may be used in both NAK- and ACK-based protocols, attempts to reduce unnecessary network traffic by directing feedback from a receiver to a specific destination. In structured local recovery, receivers are arranged into a logical hierarchical structure. Within this structure, internal nodes are responsible for receiving feedback from, and retransmitting data to, their children in the hierarchical structure. Feedback is unicast to a receiver's parent and retransmissions may be unicast to each requesting receiver or locally multicast to a host's children. The number of children for any given host is assumed to be relatively small, so that feedback implosion does not occur.

The Reliable Multicast Transport Protocol developed at Bell Laboratories (BL RMTP) [23] uses structured local recovery in an ACK-based protocol. In this protocol, data is first multicast to the multicast group using a best effort multicast service such as IP multicast. Each receiver is assigned a designated receiver to which is sends periodic ACKs. Designated receivers process the ACKs to determine which packets need to be retransmitted, and either unicast or multicast the requested data, depending on the number of receivers that report the loss.

The Tree-based Multicast Transport Protocol (TMTP) [24] is another protocol

that uses structured local recovery. In TMTP, participants are organized into domains, within which a domain manager is responsible for error recovery and local retransmissions. TMTP uses periodic ACKs to provide reliability between the set of domain managers and the sender, as well as between domain members and their domain manager. Because ACKs are sent only periodically, domain members may locally multicast NAKs to the domain manager for faster error recovery.

While structured local recovery reduces the number of unnecessary NAKs and retransmissions compared to unstructured local recovery, structured local recover requires extra work to be done to construct and maintain the logical structure on the receiver group. In addition, local recovery requires receivers to buffer received data and perform extra work to process NAKs and retransmissions. This may result in unfair treatment of the receivers, especially if some of the receivers are relatively slow or have limited memory. RMC uses neither unstructured nor structured recovery. Instead, recovery is centralized at the sender. Although this results in increased work at the sender, it also allows RMC to treat all receivers equally. In addition, RMC completely separates membership issues from the operation of the protocol. Thus, there is no need to adapt the protocol operation every time a host joins or leaves the group.

2.3.2 Connection Management

The second major requirement for reliable multicast protocols is connection management. Connection management includes maintaining group membership information as well as managing connection setup and tear down. Connection management is an issue for both reliable and best-effort multicast.

One issue in connection management concerns to what degree the sender must

maintain explicit knowledge of the members of the multicast group. ACK-based protocols rely on the sender to keep track of not only which hosts are members of the group, but also the state of each member. NAK-based protocols can provide reliability while maintaining minimal information of the receiver group, but some NAK-based protocols also require the sender to manage group membership in more detail for other purposes. Doing so is helpful to the sender, for example, when it is deciding whether to unicast or multicast retransmissions. Another reason that the sender may maintain more detailed information on the receiver group is to protect itself against false retransmission requests from hosts that do not have permission to be in the multicast group. Finally, some applications that operate on top of a multicast protocol may need to have access to group membership information. However, while group membership information is useful, and often necessary, for correct protocol operation, it also consumes a large amount of sender resources when the multicast group is large. Not only must the sender maintain the information in memory; it must also use valuable processor time to process and update membership status. In order to reduce the responsibilities at the sender, RMC supports anonymous group membership. Explicit group membership may be added at in the application if necessary.

A second issue in connection management is connection setup and tear down. Here, the protocol must specify how and when a receiver may join and leave a multicast group at the protocol level, as well as how and when the sender may open and close a multicast connection. Some protocols may specify that receivers can only join at the beginning of a transmission and leave at the end of a transmission, while others may allow late joins and early leaves. For example, in protocols that are designed for bulk data distributions, it doesn't make sense for receivers to join in mid-stream, or to leave the connection early. In addition, is the protocol is required to maintain a

structure on the receiver group, as is the case for TMTP [24] and BL RMTP [23], the protocol can avoid adjusting the structure by disallowing late joins and early leaves. However, for protocols that are designed for ongoing interaction, as is the case in RAMP [22] and SRM [6], it is useful to allow new users to join late or leave early. RMC can be used for both interactive applications and for bulk data distribution. Because RMC supports anonymous group membership, it can easily support late joins and early leaves. Applications may disallow late joins and early leaves if necessary.

2.3.3 Flow control

The third major requirement of reliable multicast protocols is flow control. The goal of flow control is to prevent the sender from overrunning a receiver's buffer space. Packets that are sent when a receiver's buffer is full must be dropped and, consequently, retransmitted. Thus, flow control is necessary for efficient, reliable data transfer. Flow control is especially important when the transport protocol is implemented in the kernel, where buffer space is limited.

The two most commonly used methods for flow control in reliable multicast protocols are rate-based and window-based control [25]. In rate-based flow control, timers are used to limit the sender to transmitting data at or below a given rate. Usually, the amount of data loss that is reported by the receivers is used as an indication of congestion, and sending rates are adjusted accordingly. In window-based flow control, as used by TCP, a sender is limited to sending only those packets that fall within its send window. The send window advances as the sender receives sufficient acknowledgments from receivers. In addition, in order for the protocol to make the most efficient use of the network, the size of the window is allowed to increase linearly during periods of successful transmission without loss. When transmission is unsuccessful, as

signaled by a NAK or missing ACKs, the window size is reduced to prevent further loss of packets.

For reliable multicast, the choice of flow control is influenced by whether the protocol is ACK- or NAK-based. In ACK-based protocols, the sender receives feedback from the multicast group on a regular basis so that it is fairly straightforward to use window-based flow control. Further, the sender can monitor round-trip times to estimate the amount of congestion in the network and adjust its window size appropriately. The XTP, SCE, and BL RMTP protocols all use window-based flow control. TMTP uses a combination of rate-based and window-based flow control. Windows are maintained both at the sender and at each of the receivers. Domain managers track the windows of each of the receivers in their domain to determine the amount of data that may be sent at a given time. Rate-based control is used to control the transmission rate of the data within the current window. Adding rate-based control is intended to prevent congestion in the network due to bursty traffic [24]. Like TMTP, RMC uses a combination of window-based and rate-based flow control. However, TMTP uses ACKs to relay information about the state of the windows, while RMC is a purely NAK-based protocol.

In NAK-based protocols, senders receive feedback only when negative events occur. The sender does not know the state of the receivers' buffers and cannot use this information to maintain a send window. Thus, many NAK-based protocols use rate-based flow control. In SRM, for example, a single transmission rate is specified to control transmission of both original and retransmitted packets. Usually this rate is based on the frequency of incoming NAKs.

RAMP also uses rate-based flow control. To determine the appropriate sending rate, RAMP records the number of NAKs received and the number of packets sent in defined intervals of the transmission. The delay between the transmission of individual

packets is based on the ratio of the NAKs received to the number of packets sent during a given interval. To encourage the sender to throttle the sending rate before a receiver's buffer is actually full, the receivers intentionally drop segments when their buffers become 80% full. This drop is reported to the sender in the form of a NAK, causing the sender to slow its transmission rate.

Researchers at TASC have also proposed a more proactive flow control method for the RAMP protocol [22]. In this new approach, as a receiver's buffer reaches a critical occupancy, the receiver estimates a reduced transmission rate and rate duration required for recovery. This information is sent to the sender in the form of a throttle request. A sender satisfies throttle requests by falling back to the requested transmission rate for the given period, and then resuming normal transmission. This approach is better than the previous method in that it uses explicit receiver state to set the transmission rate, rather than relying on implicit NAK counts. In addition, the throttle messages may be treated anonymously, freeing the sender from maintaining the state of each receiver in the group. RMC uses a method similar to that proposed by RAMP to control the transmission rate. Rather than calculating a reduced transmission rate and rate duration, in RMC receivers are allowed to send one rate requests to the sender every round-trip time for as long as its buffer is past the given threshold. The sender responds to each rate request by further slowing or continuing to halt the forward transmission.

In addition to controlling the amount of data that is sent, a flow control policy must also manage a sender's buffer space. Specifically, since the sender must be able to retransmit data when it is lost at the receivers, the sender must carefully manage buffers of previously transmitted data to ensure that no data is "released" before all receivers have received it correctly. In ACK-based protocols, the sender can safely release buffer space when it has received ACKs from all receivers for the given

data. However, with pure NAK-based multicast protocols, it has been shown that deadlock cannot be prevented when finite buffers are used [26]. That is, the sender has no way to know that the data has arrived at all the receivers and cannot safely release buffer space. One solution to preventing deadlock with NAK-based protocols would be to require the sender to buffer all data until the transmission is complete. This solution is feasible when the sender has unlimited buffering space. For example, the sender may keep a copy of the sent data on disk. However, if the protocol is implemented in the kernel, buffer space is limited, and another approach must be taken to prevent deadlock. One approach that many NAK-based protocols take is to require the receivers to send periodic ACKs, enabling the sender to safely release buffer space. This technique is used by both TMTP [24] and LBRM [1]. RAMP [22] takes a slightly different approach by requiring receivers to send an ACK only when the connection is closing. Finally, other implementations may require the application to buffer the data. Application buffering is appropriate for applications such as shared whiteboards, where data is maintained within the application.

Because RMC uses a purely NAK-based approach and is designed to be implemented in the kernel, there is a possibility that the protocol may release data prematurely. RMC approaches this problem by following a conservative approach to releasing data. In RMC, senders are required to buffer data for a minimum of 10 round-trip times. If data is released too early, both the sending and the receiving applications are informed of the retransmission error. At this point, the applications may take appropriate actions to retrieve or recreate the lost data.

Chapter 3

RMC Protocol Description

The RMC reliable multicast protocol is designed to be scalable and to fit well into an operating system kernel. To provide scalability, RMC limits work of the sender whenever possible. RMC shifts the responsibility of reliability to the receivers by using NAK-based reliability. Further, RMC supports anonymous group membership so that the sender is required to maintain minimal information about the receiver group. Pure NAK-based protocols typically use rate-based flow control, which requires an infinite buffer. However, kernel-level protocols require a limited buffer and typically used window-based flow control. To support a limited buffer with NAK-based reliability, RMC uses a combination of window-based and rate-based flow control.

In this chapter, we first give an overview of the RMC protocol and the RMC packet format. We then describe the strategies that RMC uses to implement reliability, connection management, and flow control.

3.1 RMC Protocol Overview

Multicast communication by means of the RMC protocol involves three key entities: the sending and receiving applications, the RMC protocol itself, and an underlying

best-effort multicast service, such as IP multicast. Figure 3.1 depicts the data flow among these entities. The application process at the sending host passes a data stream to the RMC protocol for transmission. The RMC protocol fragments this data stream into a sequence of data packets, each of which is assigned a sequence number and prefixed with an RMC header. The data packets are then transmitted using the best-effort network multicast service, which attempts to deliver a copy of each packet to each of the hosts in the multicast group. As the data packets arrive at the receiving hosts, the RMC protocol checks the packets for correctness and reassembles the individual packets into a data stream that is identical to the data stream that was sent from the source application. Finally, the RMC protocol at each of the hosts delivers the reassembled data stream to the receiving application at that host.

Reliability. The responsibility of ensuring reliability in RMC is placed at each of the receivers. To detect data loss, receivers scan the sequence numbers of incoming data packets for possible gaps. When a receiver detects a loss, it uses a negative acknowledgment to request retransmission of the lost data. RMC uses centralized recovery, which means that NAKs are sent directly to the sender, and the sender is solely responsible for retransmitting lost data.

A common problem with pure NAK-based error recovery is the possible loss of the last packet in a transmission. Because receivers detect loss by finding gaps in the sequence numbers of incoming packets, if the last packet is lost or the next packet transmission has an unbounded delay, a receiver will not detect the loss in a timely manner. To allow the receivers to quickly detect data loss that may have occurred at the end of the current data stream, the sender transmits *keepalive messages* to the receiver group during times when the sender is otherwise idle. Keepalive messages are

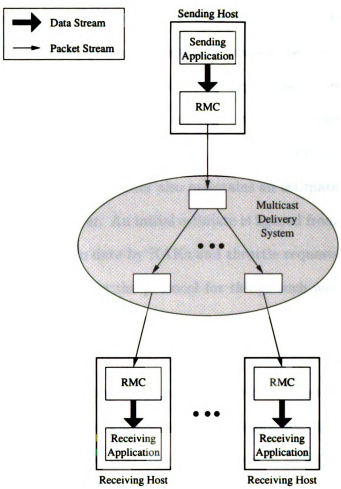


Figure 3.1: Flow of data in RMC.

sent according to an exponential backoff algorithm: the sender transmits a keepalive message immediately following a transmission, enabling the receivers to quickly detect a loss, but decreases the frequency of keepalive messages over time so as to reduce network traffic.

Because NAKs may also be lost during transmission, receivers must be prepared to retransmit NAKs. Receivers should be careful, however, to not unnecessarily retransmit NAKs before the sender has sufficient time to retransmit the data. In RMC, *local NAK suppression* is used to determine when to retransmit NAKs. In this approach, pending NAKs are retransmitted at suppression intervals, which are set to a multiple of the round-trip time from the receiver to the sender.

Connection Management. To reduce the workload at the sender, group management is very basic in the RMC protocol. Receivers send join requests when they begin to listen to the multicast connection, and leave messages when they stop listening to the connection. From these join and leave messages, the sender maintains a count of the number of receivers to signal the start and end of transmission. In addition to maintaining a receiver count, the sender also maintains an estimate of the round-trip time to the most distant receiver. An initial estimate is figured from the join requests, and the estimate is kept up to date by NAKs and throttle requests. This round-trip time estimate is used throughout the protocol for things such as determining when to release buffer space.

Flow Control. RMC uses a combination of window-based and rate-based methods for flow control. Like traditional window-based protocols, such as TCP, in RMC the sender maintains a *send window* that defines a window of bytes that may currently be sent. The send window fills as the sending application passes data to the protocol. Each data byte must remain in the window for a minimum retention time after it is transmitted. The send window may advance past each byte as its retention time expires. Similarly, each of the receivers maintains a *receive window* that defines the range of data that the receiver is currently able to receive. The receive window fills as data arrives from the network and advances as the receiving application consumes the received data. In the rate-based part of flow control, the sender transmits packets within the send window at a specified transmission rate. The transmission rate is limited by a maximum rate which is defined by the speed of the underlying network, and which is influenced by the frequency of incoming NAKs.

The responsibility of preventing buffer overflow is located primarily at each of the receivers. In order to monitor the data flow, each of the receivers divides its receive

window into three regions: the *safe* region, the *warning* region, and the *critical region*. Within the safe region, the receiver assumes there is no danger of overflow, and takes no flow control action. If the window fills into the warning region, however, the receiver sends throttle messages that request the sender to slow its sending rate. Finally, if a receiver's window fills into the critical region, the receiver sends urgent throttle messages to stop forward transmission by the sender. In response to throttle messages and NAKs, the sender goes through slow start and congestion avoidance phases, which are similar to those used by TCP. In response to urgent throttle messages, the sender stops forward transmission and sends a keepalive message, followed by a period of slow start.

3.2 RMC Packet Format

RMC segments are prefixed with a 20-byte header before they are passed to the multicast service for transmission. The RMC header, shown in Figure 3.2, includes information necessary for data delivery, as well as information used to manage the RMC connection. The reader will recognize similarities and differences with the header used by TCP [27]. The fields in the header are as follows:

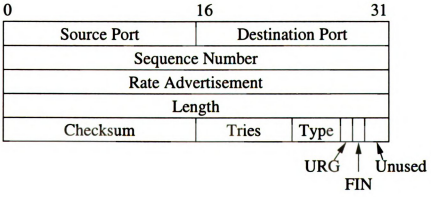


Figure 3.2: The RMC header.

Source Port. The 16-bit source port field is used to identify the sending process on the source host.

Destination Port. The 16-bit destination port field is used to identify the receiving process on the destination hosts.

Sequence Number. Like TCP, every octet of data that is sent over an RMC connection is assigned a sequence number. Sequence numbers give the sender and the receivers a common way to refer to portions of the data stream. For example, receivers use sequence numbers to reassemble incoming data packets into the original data stream and to detect missing portions of the data stream. At the sender, sequence numbers indexes buffered data so that it can be easily identified for retransmission.

The sequence number field in the RMC header is a 32-bit unsigned integer that contains the sequence number of the first octet of data in the packet. Since sequence numbers are stored as 32-bit unsigned integers, the sequence number space is finite, ranging from 0 to $2^{32} - 1$. For this reason, all arithmetic that deals with sequence numbers should be done modulo 2^{32}. The initial sequence number for a connection is a random number chosen by the sender when the connection is opened. Each octet from the data stream is labeled in sequence from this point.

Rate Advertisement. The rate advertisement is a 32-bit unsigned integer that is used for flow control purposes. In packets from the sender, this field is used to inform the receivers of the current transmission rate. Receivers use this field in feedback messages to suggest a lower sending rate from the sender.

Length. The length field is a 32-bit integer that gives the number of octets of data

contained in each data packet. This field is also used in negative acknowledgments to identify the length of the requested data retransmission.

Checksum. Similar to the TCP header checksum, the checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of data text octets, the last octet is padded on the right to form a 16-bit word for checksum purposes. The padding is not transmitted as a part of the segment. The checksum field is set to all zeroes while the checksum is being computed.

Tries. The 8-bit tries field indicates the number of times that this packet has been transmitted. The main purpose of this field is to distinguish original transmissions from retransmissions. Similar to Karn's algorithm [28], round-trip time estimates are based only on original transmissions.

Type. There are presently nine RMC packet types. These packet types and their uses are summarized in Table 3.1. DATA and KEEPALIVE packets are always multicast by the sender to the entire multicast group. Feedback messages, including NAK, JOIN, LEAVE, and CONTROL packets, are always unicast by the requesting receiver to the sender. NAKERR, JOINACCEPT, and LEAVEOK packets are unicast by the sender to the receiver that made the corresponding request.

Urgent. The urgent field is a 1-bit flag that is set to true to distinguish a throttle request as urgent.

Fin. The fin field is also a 1-bit flag that is set by the sender in DATA and KEEPALIVE packets when the sender has completed transmission for the current connection.

Type	*Use*
DATA	Used by the sender for data transmission. The transmission may be original or a retransmission.
NAK	Used by the receiver to request data retransmissions.
NAKERR	Used by the sender to inform the receiver that it is unable to satisfy a request for retransmission.
JOIN	Used by a receiver to request to join the multicast group.
JOINACCEPT	Used by the sender to confirm that a join request has been received.
LEAVE	Used by a receiver to inform the sender that it is leaving the multicast group.
LEAVEOK	Used by the sender to confirm that a leave request has been received.
CONTROL	Used by a receiver to request a reduced transmission rate.
KEEPALIVE	Used by the sender to keep the connection active during idle time.

Table 3.1: RMC packet types.

3.3 NAK-Based Reliability

RMC uses a NAK-based, centralized recovery strategy to implement reliability. As each receiver reassembles the data stream, it detects missing segments by monitoring the sequence numbers of incoming data packets. When a gap occurs in the sequence numbers of two incoming packets, the receiver knows that one or more data packets has been lost or delivered out of order. The receiver immediately sends a NAK for the missing data directly to the sender, provided that a NAK for this range of data has not previously been sent. The sender buffers previously sent data and responds to NAKs by retransmitting the requested data.

Out-of-order data that is received following a data loss cannot be incorporated into the data stream until the lost data is recovered. This out-of-order data does not cause redundant NAKs to be generated, but is stored in an out-of-order queue for integration after the missing data arrives.

An example of RMC's basic recovery process is shown in Figure 3.3. In the figure,

the receiver successfully receives packets 1 and 2. Packet 3 is lost during transmission, and packet 4 is received successfully. On the arrival of packet 4, the receiver detects a gap in the incoming data stream at packet 3, and immediately sends a NAK for packet 3. While the receiver is waiting for packet 3, packets 5 and 6 arrive and are buffered with packet 4 in an out-of-order queue. When packet 3 arrives, packets 3, 4, 5, and 6 are added to the end of the data stream and made available to the application.

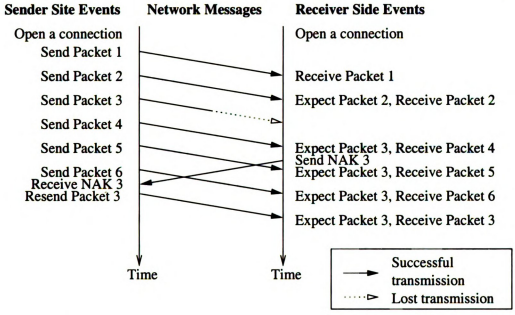


Figure 3.3: NAK-based reliability with centralized recovery.

RMC uses centralized recovery for data retransmissions. This means that all NAKs are sent directly to the sender and the sender is solely responsible for retransmitting data to the receivers. Although centralized recovery places the burden of retransmission totally on the sender, it is less complicated than local recovery. First, centralized recovery does not require any of the receivers to perform extra work or to buffer the received data, as does local recovery. Second, centralized recovery does not require a structure to be imposed and maintained on the receiver group. This lack of structure requires less work by the sender than does local recovery, especially when the receiver group membership changes frequently.

As discussed earlier, an important issue in multicast reliability is whether retransmissions are unicast to each requesting receiver or multicast to the entire group. In the current version of RMC, all data retransmissions are multicast. Depending on the loss characteristics of the underlying network and the size of the multicast group, however, it may be beneficial to adopt adaptive retransmission. Under adaptive retransmission, after receiving a NAK for a given segment of data, rather than immediately resending the requested data, the sender would collect NAKs for the same data for a given amount of time. After that period of time, if the number of NAKs exceeded some threshold, the sender would multicast the retransmission. Otherwise, the sender would unicast the retransmission to each of the requesting receivers.

Another issue in NAK-based protocols is the possibility of last packet loss. Because receivers detect loss by finding gaps in the sequence numbers of incoming packets, if the last packet is lost, a receiver will never detect the loss. To allow the receivers to quickly detect data loss that may have occurred at the end of the data stream, RMC sends keepalive messages to the receiver group during times when the sender is idle. In keepalive messages, the sequence number field of the header is set to equal the next unused sequence number at the sender. Receivers handle keepalive messages by checking the sequence number from the keepalive message against the next expected sequence number. If there is a gap, the receiver knows that data has been lost, and sends a NAK for the lost data.

In RMC, keepalive messages are sent according to an exponential backoff timer, as in LBRM [1]. An example of keepalive with exponential backoff is shown in Figure 3.4. The sender maintains a keepalive interval k, which specifies the time between when the previous packet was sent and the time the next keepalive message should be sent. Each time a data packet is sent, k is reset to a minimum value k_{min}, which is defined as one round-trip time. Each time a keepalive message is sent, the value

of k is doubled, up to a maximum value k_{max}, which is currently defined as five round-trip times. The idea of using exponential backoff is to cluster the keepalive messages following a data transmission, enabling receivers to quickly detect isolated packet losses. On the other hand, keepalive messages are sent less frequently during long idle periods, so as to reduce network traffic.

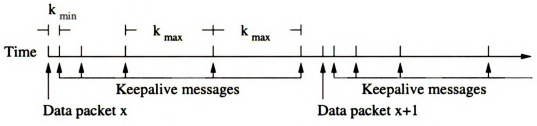


Figure 3.4: Keepalive messages sent with exponential backoff [1].

Because the underlying data delivery service is unreliable, receivers must be prepared to resend NAKs in the event that they are lost. Ideally, only one NAK should arrive at the sender for each segment that is lost at each receiver. Thus, receivers must be careful to not resend a NAK before the original NAK has had ample time to be received and processed by the sender and the retransmitted data has had time to traverse the network.

RMC uses *local NAK suppression* to control NAK retransmissions. In local NAK suppression, after a NAK is sent, it is placed on a list of pending NAKs. Each pending NAK is then resent at suppression intervals, until the requested data is delivered. The suppression interval defines an ample amount of time for a NAK to be processed and the data to be resent under normal circumstances. In the current version of RMC, the suppression interval is defined as four round-trip times to the sender.

3.4 RMC Connection Management

In order to keep the workload at the sender small, RMC requires very little of the sender in terms of group membership and connection management. RMC supports anonymous group membership and allows receivers to join after transmission has started and to leave before the transmission completes. A late joiner may not request retransmission of data that was sent before the first data packet that it received after joining the group.

Receivers are required to send JOIN messages when joining a multicast connection and LEAVE messages when leaving the connection. The sender uses these requests to maintain the number of receivers in the group, but maintains no information about individual receivers. This type of membership frees the sender from maintaining and ensuring correctness of the state for each receiver. If needed, explicit membership may be maintained at a higher level by the user application.

3.4.1 Port numbers and connections

Like UDP and TCP, RMC uses port numbers to identify the sending and receiving processes on the participating machines. Sending and receiving processes must bind to a local port before any data communication is allowed to take place.

Conceptually, an RMC connection is defined by a pair of endpoints: one for the sender and one for the multicast group. An endpoint is identified by a (address, port) pair, where the address is the identification that the lower network layer assigns the machine or the multicast group and the port is an RMC port on that host. The sender sends data to the multicast receiver endpoint, and receivers send feedback to the sender endpoint. The multicast receiver endpoint may be further defined by the set of hosts that have joined the multicast group. This means that all receivers must

bind to a common port.

3.4.2 Establishing a connection

Because RMC supports anonymous group membership, establishing a connection at the sending host is quite simple. The only requirements are that the sender binds to a local port and establishes a destination endpoint consisting of the multicast address and the port number.

Receivers are required to perform two tasks in order to establish a connection. First, the receiver must inform the supporting network layer that it wishes to join the multicast group. Second, the receiver must send a JOIN message to the sender in response to the first data packet that is received from the data stream. Although the sender keeps no specific information on each of the receivers, the join request serves two purposes. The primary use of the join requests is so that the sender can keep an estimate of the number of receivers that are listening to the transmission. In addition, the sender uses the join request to calculate a round-trip time for the new receiver, which is used to update the round-trip time value used by the sender to manage data release and keepalive timers.

To confirm that the join request was received, the sender unicasts a JOINACCEPT packet back to the joining receiver. The receiver uses this response to estimate the round-trip time to the sender.

3.4.3 Closing a connection

In order to close a connection, a receiver is again required to complete two tasks. First, the receiver informs the supporting network layer that it wishes to leave the multicast group. This allows the supporting service to stop delivering multicast packets for

that multicast group to the receiving host. Second, a receiver informs the sender that it is leaving the RMC connection by sending a LEAVE message to the sender. This enables the sender to adjust its count of the listening receivers. The sender responds to LEAVE messages by unicasting a LEAVEOK packet back to the requesting receiver. At this point the receiver is free to close its connection.

Closing a connection on the sender endpoint is more complicated because the sender must be sure that all sent data has been reliably delivered to all of the receivers before it closes the connection. First, the sender must ensure that any data that was previously queued to be sent is pushed on to the network. Next, it must give receivers ample time to request retransmissions for any lost data. The algorithm for determining this time is based on the algorithm that determines when data packets may be released from the send buffer. That is, the connection may be closed after the last data packet has been released from the send buffer.

3.4.4 Round-trip time estimation

Both the sender and the receivers make use of round-trip time information to control certain aspects of the protocol behavior. The receivers use the round-trip time information mainly for suppressing feedback messages. Receivers obtain an initial estimate of the round-trip time by sending a join request when they receive the first data packet. In addition, each receiver updates the round-trip time each time a data response to a NAK is received. Although the receiver can not be sure that its NAK resulted in the retransmission, the resulting estimate is good enough for the intended purposes.

The sender uses an estimation of the round-trip time to set the congestion window timer and to determine how long to buffer data. The sender calculates an estimate

of the round-trip time from join requests. The estimate is kept up to date by NAKs and throttle requests.

3.5 RMC Flow Control

As previously discussed, RMC uses a combination of window-based and rate-based flow control. The window-based portion of the flow control manages the buffer space. The rate-based portion of the flow control, on the other hand, actually sets a limit on how quickly data within the send window may be transmitted.

3.5.1 Window-based control

As a part of the window-based control, the sender and each of the receivers enforce a set of rules on the local sequence number space that defines a window of data that may be sent and received, respectively. The following paragraphs discuss in detail the rules that the sender and the receivers use to define the sequence space, as well as the terminology used to refer to the sequence space.

Send Sequence Space. The send sequence space is shown in Figure 3.5. The send window, shown in gray in the figure, includes all data that is currently buffered or that can be currently buffered by the protocol. The send window begins at *snd_wnd* and has a size of *snd_wnd_size*. The *snd_nxt* index marks the boundary in the sequence space between data that has been sent and data that is yet to be sent. Because sent data must be within the buffer, *snd_nxt* will always fall at or within the boundaries of the send window.

As shown in Figure 3.5, *snd_wnd*, *snd_nxt*, and *snd_wnd* + *snd_wnd_size* divide the send sequence space into four regions. Region S1, which falls to the left of the

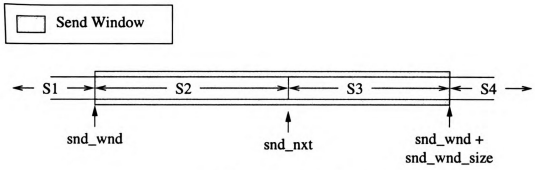


Figure 3.5: The send window.

send window, refers to the part of the data stream that has already been sent and subsequently released from the buffer. Data that has been sent and that is being buffered for possible retransmissions falls within region S2. Data in region S3 has not yet been sent, but will fit in the send window. Data in region S4 does not currently fit into the send window.

Unlike positive acknowledgment protocols, where the receiver explicitly informs the sender that data has been successfully received and can be released from its buffer, in NAK-based protocols, the receivers inform the sender only when data has not been received. Thus, without using an infinite buffer, it is impossible to provide 100% reliable delivery of data to the entire receiver group. RMC handles possible data loss by informing both the sending and the receiving application.

In RMC, buffer release is based on when a packet was most recently sent and an estimate of the round-trip time to the most distant receiver. In the current version of RMC, the minimum time that any data packet must be buffered is ten round-trip times from the most recent time the packet was sent, although this parameter is configurable.

Receive Sequence Space. The receive sequence space is depicted in Figure 3.6. Like the sender, receivers also define a window on the sequence space. The receive

window identifies the portion of the data stream that has been received and is being buffered for the application, as well as data yet to be received for which there exists available buffer space. The *rcv_wnd* index marks the front of the receive window and is the sequence number of the first octet that is being buffered and ready to be delivered to the receiving application. The receive window has size *rcv_wnd_size*. Within the receive window, the *rcv_nxt* index marks the boundary between data that has been received and data that is yet to be received. Because packets that do no fit in the buffer are dropped, *rcv_nxt* will always fall at or within the boundaries of the receive window.

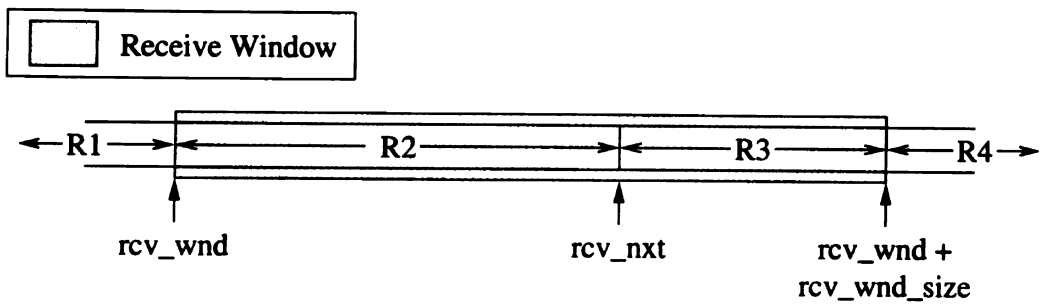


Figure 3.6: The receive window.

Similar to the sending side, the receive sequence space is also divided into regions, as shown in Figure 3.6. Region R1 refers to that part of the data stream that has been received and consequently consumed by the receiving application. Region R2 refers to data that has been received and is being buffered, but which has not yet been read by the application. Data that can be immediately received and buffered falls in region R3. Finally, R4 is the part of the data stream to be received in the future, but which does not fit into the current receive window, and cannot be currently buffered.

3.5.2 Rate-based control

In the rate-based portion of RMC's flow control, the sender maintains a current transmission rate that defines how quickly the sender may transmit data from within the send window. The value of the current transmission rate is advertised in every outgoing packet, and is limited by the speed of the underlying network.

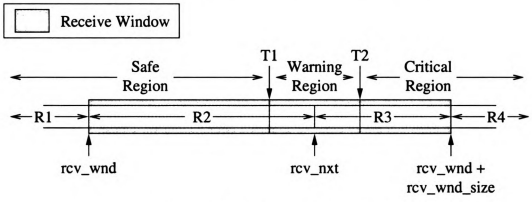


Figure 3.7: The receive window.

The responsibility of ensuring that the receivers' buffer spaces do not overflow is placed at each of the receivers. Receivers monitor data flow by dividing the receive window into three flow regions, and then taking different flow control actions depending on how full the receive window becomes. These flow regions are defined by two thresholds, $T1$ and $T2$, as shown in Figure 3.7. $T1$ marks the beginning of the warning region, and is set by the protocol to the point in the window where there is room to receive four round-trip times worth of data at the maximum transmission rate. Similarly, $T2$ marks the beginning of the critical region and is set to the point where there is room to receive two round-trip times worth of data. The area to the left of $T1$ is considered a safe region.

Receivers use throttle requests to request the sender to reduce its sending rate when the receiver has determined there is a possibility of overflow in the local receive window. Each receiver is locally suppressed to sending one throttle reduce request

per round-trip time to the sender.

On the receipt of each packet, a receiver uses the advertised rate to evaluate the state of its receive window. The receiver uses the following rules to determine when to send a throttle request.

1. If the receive window is filled only into the safe region, no flow control action is taken.

2. If the receive window is filled into the warning region, the receiver sends a throttle request only if the amount of data that may be sent at the advertised rate for the next four round-trip times is greater than the empty portion of the receive window.

3. If the receive window is filled into the critical region, the buffer is considered dangerously full. The receiver sends a throttle request to stop forward transmission regardless of the advertised rate.

If the throttle requests fail to arrive at the sender in time to sufficiently slow or stop forward transmission and a receiver's window becomes full, the receiver must drop those packets that do not fit in the window. As space in the receive window opens up, the receiver sends NAKs to request retransmission of those packets that it previously dropped.

3.5.3 Slow start and congestion avoidance

At the beginning of data transmission for a new connection, and any time following a NAK or a throttle request, the sender sets the transmission rate to a minimum value, which is approximately one-tenth of the maximum transmission rate. In this way, the sender can test the transmission rate under the current network conditions without

quickly overwhelming the network or the receivers. However, because the sender only receives notification from receivers of negative events, it has no indication of when or by how much it may increase the transmission rate to make full use of the potential of the network. For this reason, the sender uses slow start and congestion avoidance phases to increase the transmission rate following a rate decrease. RMC's slow start and congestion avoidance phases are based on the slow start and congestion avoidance methods used by TCP [29], but are applied to the transmission rate, rather than the size of the window.

In the slow start phase, the transmission rate is allowed to increase on an exponential scale. That is, during slow start, the transmission rate starts at the minimum rate and is allowed to double every round-trip time. In contrast, during the congestion avoidance phase, the transmission rate is only allowed to increase linearly.

Following a rate decrease, the sender starts in the slow start phase and uses a rate threshold to determine when to move to the more conservative congestion avoidance phase. The rate threshold is set at what is considered to be a "safe" rate at the time the rate decrease occurs. If a NAK initiates the rate decrease, the rate threshold is set at one half of the current transmission rate. If, however, a throttle request initiates the rate decrease, the rate threshold is set at one half of the transmission rate that is requested by the receiver in the throttle request. If the sender receives another NAK or throttle request during its slow start or congestion avoidance phases, it sets a new rate threshold and begins slow start again.

Chapter 4

RMC Implementation in Linux

For this thesis project, the RMC protocol was implemented as a network driver in the Linux operating system kernel. We chose to implement the RMC protocol at the kernel-level rather than at user-level for performance purposes. In most cases, kernel-level network drivers perform better than user-level drivers because kernel-level drivers have closer access to the network and run at higher priority than user-level code. In addition, when the protocol is located in the kernel, much of the protocol processing can take place without having to frequently cross the user/kernel boundary. The Linux kernel was chosen as the medium for the protocol implementation because the Linux kernel source code is publicly available. Further, the Linux operating system is widely used and is fairly well tested.

This chapter provides a description of the implementation of the RMC protocol in the network stack of the Linux kernel. First, we present background material on the implementation of the Linux networking stack, focusing on the TCP/IP network stack. We then describe where and how RMC fits into the Linux network stack. Finally, we discuss details of the RMC protocol implementation, including the implementation of socket creation, the architecture of the receiver and sender, and the

implementation of socket close.

4.1 Linux Networking

Linux is a freely available UNIX-based operating system that began as a project of Linus Torvalds, a Finnish student of computer science. Linux is distributed under the GNU Public License, which means that the full source code is available to users. The Linux source code was first made available on the Internet in 1991, and continues to evolve due to the support of developers located worldwide [30].

Networking support is an integral and important part of the Linux kernel. Networking is important to Linux not only because of the growing need for networked computers, but also because the success of the Linux operating system is due largely to the collaborative environment that the Internet provides [31].

Linux supports several types of networking sockets, also known as *address families.* The most widely used address family is the INET address family, which implements the TCP/IP network protocol. In addition to the INET address family, Linux also supports the UNIX, AX25, IPX, AppleTalk, and X25 socket address families.

Linux networking is implemented as a series of connected layers as shown in Figure 4.1. User-level applications access networking capabilities through the BSD socket interface. In the INET address family, BSD sockets are supported by the INET socket layer, which manages communication endpoints for IP-based transport protocols. INET sockets normally interface with either the TCP or UDP transport layer protocols, but may also access the IP layer directly, given that the application is being run by a super user. The IP layer implements the Internet Protocol and is supported directly by the network devices, including Ethernet, PPP, and SLIP [31]. Within the Linux networking layers, RMC is implemented as a transport layer protocol and is

located between the BSD socket layer and the IP layer.

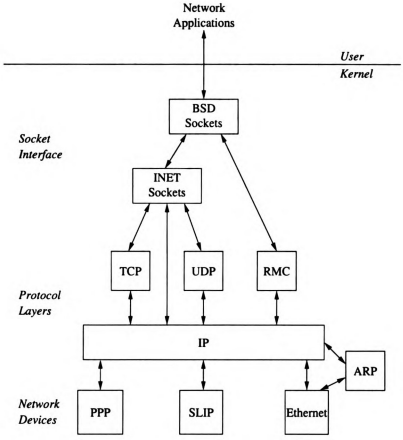


Figure 4.1: Linux networking layers.

4.1.1 BSD Sockets

BSD sockets provide a unified interface between user-level applications and kernel-level networking services from the supported address families. The BSD socket interface, shown in Figure 4.2, consists of a set of system calls that user-level applications use to create network sockets, initialize communication endpoints, and communicate with other applications on remote machines. RMC supports a subset of the BSD system socket calls and provides a BSD socket interface to user-level applications.

Often, two communicating applications take on a client-server relationship, where the server continuously satisfies requests by the client. A typical sequence of calls

```
int socket(int addr_family, int type, int protocol);
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
int connect(int sockfd, struct sockaddr *dest_addr, int addrlen);
int send(int sockfd, const void *msg, int len, unsigned int flags);
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
   const struct sockaddr *to, int tolen);
ssize_t write(int sockfd, const void *buf, size_t count);
int recv(int sockfd, void *buf, int len, unsigned int flags);
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
   struct sockaddr *from, int *fromlen);
ssize_t read(int sockfd, void *buf, size_t count);
int setsockopt(int sockfd, int level, int optname, const void *optval,
   int optlen);
int close(int sockfd);
```

Figure 4.2: BSD socket system calls.

made by the client and server to carry out such a data transfer is shown in Figure 4.3. The server runs in a continuous loop, accepting connections, receiving and processing requests from that connection, and closing the connection [32].

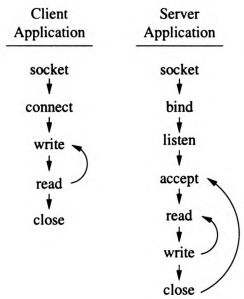


Figure 4.3: Sequence of system calls for a simple client and server.

BSD Socket Structure. For each socket that is created, BSD allocates and initializes a `socket` structure to maintain high-level information about the socket. The `type` field and the `ops` field in the `socket` structure are used to distinguish which address family the socket belongs to. The `type` field holds the identifier for the connection type. For example, the SOCK_STREAM type corresponds to a TCP socket, while the SOCK_DGRAM type corresponds to a UDP socket. The `ops` field is a pointer to a vector of address family protocol operations.

BSD Socket Interface. In the Linux kernel, the initial interface between BSD sockets and each lower layer address family is defined by a `net_proto_family` structure, which includes a unique address family identifier and a pointer to a `create` function. During initialization of the networking services during system startup, each address family registers a `net_proto_family` vector with the BSD layer. The BSD layer stores the `net_proto_family` vectors in the `net_families` table. An example of the `net_families` table after system initialization is shown in Figure 4.4.

net_families

AF_INET	inet_create()
AF_UNIX	unix_create()
AF_AX25	ax25_create()
AF_IPX	ipx_create()
...	...

Figure 4.4: The net_families table.

Each address family also initializes a protocol operations vector that defines additional functions specific to the address family. The protocol operations vector is defined by the `proto_ops` structure, which is shown in Figure 4.5. The `family` field

indicates which address family the structure represents and is initialized to the same unique address family identifier that is included in the **net_proto_family** structure. The remaining fields are function pointers to other address family specific functions. For example, in the **AF_INET** address family's protocol operations vector, the **sendmsg** and **recvmsg** fields are initialized to point to the **inet_sendmsg** and **inet_recvmsg** functions.

```
struct proto_ops {
  int   family,             /* Unique address family identifier.     */
  int   (*dup)(...);        /* Duplicate a socket.                   */
  int   (*release)(...);    /* Release a socket structure.           */
  int   (*bind)(...);       /* Bind a socket to a local port.        */
  int   (*connect)(...);    /* Connect a socket to a remote addr.    */
  int   (*accept)(...);     /* Accept connections from remote hosts. */
  int   (*listen)(...);     /* Listen for connections.               */
  int   (*shutdown)(...);   /* Close a socket connection.            */
  int   (*setsockopt)(...); /* Set option specific to addr. family.  */
  int   (*getsockopt)(...); /* Get option specific to addr. family.  */
  int   (*sendmsg)(...);    /* Send a message on the socket.         */
  int   (*recvmsg)(...);    /* Receive a message on the socket.      */
  ...
};
```

Figure 4.5: The proto_ops structure used by BSD as the interface to different address families.

After the system is up and running, user applications use the **socket** system call to create new network sockets, specifying the address family identifier as a parameter. In the implementation of the **socket** system call, BSD uses the address family identifier as an index into the **net_families** table and calls the corresponding **create** function. Each address family's **create** function is responsible for completing any address family specific initialization for the new socket, including setting the BSD socket **type** field to the connection type and the BSD socket **ops** field to point to the address family's **proto_ops** vector. The resulting data structures are shown in

Figure 4.6. Later, as the network sockets are used, BSD references the protocol operations vector in the given BSD **socket** structure to complete address family specific operations for that socket.

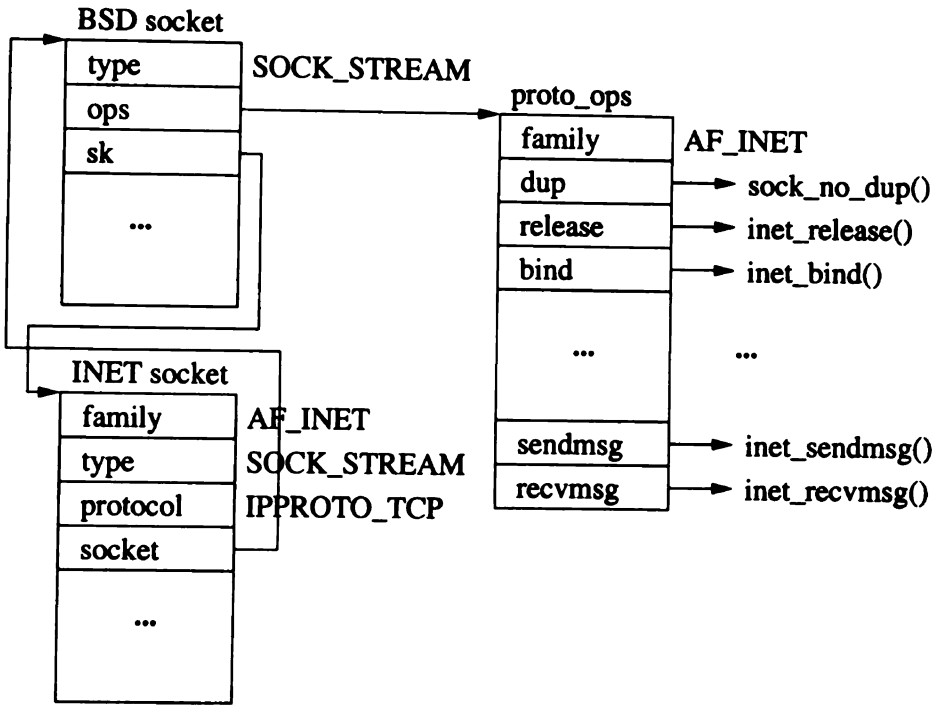


Figure 4.6: Linux BSD and INET socket structures.

4.1.2 INET Sockets

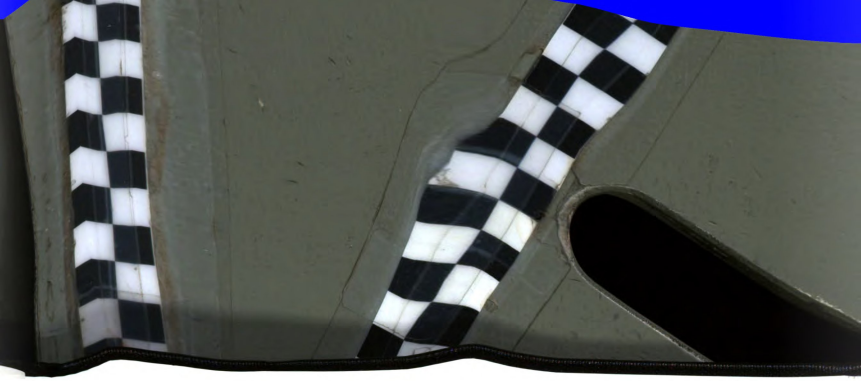
The INET socket layer is located one layer below BSD sockets in the Linux networking stack. The purpose of the INET socket layer is to provide a common interface between BSD sockets and the TCP, UDP, and IP protocols, as well as to provide common supporting functions for those protocols that are built on top of the IP layer. RMC bypasses the INET socket layer, but uses many of the data structures and functions that the INET socket layer provides, including the INET socket data structure, socket buffers and the functions used to manage socket buffers, and the socket locking and socket memory management functions.

The INET Socket Structure. The INET socket layer uses its own socket data structure, the **sock** structure, to maintain network information that is specific to those protocols that are based on the IP layer. A **sock** structure is allocated and initialized for each INET socket as part of the **inet_create** function. Because RMC is based on the IP layer, it also allocates an INET socket to maintain information about each RMC socket.

A portion of the **sock** structure is shown in Figure 4.7. Among other things, the **sock** structure includes IP address and transport layer port information for the local and remote communication endpoints. The **sock** structure also includes a series of variables that track memory allocation for the socket and a series of variables that define the data queues for incoming and outgoing packets, as well as a place to add in a transport protocol control block for transport layer specific information. Other fields in the **sock** structure maintain information about the local state of the socket and miscellaneous information about the state of the connection.

Socket buffers. Linux uses socket buffers, defined by the **sk_buff** structure, to manage individual communication packets and to pass them between the different socket layers and the device drivers. RMC receives packets from and passes packets to the IP layer in the form of socket buffers. The **sk_buff** structure, depicted in Figure 4.8, is composed of a data block with an attached control block. The data block holds the data that will be sent to, or that was received from, the network. The control block consists of length and pointer fields that allow each protocol layer to manipulate the application data using standard methods.

Normally, when a process allocates an **sk_buff** for an outgoing packet, it allocates enough space for both the data and all of the network headers. It copies the data into the data block, reserving enough space in front of the data for the lower network



```
struct sock {
...
/* Socket addressing information. */
   __u32            daddr;          /* Foreign IPv4 addr            */
   __u32            rcv_saddr;      /* Bound local IPv4 addr        */
   __u32            saddr;          /* Sending source               */
   __u16            dport;          /* Destination port             */
   unsigned short   num;            /* Local port                   */
   __u16            sport;          /* Source port                  */
...
/* Memory allocation information. */
   int              rcvbuf;         /* Size of receive buffer in bytes */
   atomic_t         rmem_alloc;     /* Receive queue bytes committed   */
   int              sndbuf;         /* Size of send buffer in bytes    */
   atomic_t         wmem_alloc;     /* Transmit queue bytes committed  */
   unsigned int     allocation;     /* Allocation mode                 */

/* Queues for incoming and outgoing packets. */
   struct sk_buff_head   receive_queue;  /* Incoming packets       */
   struct sk_buff_head   write_queue;    /* Packet sending queue   */
   struct sk_buff_head   back_log;       /* Backlog packet queue   */
...
/* Transport protocol specific information. */
   union {
      struct tcp_opt     af_tcp;
   } tp_pinfo;

   unsigned short        mss;            /* current eff. mss       */
   volatile unsigned char state;         /* Connection state       */
   unsigned short        shutdown;       /* Ready to shut down.    */
   struct wait_queue     **sleep;        /* Sock wait queue        */
   int                   sock_readers;   /* User count             */
   struct proto          *prot;          /* Transport layer func.  */
   unsigned char         protocol;       /* Transport layer id     */
   struct socket         *socket;        /* BSD socket             */
...
/* Callbacks */
   void            (*state_change)(struct sock *sk);
   void            (*data_ready)(struct sock *sk,int bytes);
   int             (*backlog_rcv)(struct sock *sk, struct sk_buff *skb);
...
};
```

Figure 4.7: The sock data structure.

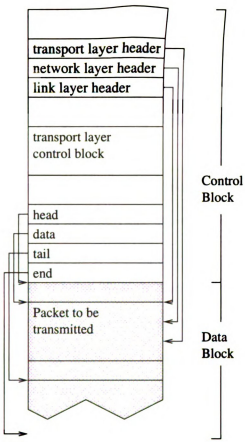


Figure 4.8: A socket buffer.

layers to add in their network headers. By initially reserving space for headers in the data block, each of the network layers can easily insert their headers into the data block and avoid costly copy operations.

The control block of an `sk_buff` also includes pointers to the network headers in the data packet. These are useful, for example, as an incoming data packet is being passed up the network stack. As each layer processes the packet, it sets the appropriate header pointer to point to the beginning of its header. It then sets the data pointer to point just past its network header. Thus, when the next level receives the packet, it can easily find the beginning of its header. In addition, if the current network level needs to access lower layer headers, it can easily locate the network headers using the network header pointers that were set by the lower layers.

Like the INET socket structure, socket buffers also have a transport layer control block that transport layer protocols may use to maintain transport layer specific information. For example, TCP may use the transport layer control block to maintain information such as the beginning and ending sequence numbers of the data packet.

Socket Locking. Incoming data packets are delivered by interrupt and bottom half handlers, which are allowed to interrupt other processes. The processing that occurs as a result of an incoming packet has the potential of changing the socket state, possibly disrupting work that is being done in the interrupted process. In addition, it is common for network protocols to use timers to schedule future work. Often, the scheduled work involves updating the socket state. Thus, a timer interrupt could also have an unpredictable effect on the socket.

Packets are delivered to RMC as a part of the network bottom half handler. In addition, RMC uses timers to schedule various events. To prevent sockets from being put in an inconsistent state, the RMC protocol must be careful to lock sockets while they are being accessed or modified.

Socket locking is implemented by maintaining a socket reader count for each socket. To check the lock status of a socket, interrupt handlers simply check the value of the socket reader count. If it is zero, the socket is considered unlocked and the handler continues. If, however, the count is one or more, the socket is locked, and the handler's processing is delayed.

The socket reader count is maintained in the `sock_readers` field of the INET socket structure. The `lock_sock` and `release_sock` functions, shown in Figure 4.9, are used to lock and unlock a socket. To lock a socket, the `lock_sock` function increments the socket reader count. Likewise, the `release_sock` function decrements the socket reader count to unlock a socket. In addition, if a socket becomes unlocked

as a result of the release, packets on the backlog queue are processed and delivered to the socket via a backlog receive function.

```
static inline void lock_sock(struct sock *sk)
{
   sk->sock_readers++;
   barrier();
}

static inline void release_sock(struct sock *sk)
{
   barrier();
   if ((sk->sock_readers = sk->sock_readers-1) == 0)
      __release_sock(sk);
}

void __release_sock(struct sock *sk)
{
   if (!sk->prot || !sk->backlog_rcv)
      return;

   /* See if we have any packets built up. */
   start_bh_atomic();
   while (!skb_queue_empty(&sk->back_log)) {
      struct sk_buff * skb = sk->back_log.next;
      __skb_unlink(skb, &sk->back_log);
      sk->backlog_rcv(sk, skb);
   }
   end_bh_atomic();
}
```

Figure 4.9: The lock_sock and release_sock functions.

Socket Memory Management. Memory is a valuable and limited resource at the kernel-level of an operating system, and must be carefully managed. RMC manages socket memory in the same way that the INET socket layer does. In the INET layer, each INET socket that is created is given a limited amount of memory that it may use for processing the incoming and outgoing data stream. The maximum amount of memory that may be allocated for writing and reading is set in the **sndbuf** and

rcvbuf fields in the INET socket structure during socket creation. For example, the default value for each TCP socket is 64 KB.

The INET functions **sock_wmalloc** and **sock_rmalloc** handle the allocation of memory for writing and reading, respectively. These functions keep track of the amount of memory that has been allocated for each socket by incrementing the **wmem_alloc** and **rmem_alloc** fields in the INET socket structure. Should the requested amount of memory cause a socket to exceed its allocation limit, the functions deny the request and the socket must wait until memory that it previously allocated is freed. The **sock_wfree** and **sock_rfree** INET functions handle freeing memory to the socket for the purposes of writing and reading. In addition to freeing the memory, they decrement the **wmem_alloc** and **rmem_alloc** fields by the amount of memory that was freed, and awaken any processes that are waiting for memory for the given socket.

4.1.3 The IP Interface

RMC is built on top of the IP layer. The interface to the IP layer may be defined in terms of receiving and sending IP packets. In this section we discuss the interface that IP uses to communicate with other layers of the network stack.

Receiving IP Packets. The interface between transport layer protocols and the supporting IP layer in the receiving direction is defined by the **inet_protocol** structure, which is shown in Figure 4.10. Among other fields, the **inet_protocol** structure includes the **handler** field, which is a pointer to the transport layer's receiver interrupt handler for processing incoming data packets, and the **err_handler** field, which is a pointer to the transport layer's error interrupt handler for handling ICMP packets. The **inet_protocol** structure also includes a **protocol** field, which is used for

protocol identification.

During system initialization, each transport protocol registers an **inet_protocol** vector with IP. Later, as data or ICMP packets arrive from the network, IP uses the protocol identification field in the IP header as an index into the registered **inet_protocol** vectors. IP then delivers packets to the appropriate transport layer protocol by calling the corresponding handler.

```
struct inet_protocol
{
   int (*handler)(struct sk_buff *skb, unsigned short len);
   void (*err_handler)(struct sk_buff *skb, unsigned char *dp, int len);
   struct inet_protocol *next;
   unsigned char protocol;
   unsigned char copy:1;
   void *data;
   const char *name;
};
```

Figure 4.10: The inet_protocol structure.

Sending IP Packets. The interface between IP and transport layer protocols in the outgoing direction is defined by two main sending functions: **ip_queue_xmit** and **ip_build_and_send_pkt**. The function definitions for these functions are shown in Figure 4.11. Both functions take a socket buffer as input, prefix the packet stored in the socket buffer with an IP header, and queue the packet to be sent out by the lower level device driver. The **ip_queue_xmit** determines to which socket the socket buffer belongs and uses the bound destination address from the socket to route the outgoing packet. The **ip_queue_xmit** function is commonly used by the TCP protocol since sockets are bound to a single destination. The **ip_build_and_send_pkt** function, on the other hand, allows the calling entity to specify the destination address independent of the bound destination. However, it also requires the calling entity to have already

calculated the routing information for the desired destination and have stored it in the control block of the socket buffer. To calculate the routing information, IP provides another function, `ip_route_output`, which is normally called immediately before `ip_build_and_send_pkt`. The `ip_build_and_send_pkt` function is commonly used by UDP, where sockets may send to a number of different destinations.

```
void ip_queue_xmit(struct sk_buff *skb);
void ip_build_and_send_pkt(struct sk_buff *skb, struct sock *sk,
                           u32 saddr, u32 daddr, struct ip_options *opt);
```

Figure 4.11: IP sending functions.

4.1.4 Network Devices and Bottom Half Handling

At the lowest layer of the network stack in the Linux kernel, packets are received from and sent to the network by network device drivers. When a network device receives data from the network, it triggers an interrupt in the operating system, which causes the corresponding interrupt handler in the device driver to be invoked.

Because interrupts suspend the operating system from all other processing while the interrupt is handled, it is desirable to have interrupt handlers do only the "important" processing and delay less important processing to a time that is more convenient for the operating system. Linux uses bottom half handlers to delay non-critical interrupt processing to a later time.

In Linux networking, a bottom half handler is used by the network device drivers to deliver incoming data packets to the network level protocols. Instead of immediately delivering incoming packets to the IP layer, for example, the interrupt handler of the network device driver puts the packet on a backlog queue and mark the network bottom half as active. Later, on the return from every system call and every slow

interrupt, if no further interrupt is running at the time, a list of 32 bottom halves is scanned. Each bottom half routine is run if it is marked active, and then automatically marked inactive. The network bottom half reads packets from the backlog queue and delivers the packets to the appropriate protocol as identified in the Ethernet header, in this case, the IP layer. As a part of the network bottom half, the IP layer delivers each packet to the appropriate transport layer protocol such as TCP or RMC.

4.2 The RMC Interface and Data Structures

In this section, we discuss the user interface to RMC and the RMC interface to the BSD and IP layers of the network stack. We also discuss the data structures that RMC uses to manage sockets.

4.2.1 The RMC Interface

The RMC protocol has been implemented to support a subset of the BSD socket calls, as listed in Table 4.1. RMC sockets support half-duplex communication. An RMC socket can be a sender or a receiver, but not both. Data flows in the forward direction, from the sender to the receivers, and control information flows in the backward direction, from a receiver to the sender.

A sample receiving application is shown in Figure 4.12. The receiver first creates a socket with protocol family `AF_RMC`, type `SOL_IP` and protocol `IPPROTO_RMC`. After binding to a well-known multicast port, the receiving application uses `setsockopt` to join a multicast group. It then uses the `recv` system call to receive data on the RMC socket. When the application finishes using the connection, it calls `close`.

The corresponding sending application is shown in Figure 4.13. The sender first creates a socket in the same way that the receiver did. The sender then binds to

```
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/rmc.h>
#define MCPORT 4096

int main(int argc, char **argv) {
   int sock,c = 0;
   struct sockaddr_in my_addr, data_addr;
   struct ip_mreq mreq;
   char buf[BUFSIZ];
   long int myaddr = inet_addr("35.9.26.153");
   long int mcaddr = inet_addr("234.5.5.5");

   /* Set up the RMC socket. */
   if ((sock = socket(AF_RMC,SOCK_IP,IPPROTO_RMC)) < 0)
      { perror("socket"); exit(1); }

   /* Bind to data (multicast) port. */
   memset(&data_addr,0,sizeof(data_addr));
   my_addr.sin_family = AF_RMC;
   my_addr.sin_addr.s_addr = INADDR_ANY;
   my_addr.sin_port = htons(MCPORT);
   if (bind(sock,(struct sockaddr*)&my_addr,sizeof(my_addr)) < 0)
      { perror("bind"); exit(1); }

   /* Join the multicast group. */
   mreq.imr_multiaddr.s_addr = mcaddr;
   mreq.imr_interface.s_addr = htonl(INADDR_ANY);
   if (setsockopt(sock,SOL_RMC,RMC_ADD_MEMBERSHIP,&mreq,sizeof(mreq)) < 0)
      { perror("setsockopt"); exit(1); }

   /* Receive the data. */
   if ((c = recv(sock,buf,sizeof(buf),0)) < 0)
      { perror("netin"); exit(1); }

   /* Close the socket. */
   close(sock);
}
```

Figure 4.12: A receiving application.

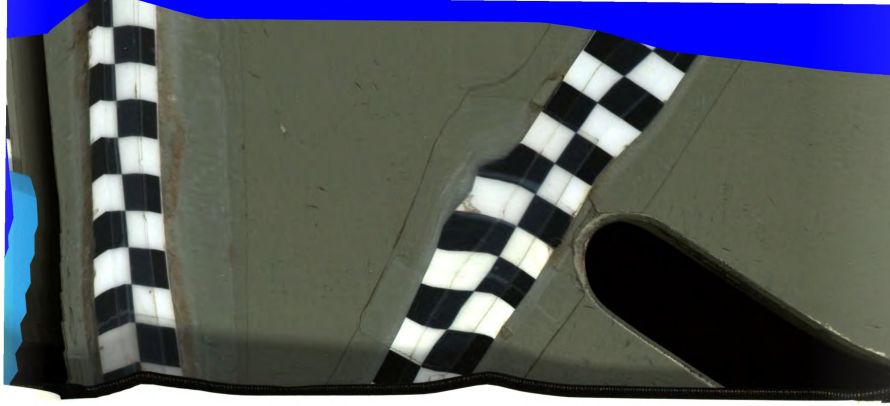
Function Name	*Meaning*
socket	Create a descriptor for use in network communication.
bind	Bind a local IP address and protocol port to a socket.
connect	Connect to a multicast IP address and protocol port.
send sendto write	Send outgoing data.
recv recvfrom read	Receive the next incoming data.
setsockopt	Change the options for the RMC socket.
close	Terminate communication and deallocate the socket descriptor.

Table 4.1: RMC-supported BSD functions.

a local port and connects to the same well-known multicast port that the receivers are bound to. The sender uses **send** to send data to the multicast group. When the sender has completed its communication, it calls **close**.

As was shown in Figure 4.1, the RMC protocol is layered between the BSD socket layer and the IP layer in the Linux kernel. RMC bypasses the INET socket layer and provides a direct interface to the BSD layer for two reasons. First, by going around the INET socket layer, the RMC implementation remains separate from the existing implementation of TCP and UDP. While RMC uses many of the data structures and functions that are provided by the INET socket layer, very few changes were made to the existing code to accommodate RMC. Second, by bypassing INET sockets, the path between the user application and the network is shorter, so the performance should be better.

The **rmc_family_ops** and the **rmc_proto_ops** vectors, which are shown in Figure 4.14, define the interface between BSD and RMC. The **rmc_family_ops** vector provides BSD with an entry point into the RMC protocol when new sockets are created. This vector is registered with BSD upon initialization of the system, and is stored as a part of the **net_families** table, which is shown in Figure 4.4. The



```
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/rmc.h>
#define MCPORT 4096
#define MYPORT 4097

int main(int argc, char **argv) {
   int sock,c = 0;
   struct sockaddr_in my_addr, data_addr;
   char buf[BUFSIZ];
   unsigned long myaddr = inet_addr("35.9.26.154");
   unsigned long mcaddr = inet_addr("234.5.5.5");

   /* Initialize the data buffer. ... */

   /* Set up the socket. */
   if ((sock = socket(AF_RMC,SOCK_IP,IPPROTO_RMC)) < 0)
      { perror("socket"); exit(1); }

   /* Bind to my address. */
   memset(&my_addr,0,sizeof(my_addr));
   my_addr.sin_family = AF_RMC;
   my_addr.sin_addr.s_addr = INADDR_ANY;
   my_addr.sin_port = htons(MYPORT);
   if (bind(sock,(struct sockaddr*)&my_addr,sizeof(my_addr)) < 0)
      { perror("bind"); exit(1); }

   /* Connect to the multicast address. */
   memset(&data_addr,0,sizeof(data_addr));
   data_addr.sin_family = AF_RMC;
   data_addr.sin_addr.s_addr = mcaddr;
   data_addr.sin_port = htons(MCPORT);
   if (connect(sock,(struct sockaddr*)&data_addr,sizeof(data_addr)) < 0)
      { perror("connect"); exit(1); }

   /* Send the data. */
   if ((c = send(sock,(void*)buf,sizeof(buf),0)) < 0)
      { perror("send"); exit(1); }

   /* Close the socket. */
   close(sock);
}
```

Figure 4.13: A sending application.

`rmc_proto_ops` vector specifies the remainder of the RMC specific functions as defined by the `proto_ops` structure, which is defined in Figure 4.5.

```
struct net_proto_family rmc_family_ops = {
   AF_RMC,     /* Unique identifier for RMC. */
   rmc_create  /* Create a new RMC socket.   */
};

static struct proto_ops rmc_proto_ops = {
   AF_RMC,            /* Unique identifier for RMC.                    */
   rmc_dup,           /* Duplicate an RMC socket.                      */
   rmc_release,       /* Release an RMC socket.  Free all assoc. mem.  */
   rmc_bind,          /* Bind an RMC socket to a local port.           */
   rmc_connect,       /* Connect an RMC socket to a multicast addr.    */
   rmc_accept,        /* Not supported.                                */
   rmc_listen,        /* Not supported.                                */
   rmc_shutdown,      /* Close an RMC socket connection.               */
   rmc_setsockopt,    /* Set RMC options.                              */
   rmc_getsockopt,    /* Get RMC option values.                        */
   rmc_sendmsg,       /* Send a message on the RMC socket.             */
   rmc_recvmsg,       /* Receive a message on the RMC socket.          */
...
};
```

Figure 4.14: The RMC network protocol family structure.

The interface between RMC and IP is defined by the `rmc_protocol` vector, which defines the `inet_protocol` structure (Figure 4.10) for RMC. During initialization of the system, RMC also registers the `rmc_protocol` vector with the IP layer. Among other fields, the `rmc_protocol` vector includes a pointer to the RMC handler, `rmc_ip_rcv`, and a unique protocol ID, `IPPROTO_RMC`. To identify IP packets as RMC packets, RMC sets the protocol field of the IP header of all outgoing packets to `IPPROTO_RMC`. As IP processes incoming packets, it uses the protocol field in the IP header to identify packets of type `IPPROTO_RMC` and deliver them to the `rmc_ip_rcv` function.

For sending packets, the RMC protocol implementation makes use of two IP send-

ing routines. The `ip_queue_xmit` routine takes care of building IP packets and routing them to the destination specified in the INET socket structure. For the sender, the specified destination is the multicast group, and for the receivers, the specified destination is the sender. Since most traffic flows from the sender to the multicast group and from multicast group members back to the sender, RMC uses `ip_queue_xmit` to transmit most outgoing data packets. However, JOINACCEPT and NAKERR messages are delivered only to the group member that is joining the group or the group member affected by the error. Thus, packets of these types must be routed to a destination other than the one specified in the `sock` structure. In this case, RMC first calls `ip_route_output` to route the packet to the appropriate destination, and then calls `ip_build_and_send_pkt` to build the IP packet and send it to the destination specified in the function parameters.

4.2.2 RMC Data Structures

RMC uses the control blocks in both the INET socket structure and in socket buffers to maintain information that is specific to RMC sending and receiving sockets. In the INET socket structure (Figure 4.7), the RMC socket control block is defined by the `rmc_opt` structure, and is stored as a part of the `tp_pinfo` union. The `rmc_opt` structure is shown in Figures 4.15 and 4.16. All RMC sockets use the multicast hash information in the RMC socket control block to support quick access to each RMC socket. For receiving sockets, the RMC socket control block includes variables that maintain the receive window, the NAK list, and the NAK timer, as well as the out-of-order queue. Fields that are relevant only to sending sockets include those that maintain the send window and rate window, as well as retransmission and keepalive information.

```
struct rmc_opt {
   /* multicast hash info */
   struct sock *mc_next;  /* Next RMC socket in this hash list.     */
   struct sock *mc_prev;  /* Previous RMC socket in this hash list. */
   unsigned short master; /* Is this the master receiver for rmc?   */

/* Receive information */
   __u32 rcv_wnd;      /* Seq. number of first byte in receive window. */
   __u32 rcv_wnd_size; /* Size of the receive window.                  */
   __u32 rcv_nxt;      /* Seq. number that we want to receive next.    */
   __u32 unnaked_nxt;  /* Last unnaked sequence number.                */
   __u32 copied_seq;   /* Next seq. number to copy to the user.        */
   int wnd_time;       /* Next time we can request a window reduction. */
   __u32 rcv_thresh1;  /* Border between safe and warning regions.     */
   __u32 rcv_thresh2;  /* Border between warning and critical regions. */

/* nak structures */
   struct timer_list nak_timer;  /* Timer to control resending of NAKs. */
   struct sk_buff_head nak_list; /* List of pending NAKs.               */
   struct sk_buff_head out_of_order_queue; /* Out of order segments     */
...
}; /* struct rmc_opt */
```

Figure 4.15: RMC receiver specific information.

Packets are passed to and from the RMC layer in socket buffers. RMC uses the control block in the socket buffer structure to hold RMC specific data. The control block is a block of 48 bytes set aside by the socket buffer structure for transport layer specific information. The RMC socket buffer control block is defined by the `rmc_skb_cb` structure, which is shown in Figure 4.17. Information in the RMC control block includes the beginning and ending sequence numbers of the data packet, as well as time stamps for when the packet arrives or is sent and for when the packet expires.

4.2.3 RMC Ports

Like TCP and UDP, RMC uses ports to identify participating processes on the sending and receiving machines. RMC hashes each sockets by its bound port number, which

```
struct rmc_opt {
...
/* Send information */
   int members;        /* Number of receivers in the multicast group.    */
   __u32 snd_wnd;      /* Seq. number of first packet in send window.    */
   __u32 snd_wnd_size; /* Size of the send window.                       */
   __u32 snd_nxt;      /* First seq. number to use on next send.         */
   __u32 prev_snd_nxt; /* first seq. number to use on next send          */
   int advance_time;   /* How long to keep an skb in the window.         */
   int min_sends;      /* Min. number of sends before releasing an skb.  */
   __u32 write_seq;    /* Next seq. number to use for a new packet.      */
   int total_sends;    /* Total number of sends/resends in the window.   */
   int wait_for_mem;   /* Flag, true if waiting for memory.              */
   struct sk_buff *send_head;  /* Next packet to send on the backlog.    */

   /* Rate-based flow control information. */
   __u32 last_pkt;         /* End of the rate window.                    */
   __u32 snd_rate_wnd;     /* Start of the rate window.                  */
   __u32 max_snd_rate_wnd; /* Maximum value for the rate window.         */
   __u32 snd_ssthresh;     /* Slow start size threshold.                 */
   struct timer_list slow_start_timer; /* Timer to control packet trans. */
   __u32 wnd_increase;     /* Size of possible rate window increase.     */
   struct probe join_req;  /* Initial seq. number join information.      */
   int rtt;                /* Estimated round-trip time.                 */

   /* Retransmission information */
   struct retrans_req *retrans_queue; /* List of NAKs to service.        */
   struct timer_list retrans_timer;   /* Timer to control packet retrans. */

/* keepalive structures */
   struct timer_list ka_timer; /* Timer to control keepalive pkt. trans.  */
   int ka_time;                /* Current keepalive interval.             */
   int sent_kas;               /* Number of consecutive keepalive pkts.   */
   unsigned int idle_time;     /* Last time non-keepalive pkt. sent.      */

__u8 flags;                 /* Miscellaneous flags.                    */
}; /* struct rmc_opt */
```

Figure 4.16: RMC sender specific information.

```
struct rmc_skb_cb {
    struct inet_skb_parm header; /* For incoming frames.       */
    __u32 seq;                   /* Starting sequence number   */
    __u32 end_seq;               /* Ending sequence number     */
    unsigned long when;          /* Time the packet was last sent. */
    unsigned long timeout;       /* For NAKs, when to resend.  */
    unsigned char type;          /* RMC packet type.           */
    __u8 flags;                  /* Miscellaneous flags.       */
    __u8 tries;                  /* Times this packet has been sent. */
};
```

Figure 4.17: RMC socket buffer information.

is stored in the num field in the INET socket structure (Figure 4.7). References to RMC sockets are stored in a hash table so that destination sockets can be quickly located as packets arrive from the network.

In a multicasting environment, it is possible that multiple processes on the same machine wish to receive data from the same multicast source. In this situation, these processes bind to the same local port, but each process in this group should be given its own copy of the incoming multicast data. One way to handle this situation is to immediately deliver a copy of each incoming packet destined for the shared multicast endpoint to each of the destination sockets in the group. As a result, however, each socket in the group has to reassemble the incoming data stream and handle its own error recovery and flow control. This method clearly results in a replication of effort among the sockets on the same host that are listening to the same multicast group.

To reduce both the local processing needs and the amount of feedback to the sender, it is preferable to have a single controlling mechanism for the group of sockets on the same host that are listening to the same multicast endpoint. RMC implements this idea by assigning one of the sockets to be a master socket and making all other sockets in the group to be children of the master socket. Master sockets are marked by

setting the `master` field in the INET socket structure (Figure 4.7) to 1. All incoming data is first delivered to the master socket, which reassembles the incoming data stream and handles error recovery and flow control for the group. The master socket delivers a separate copy of the in-order data to each of its children.

The RMC hashing structure is shown in Figure 4.18. RMC supports master and children sockets by adding the multicast hash list fields `mc_next` and `mc_prev` to the RMC socket control block (Figure 4.15). These fields are used to maintain a linked-list of the RMC sockets listening to the same multicast source. A master socket is hashed by its bound address and uses the `mc_next` variable to point to the list of its children. In the figure, the normal hash list extends from the RMC hash table entry, and the multicast hash list extends down from the master socket. Hashing the master socket in this way makes it easy for the receive interrupt handler to initially deliver incoming data to the master socket. Extending the multicast hash list from the master socket makes it easy for the master socket to distribute in-order data to all its children.

4.3 RMC Protocol Implementation

Now that we have discussed where and how RMC fits into the Linux networking stack, as well as the data structures that RMC uses to maintain sockets, we will describe the implementation of the RMC protocol. We will first discuss the implementation of RMC socket creation and connection initialization. We will then discuss the architecture of the RMC receiver, followed by the architecture of the RMC sender. We will finish with a discussion of RMC connection tear down.

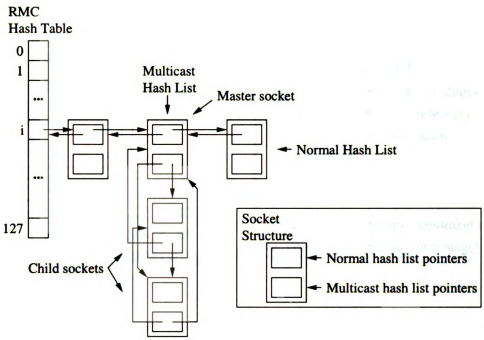


Figure 4.18: RMC's hashing table structure.

4.3.1 RMC Socket Creation and Connection Initialization

Both the sender and the receiver must perform special tasks when an RMC socket is created and when the RMC connection is first established. User applications use the `socket` system call to create RMC sockets. As a result of this call, the BSD layer allocates a BSD socket structure and calls the `rmc_create` function. The `rmc_create` function allocates an INET socket structure for the new socket and creates links between the BSD and INET socket structures. In addition, the `ops` field of the BSD socket structure is set to point to RMC's protocol operations vector, `rmc_proto_ops`. Further initialization of the INET socket structure are performed by the `sock_init_data` and `rmc_init_sock` functions. This initialization includes setting the size of the rate window to a single segment and the state of the socket to `RMC_LISTEN`. The resulting structures are shown in Figure 4.19.

After a receiving application creates a new RMC socket, and before it is able to receive data on an RMC socket, the application must bind the socket to a well-

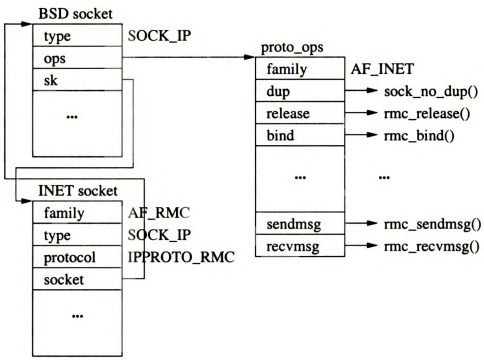


Figure 4.19: The RMC socket structures.

known multicast port and join an IP multicast group. The application uses the `bind` system call to bind the socket. In RMC, the `bind` system call is implemented by the `rmc_bind` function, which initializes the `num` in the INET socket structure to the bound port. The application calls `setsockopt` with `RMC_ADD_MEMBERSHIP` and the desired multicast address as parameters to join the IP multicast group. The `rmc_setsockopt` function, which implements the `setsockopt` system call for RMC, calls the IP `ip_add_mc_membership` function to carry out the join process.

After the receiving application has bound the socket and joined the IP multicast group, it may begin to receive data on the socket by calling one of the `recv` system calls. On receipt of the first data packet, the receiver sets up the receive window by initializing the `rcv_nxt`, `rcv_wnd`, `unnaked_nxt`, and `copied_seq` variables in the RMC socket control block (Figure 4.15) to the beginning sequence number of the packet. The receiver then calls `rmc_join_mc` to send an RMC join request to the

sender.

Before transmitting data on an RMC socket, the sending application must also bind the socket to a local port using the `bind` system call. In addition, the sending application must then connect the socket to the multicast port. To achieve this, the application calls the `connect` system call with the desired destination address as a parameter. The `rmc_connect` function, which implements the `connect` system call for RMC, initializes the `daddr` and `dport` variables in the INET socket structure (Figure 4.7) to the specified multicast destination address and port.

After the socket is bound and connected, the sender may use one of the send system calls to transmit data on the RMC socket. The sending socket is initially in the `RMC_LISTEN` state and remains in this state until it receives the first join request. Because it is useless to transmit data to an empty multicast group, the sending function transmits the first data packet and puts itself to sleep while waiting for incoming join requests. When the keepalive controller runs, it observes that the state is `RMC_LISTEN` and transmits the first packet instead of normal keepalive packets. After the sender receives the first join request, the socket state is set to `RMC_ESTABLISHED` and the sending function is awakened. At this point, the connection is established and normal transmission is allowed to continue.

4.3.2 The RMC Receiver

The architecture of the RMC receiver is shown in Figure 4.20. The RMC receiver has three major components: the packet processor, the NAK manager, and the application interface. The packet processor manages incoming data packets, the NAK manager controls NAK retransmissions, and the application interface delivers available data to the receiving application.

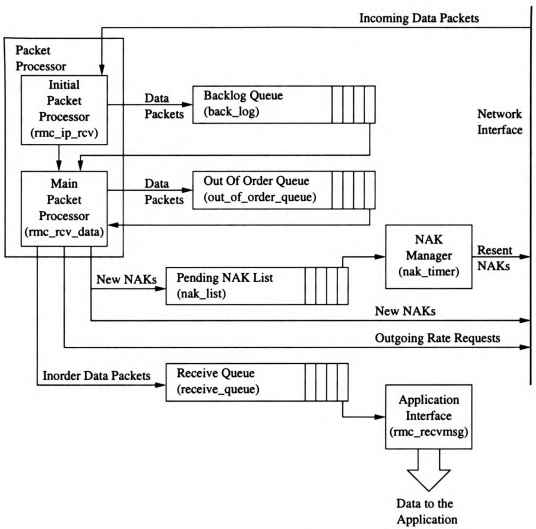


Figure 4.20: The RMC receiver.

The receiver uses three packet queues to manage data packets as they pass through the receiver. The backlog queue, which is implemented by the `back_log` socket buffer list in the INET socket structure (Figure 4.7), is used to temporarily hold data packets that arrive while the destination socket is locked. The out-of-order queue is implemented by the `out_of_order_queue` socket buffer list in the RMC socket control block (Figure 4.15) and holds packets that cannot yet be integrated into the data stream due to one or more gaps in the data stream. The receive queue, implemented by the `receive_queue` socket buffer list also located in the RMC socket control block,

holds in-order packet that are available to the receiving application.

The receive window, as shown in Figure 4.21, is composed of the receive queue and the out-of-order queue. Several variables in the RMC socket control block (Figure 4.15) are used to mark different points in the receive window. The `rcv_wnd` variable marks the sequence number of the first data byte in the receive queue and the `rcv_nxt` variable marks the sequence number of the last in-order data byte stored in the receive queue. The `rcv_wnd_size` variable gives the size of the receive window. It can be added to `rcv_wnd` to calculate the right edge of the receive window. The variable `unnaked_nxt` gives the value of the sequence number of the last data byte in the receive queue. In addition to the three data queues, the NAK list, implemented by `nak_list` in the RMC socket control block (Figure 4.15), holds pending NAKs.

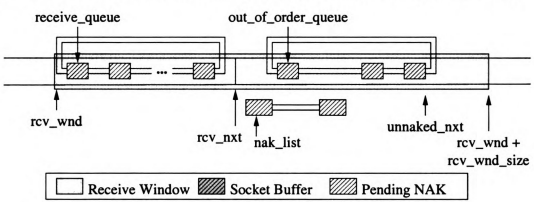


Figure 4.21: Implementation of the receive window.

Packet processor. Incoming data packets are initially delivered to the RMC protocol through the packet processor, which has two parts. Packets are first processed by the initial packet processor, which is implemented by the `rmc_ip_rcv` function. This function receives packets from the IP layer and calls the `rmc_lookup` function to locate the destination socket in the hash table, `rmc_hash`, according to the packet's destination. If the destination socket is located and it is not currently locked, the packet is delivered immediately to the main packet processor. Otherwise the packet

is appended to the backlog queue for the socket. Packets in the backlog queue are passed to the main packet processor the next time the socket becomes unlocked.

The main packet processor is implemented by the **rmc_rcv_data** function, which is shown in Figure 4.22. The responsibilities of the main packet processor include reassembling the incoming data packets into the original data stream, determining when to generate NAKs, and monitoring the receive window to determine when to send rate requests.

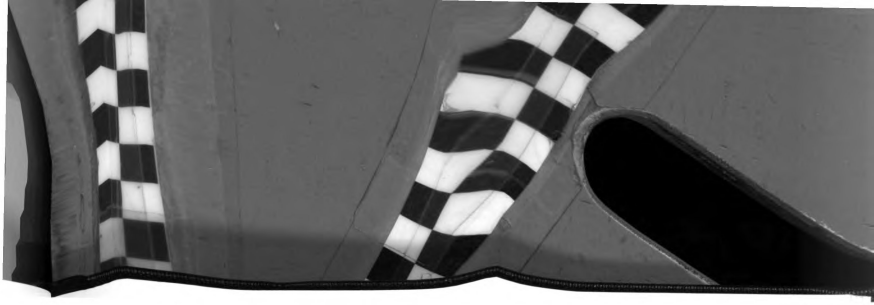
The main packet processor calls the **rmc_send_nak** function to generate and send NAKs for missing data. Each new NAK is inserted into **nak_list** and is given a timeout value by setting the **timeout** field in the RMC control block for the socket buffer (Figure 4.17) to the current time plus four times the estimated round-trip time. If it has not already been activated, the NAK monitor is scheduled to run when the new NAK will expire.

Rate requests are prepared and sent by the **rmc_req_rate_reduce** function. To limit the number of rate requests, each function that calls **rmc_req_rate_reduce** is responsible for making sure that the appropriate amount of time has passed since the last rate request was sent before sending a new rate request. The socket control block variable **wnd_time** (Figure 4.15) holds the earliest time when the next rate request may be sent. Before making a rate request, the calling function compares the value of **wnd_time** to the current time. If the current time is greater than or equal to **wnd_time**, the **rmc_req_rate_reduce** function may be called. In addition, every time the **rmc_req_rate_reduce** function is called, the calling function sets **wnd_time** to the earliest time that the next rate request may be sent.

When the main packet processor receives a packet, it determines what actions to take by comparing the sequence number range of the packet, indicated by the RMC socket buffer control block variables **seq** and **end_seq**, to the sequence space of the

```
int rmc_rcv_data(struct sock *sk, struct sk_buff *skb) {

   struct rmchdr* rh = skb->h.rmc;
   __u32 inseq = RMC_SKB_CB(skb)->seq;
   struct rmc_opt* rp = &(sk->tp_pinfo.af_rmc);

   if (rp->rcv_nxt == 0) { /* This is the first packet. */
      /* Update the RMC socket variables and  send a join request. ... */ }

   /* Check where this packet fits into the incoming data stream. */
   if (after(RMC_SKB_CB(skb)->end_seq,(rp->rcv_wnd + rp->rcv_wnd_size))) {
      /* Out of bounds, discard the packet. */
      in_oob(sk,skb);
      kfree_skb(skb);
      rmc_statistics.RmcInOOB++;
      return 0;
   }

   if (inseq == rp->rcv_nxt) { /* In sequence. */
      inorder_deliver(sk,skb);
      /* Check the out-of-order queue and update naks. */
      ck_ooq(sk);
      advance_nak_list(sk);
   } else if (after(inseq,rp->rcv_nxt)) {
      /* Out of order, possibly not yet received, store in the ooo queue. */
      qinsert(&rp->out_of_order_queue,skb);
      /* update pending naks and send new naks, if necessary */
      update_nak_list(sk,skb);
      if (after(inseq,rp->unnaked_nxt)) /* Need to nak. */
         rmc_send_nak(sk,skb->nh.iph->saddr,rh->src,rp->unnaked_nxt,inseq);
   } else {
      /* Previously received, free the buffer. */
      rmc_statistics.RmcInDiscards++;
      kfree_skb(skb);
      return 0;
   }

   /* Advance the unnaked seq. number if necessary. */
   if (after(RMC_SKB_CB(skb)->end_seq,rp->unnaked_nxt))
      rp->unnaked_nxt = RMC_SKB_CB(skb)->end_seq;
   /* Check the window for overflow. */
   ck_wnd(sk,skb);
   return 1;
} /* rmc_rcv_data */
```

Figure 4.22: The receive function.



receive window. The packet processor may take one of four actions listed below.

1. If **end_seq** is greater than the end of the receive window, the packet will not fit in the window. The packet is discarded and the **in_oob** function is called. The **in_oob** function calls **rmc_req_rate_reduce** to initiate an urgent rate request. In addition, if the packet leaves a gap at the end of the receive window that is large enough to hold an additional packet, **in_oob** calls **rmc_send_nak** to generate a new NAK for the free space.

2. If **seq** is equal to **rcv_nxt**, the packet is in sequence, and it is appended to the receive queue. The **data_ready** function is called to wake up any kernel processes waiting for data to be put on the receive queue. The **ck_ooq** function is then called to check the out-of-order queue for packets that may now be in sequence. In-order packets are removed from the out-of-order queue and appended to the receive queue. In addition, **advance_nak_list** is called to remove any NAKs from the NAK list for data with sequence numbers less than or equal to **end_seq**.

3. If **seq** is greater than **rcv_nxt**, the packet is not in order and has not been previously received. It is stored in the out-of-order queue. The **update_nak_list** function is called to remove any NAKs from the NAK list for data in the range of the out-of-order packet only.

4. If none of the above conditions apply, the packet has previously been received and is discarded.

For each packet that is successfully received and stored in the receive window, the main packet processor checks the state of the receive window for possible overflow by calling the **ck_wnd** function, which is listed in Figure 4.23. The **ck_wnd** function

first checks if enough time has passed since the last rate request by examining the `wnd_time` variable. If `wnd_time` is less than the current time, the `ck_wnd` function calculates the amount of free space in the receive window. If the free space is less than the critical threshold, `rcv_thresh2`, the `ck_wnd` function calls `rmc_req_rate_reduce` with the urgent parameter set to 1. If the free space is less than the warning threshold, `rcv_thesh1`, `ck_wnd` calls `rmc_req_rate_reduce` with the urgent parameter set to 0. If a rate request is sent, the `wnd_time` variable is reset to one round-trip time from the current time.

```
static void ck_wnd(struct sock *sk, struct sk_buff *skb) {
   struct rmc_opt *rp = &(sk->tp_pinfo.af_rmc);
   __u32 curr_last_pkt;
   __u32 ad_rate,curr_wnd;
   __u32 time = JIFFIES;

   /* Calculate the windows */
   ad_rate = ntohl(skb->h.rmc->rate);
   curr_last_pkt = rp->rcv_wnd + rp->rcv_wnd_size;
   curr_wnd = curr_last_pkt - rp->rcv_nxt;

   if (rp->wnd_time <= time) { /* Send only one request per rtt. */
      if (curr_wnd <= rp->rcv_thresh2) {
         /* Send urgent control packet. */
         rmc_req_rate_reduce(sk,skb,1);
         rp->wnd_time = time + rp->rtt;
      }  else if ((curr_wnd <= rp->rcv_thresh1)
         && (curr_wnd < (4*rp->rtt*ad_rate))) {
         /* Send warning control packet. */
         rmc_req_rate_reduce(sk,skb,0);
         rp->wnd_time = time + rp->rtt;

   }
   return;
} /* ck_wnd */
```

Figure 4.23: The RMC ck_wnd function.

NAK manager. The second part of the receive architecture is the NAK manager, whose duty it is to monitor the NAK list and resend pending NAKs at appropriate intervals. The NAK manager is implemented by the `nak_timer` timer, which is stored as a part of the RMC socket control block (Figure 4.15). The interrupt handler for the NAK timer is the `nak_timer` function. Each time the NAK list becomes non-empty, the NAK manager is scheduled to run by setting `nak_timer` to fire at the time the newly created NAK will expire. When the timer fires, the `nak_timer` function examines each NAK on the NAK list. If the value of the `expire` field in the NAK's control block is less than the current time, the NAK has expired. The NAK is resent and `expire` is reset to four round-trip times from the current time. Finally, if the NAK list is non-empty, the NAK manager reschedules itself by setting `nak_timer` to fire at the earliest time when a NAK on the list will expire.

Application interface. The final part of the receiver is the application interface. The application interface implements the receive system calls and delivers data to receiving applications. The `rmc_recvmsg` function implements the application interface. This function pulls data packets from the receive queue and copies the requested number of data bytes to user-level memory using the Linux `memcpy_toiovec` function. It uses the `copied_seq` variable from the RMC socket control block (Figure 4.15) to track the sequence number of the next data byte that should be delivered to the application. After a data packet has been completely consumed by the application, the data packet is released from the receive window and the receive window is slid past the consumed packet by setting `rcv_wnd` to the ending sequence number of the packet. If there are no data packets in the receive queue when `rmc_recvmsg` is called, the process releases the lock on the socket and puts itself to sleep by setting its state to `TASK_INTERRUPTIBLE` and calling `schedule`. The process will be awakened

by the main packet processor when new data has been inserted in the receive queue. At this point, `rmc_recvmsg` relocks the socket and continues to copy available data to the application.

4.3.3 The RMC Sender

An overview of the RMC sender architecture is shown in Figure 4.24. The RMC sender may be thought of as five concurrent tasks: the application interface, the transmitter, the feedback processor, the retransmitter, and the keepalive manager. The application interface and the transmitter work together to transmit original data. The feedback processor processes rate requests and works with the retransmitter to resend data packets. Finally, the keepalive controller sends keepalive messages during times when both the transmitter and the retransmitter are idle.

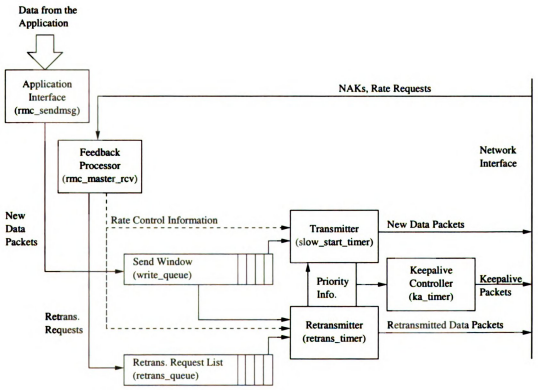


Figure 4.24: The RMC sender.

The sender uses the send window, shown in Figure 4.25, to buffer data packets that have been sent and data packets that are waiting to be sent. The send window is implemented by the **write_queue** socket buffer queue, which is stored as part of the INET socket structure (Figure 4.7). All other variables that are used to maintain the send window are stored in the RMC socket control block (Figure 4.16). The **snd_wnd** variable holds the beginning sequence number of the first packet in the send window. The **snd_wnd_size** variable holds the size of the send window, and can be added to **snd_wnd** to calculate the ending sequence number of the send window. The **send_head** variable points to the next packet queued to be sent. If all data packets in the send window have been sent, the **send_head** variable is set to **NULL**. The **snd_nxt** variable holds the value of the sequence number of the next data byte to be sent.

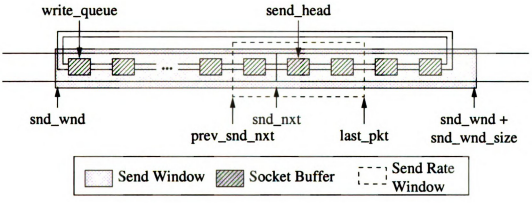


Figure 4.25: Implementation of the send window.

Because kernel processes should not block, it is impractical to implement rate-based transmission in the kernel by introducing a delay between each packet. For this reason, rate-based transmission is implemented as a window within the send sequence space. The rate window, outlined by the dashed line in Figure 4.25, begins at the point **prev_snd_nxt** and extends to the point **last_pkt**. The rate window advances at set send intervals and packets within the rate window may be sent at any rate. In this implementation, the send interval is ten milliseconds, which is

the basic unit of time in the Linux kernel. The size of the window, held in the **snd_rate_wnd** variable in the RMC socket control block (Figure 4.16), is determined by the desired rate, which is expressed as the number of bytes that can be sent in one sending interval. The upper bound on the size of the rate window is held in the **max_snd_rate_wnd** variable, also located in the RMC socket control block. The value of **max_snd_rate_wnd** is currently set to 12450 bytes. This rate window allows a maximum transmission rate of approximately 9.5 Mbps and was chosen because the underlying network of the test environment was 10 Mpbs Ethernet.

Application interface. Data enters the protocol through the application interface, which is initiated when the sending application calls one of the **send** system calls on an RMC socket. The main responsibility of the application interface, which is implemented by the **rmc_sendmsg** function, is to fragment the data into RMC packets and make the packets available for transmission by inserting them in the send window. Because it is more efficient to send larger data packets, the application interface first attempts to add data to the last packet in the queue of packets to be sent. The remaining data is fragmented into segments that do not exceed the maximum segment size. Memory is allocated for each segment by calling the **sock_wmalloc** function. If all the write memory for the socket is currently being used, the process puts itself to sleep by adding itself to a wait queue, setting its state to **TASK_INTERRUPTIBLE**, and calling **schedule**. Each time write memory is released in the **sock_wfree** function, the **write_space** function is called to awaken all processes waiting for write memory. After memory has been allocated, the application interface reserves space for lower layer headers, copies the data from user space using the Linux **csum_and_copy_from_user** function, and builds the RMC header. Each data packet is then passed to the **rmc_send_skb** function.

The `rmc_send_skb` function, which is shown in Figure 4.26, is responsible for adding packets to the send window and either sending them immediately or queuing them for later transmission. Each packet is stored in the send window by calling `skb_queue_tail` for the `write_queue` socket buffer queue. Two conditions are tested to determine if the packet may be sent immediately. First, the `send_head` variable must be `NULL`, signifying that there are no other packets waiting to be sent. Second, the ending sequence number of the packet must be less than the value of `last_pkt`, meaning that the packet falls within the current rate window. If the packet cannot be sent immediately, it is queued in the backlog for later transmission. If however, the packet can be sent immediately, the keepalive timer is stopped and the packet is passed to the `complete_skb_xmit` function. The `complete_skb_xmit` function completes the RMC header for the packet and records the sending time in the `when` field in the socket buffer's control block (Figure 4.17). A checksum is computed by calling the `rmc_send_check` function, and the packet is passed to `ip_queue_xmit` for transmission. On return from `complete_skb_xmit`, `rmc_send_skb` attempts to free any expired buffer space and then restarts the keepalive timer at the minimum keepalive interval.

Transmitter. The transmitter is responsible for taking packets from the backlog and sending them to the multicast group according to the current rate. The transmitter, which is implemented by the `slow_start_timer` timer in the RMC socket control block (Figure 4.16), enforces both the slow start and the congestion avoidance phases of the rate control. Each time the transmitter fires, it determines how many bytes of data were sent since the previous run by subtracting the current value of the next sequence number to be sent, `snd_nxt`, from the beginning of the current rate window, `prev_snd_nxt`. It uses this value as an upper bound on the increase

```
int rmc_send_skb(struct sock *sk, struct sk_buff *skb, int force_queue) {
    struct rmc_opt* rp = &(sk->tp_pinfo.af_rmc);
   /* Save in the queue and send the packet */
    RMC_SKB_CB(skb)->tries = 0;
    __skb_queue_tail(&sk->write_queue,skb);

    if (!force_queue && rp->send_head == NULL && rmc_snd_test(sk,skb)) {
       /* ok to send right away */
       cancel_keepalive(sk);
       complete_skb_xmit(sk,skb,GFP_KERNEL);
       rp->snd_nxt = RMC_SKB_CB(skb)->end_seq;
       /* Free send buffer space if near full of sent data. */
       if (rp->snd_nxt > (rp->snd_wnd + rp->snd_wnd_size - 2*sk->mss))
          advance_window(sk);
       reset_keepalive(sk,RMC_MIN_KATIME);
       return 0;
    }

    /* queue the packet */
    if (rp->send_head == NULL)
       rp->send_head = skb;
    return 0;
} /* rmc_send_skb */
```

Figure 4.26: The rmc_send_skb function.

of the rate window size, **snd_rate_wnd**. The transmitter calculates a new value for **snd_rate_wnd** according to whether the transmission is in slow start or congestion avoidance. If the transmission is in the slow start phase, the rate window is allowed to increase by the number of bytes sent during the previous interval. If, however, the transmission is in the congestion avoidance phase, the rate window is only allowed to increase by the maximum segment size. In addition, the value of **max_snd_rate_wnd** is used as an upper bound on the value of **snd_rate_wnd**. After the size of the new rate window has been calculated, the transmitter advances **prev_snd_nxt** to **snd_nxt** and **last_pkt** to **snd_nxt** plus **snd_rate_wnd**. The transmitter begins sending packets, starting with the packet at **send_head**. It continues to send all packets that fall within the rate window, advancing **send_head** and **snd_nxt** as necessary. After all

data packets in the current rate window have been sent, the transmitter reschedules itself by setting `slow_start_timer` to expire in 10 ms.

Feedback processor. The feedback processor handles feedback from the receivers in the form of NAKs and rate requests and forwards the feedback to appropriate entities in the sender. When a NAK arrives at the sender, the `add_retrans_req` function is called to allocate and initialize a new `retrans_req` structure. The `retrans_req` structure, as shown in Figure 4.27, is composed of the beginning and ending sequence numbers for the request as well as a pointer to the next socket buffer that should be sent for the request. The new request is inserted in the `retrans_queue` list in the RMC socket control block (Figure 4.16). Multiple requests for the same data range are discarded. Because retransmissions have priority over both original transmissions and keepalive messages, the `slow_start_timer` and `keepalive_timer` timers are suspended. To initiate the congestion avoidance phase, the feedback processor halves the value of `snd_rate_wnd`, and the `retrans_timer` function is called to start the retransmitter.

```
struct retrans_req {
   __u32 begin;
   __u32 end;
   struct sk_buff *retrans_head;
   struct retrans_req *next;
};
```

Figure 4.27: The retransmission request structure.

Rate requests are handled by the `reduce_rate` function. If the rate request is urgent, `reduce_rate` stops forward transmission by setting `snd_rate_wnd` to zero. The `flag` field in the RMC socket control block is set to `RMC_FLAG_KA` and the keepalive controller is set to fire in one round-trip time.

For non-urgent rate requests, **reduce_rate** sets **snd_rate_wnd** to the size of a single segment to initiate a slow start phase. The threshold between slow start and congestion avoidance is marked by setting the **snd_ssthresh** field in the RMC socket control block to one-half of the rate requested in the incoming rate request. The slow start timer is then restarted to continue forward transmission.

Retransmitter. It is the retransmitter's responsibility to serve retransmission requests from the retransmission request list, **retrans_queue**. The retransmitter is implemented by the **retrans_timer** timer in the RMC socket control block (Figure 4.16) and the handler for the timer is the **retrans_timer** function. When the retransmission timer fires, the **retrans_timer** calls **rmc_retrans_xmit** to send a window of packets according to the current rate. If the **retrans_queue** is non-empty after sending the window of packets, the **retrans_timer** reschedules itself to run in the next send interval. When the retransmitter has complete resending all requested data, the **retrans_timer** function resets the keepalive and slow start timers to continue forward transmission.

Keepalive controller. The final piece of the sender architecture is the keepalive controller, which is responsible for sending keepalive packets during idle times. The keepalive controller is implemented by the **ka_timer** timer in the RMC socket control block (Figure 4.16). After the first packet in the outgoing data stream is sent, the **ka_timer** timer is scheduled to run in the minimum keepalive time by calling **reset_keepalive**. Each time the keepalive controller fires, the **ka_timer** function is executed. This function sends a keepalive message to the group and sets the keepalive time value, **ka_time** to twice the previous value, up to the maximum keepalive time value, **MAX_KA_TIME**, which is 2 seconds. The keepalive controller

then reschedules itself to run in the time specified by the `ka_time` variable. Because the keepalive controller has the lowest transmission priority, it is suspended by calling **`cancel_keepalive`** when either the transmitter or the retransmitter become active. When the transmitter and the retransmitter complete their current task, they reschedule the keepalive controller by calling **`reset_keepalive`**. Each time the keepalive controller is suspended and reset, **`ka_time`** is set to the minimum value **`RMC_MIN_KATIME`**, which is 100 ms.

The keepalive controller handles two special cases. The first special case occurs when the sending socket is in the **`RMC_LISTEN`** state. Each time the **`ka_timer`** function runs, it checks the socket state. If the state is **`RMC_LISTEN`**, **`ka_timer`** sends the packet at the head of the send window instead of a keepalive packet. The second special case occurs when the feedback processor sets the **`RMC_FLAG_KA`** flag after receiving an urgent rate request. In this case, to restart forward transmission, the **`ka_timer`** function clears the flag and reschedules the transmitter by setting the **`slow_start_timer`** timer to two round-trip times. Two round-trip times are given rather than one to give the receiving hosts ample time to respond to the keepalive message.

4.3.4 RMC Socket Close

After the user application has finished communicating on an RMC socket, it calls the **`close`** system call to tear down the RMC connection. The **`rmc_shutdown`** function implements the **`close`** system call for RMC. The main responsibility of this function is to stop all activity occurring on behalf of the socket and release all the socket data. For a receiving socket, closing the socket is simply a matter of sending a LEAVE message to the sender by calling **`rmc_leave_ms`**, and deallocating all of the data structures that were used to maintain the socket by calling the **`rmc_destroy_sock`**

function.

At the sender, care must be taken that all data in the send window has been transmitted and receiver sockets have had a chance to request retransmissions before the sender frees all of its socket data structures. To ensure this, when a sender socket issues a close, it must go through two wait periods. During the first wait period, the sender calls `rmc_send_fin` which sets the FIN flag in the last outgoing packet. The `FIN` flag signals receivers that the sender has completed all transmissions. The sender then puts itself to sleep by adding itself to a wait queue, setting its state to `TASK_INTERRUPTIBLE`, and calling schedule. After the transmitter sends the last packet in the send window, the value of `send_head` becomes `NULL` and the transmitter calls `state_change` to awaken the waiting process. The second wait period is used to wait for retransmission requests from the receivers. This wait period is implemented by putting the sending process to sleep while all the data packets in the send window are allowed to expire. During this phase, the keepalive controller continues to send periodic keepalive messages. After all packets in the send window have expired and a minimum amount of idle time has passed, the keepalive controller awakens the sleeping process by calling `state_change`. After the second wait period expires, the `rmc_destroy_sock` function is called to delete timers for the socket and free the data structures used to maintain the socket.

Chapter 5

Performance Evaluation

In this chapter we evaluate the performance of the RMC protocol. We first investigate the behavior of RMC as implemented in the Linux kernel, through experimentation in a local network testbed. In this environment, we are able to observe the performance of the protocol with up to eight receivers. We then use simulation to study the performance of RMC in larger networks. Using a simulated environment allows us to test RMC for up to 100 receivers, while varying network conditions such as network speed, network delay, and loss characteristics. In each test, we vary kernel buffer size from 64 KB to 1 MB and use throughput to assess the protocol effectiveness.

5.1 Experimental Study

We used a local network testbed to evaluate the performance of RMC in a working local area network. In this section we discuss the conditions and results of this experimental study.

Quantity	*Processor*	*RAM*
6	Pentium II 300 MHz	128 MB
1	Pentium Pro 200 MHz	256 MB
1	Pentium Pro 180 MHz	64 MB

Table 5.1: Test machines.

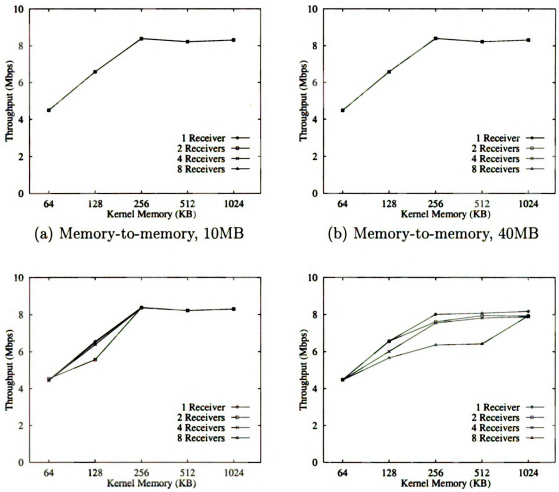
5.1.1 Experimental Conditions

For the experimental study, we used 9 PC's running RedHat Linux 5.0, kernel versio 2.1.103. In each case, the sender was a Pentium 166MHz with 64 MB of RAM. Th eight machines listed in Table 5.1 served as receivers. All machines were connecte using 10 Mbps Ethernet. Each experiment was repeated five times, and the minimu and maximum values were thrown out. Averages of the remaining three measuremen are reported here.

To study the effect of different numbers of receivers in a local environment, w conducted tests for 1, 2, 4 and 8 receivers. In the first three cases all receivers we Pentium II 300MHz machines. In the last case, the receivers were those listed Figure 5.1. For each set of receivers we collected statistics on 10 MB and a 40 M transfers, using both memory-to-memory and disk-to-disk tests. In the memory-t memory tests, the sender sent data from memory and each of the receivers receive data into a memory buffer. In the disk-to-disk tests, the sender sent a file which read from disk, and each of the receivers stored the received data to a file on disk. E performing both the memory and the disk tests, we were able to observe the protocol behavior when the applications were always ready, as well as when they were slowe by I/O operations.

5.1.2 Experimental Results

The throughput results for each of the experimental tests are shown in Figure 5.1. Each plot shows the average throughput over five tests for the given kernel buffer size. The resulting throughput ranges from 4.4 Mbps for 64 KB buffers to 8.4 Mbps for 1 MB buffers.



(a) Memory-to-memory, 10MB

(b) Memory-to-memory, 40MB

(c) Disk-to-disk, 10MB

(d) Disk-to-disk, 40MB

Figure 5.1: Throughput of RMC in a local environment.

From the results shown in Figure 5.1, we can make two general observations about RMC in a local environment. The first observation is that the kernel buffer size affects throughput for tests run with both 64 KB and 128 KB kernel buffers, but has a minimal effect on tests where the buffer size was 256 KB or greater. The limit

in throughput for smaller buffers is most likely due to the lack of buffer space at the sender. The sender is required to buffer sent packets for at least 10 round-trip times. Because the Linux clock ticks once every 10 ms and the estimated round-trip time must be greater than zero, the minimum estimated round-trip time is 10 ms. As a result, the minimum buffering time is 100 ms. When the buffer size is less than 256 KB, the sender completely fills its buffer space and must wait for the buffer space to be freed before sending more data. The second observation is that throughput varies only slightly with the number of receivers, especially when there is sufficient buffer space allocated to the protocol. For example, in the disk-to-disk 40 MB test, throughput values are less for 8 receivers than for lesser number of receivers. However, given a sufficient amount of memory, the throughput values for 8 receivers are close to those for lesser number of receivers. This indicates that RMC is scalable up to 8 receivers in a local environment.

To further understand the behavior of RMC, we monitored the NAK and rate request feedback activity that occurred during each test. We report this activity in terms of the total number of NAKs and rate requests that arrive at the sender during each test.

In the memory-to-memory tests for both 10 MB and 40 MB there was no data loss, and receivers' buffers did not fill past the thresholds, so no feedback was generated. Thus, as is evident in Figures 5.1 (a) and (b), the throughput values are nearly identical for these cases. For the disk-to-disk 10 MB test there was also no data loss and therefore no NAK activity. Receivers did, however, utilize enough of their buffer space to generate rate requests. As shown in Figure 5.2, the number of rate requests for this test is proportional to the number of receivers for both the 64 KB and the 128 KB buffer cases. Rate requests for 256 KB buffers and larger are minimal. Comparing against Figures 5.1 (a) and (c), we see that the difference in throughput between the

10 MB disk tests and the 10 MB memory tests (in which there was no feedback) is small, indicating that the rate requests had little effect on the throughput.

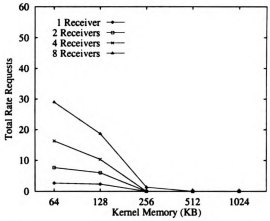
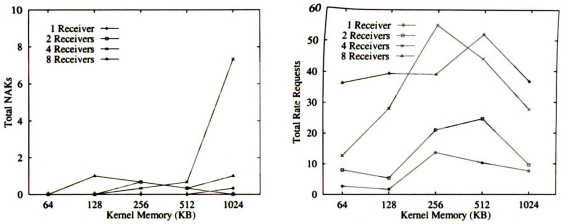


Figure 5.2: Total rate requests for the 10 MB disk test.

Figures 5.3 (a) and (b) show the NAK and rate request feedback statistics for the 40 MB disk-to-disk tests. The number of NAKs for the 40 MB disk-to-disk transfer is less that 2 in most cases. In the one case where a larger number of total NAKs occurred, a large portion of the total is due to NAK retransmissions. In this case, a receiver may have had an incorrect estimate of round-trip time, causing it to retransmit NAKs too early. While the number of NAKs is quite small, the number of rate requests for these tests is noticeable and seemingly unpredictable. Because the protocol is running at kernel-level, it is difficult to assess the cause for this unpredictability. A number of different activities in the operating system or delays in the I/O may have caused the application to slow, allowing the kernel buffers to quickly fill. As shown in Figure 5.1(b) and (d), there is a decrease in throughput from the 10 MB file test to the 40 MB file test. However, in most cases, the decrease in throughput is less than 1 Mbps.

In Figure 5.4 we break down the experimental results by the number of receivers and give the minimum and maximum throughput values for each case. In all cases,



(a) Total NAKs. (b) Total rate requests.

Figure 5.3: Feedback for the 40 MB disk test.

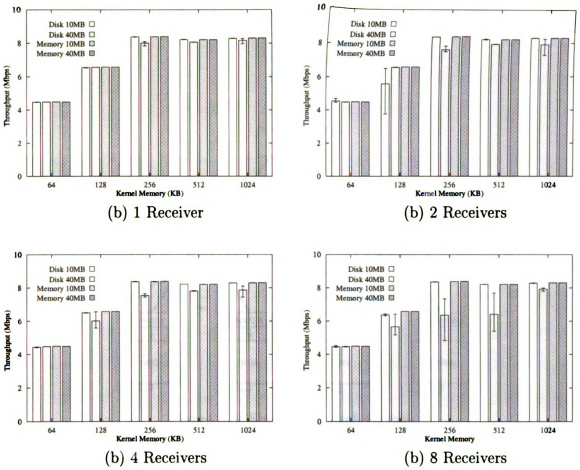
the performance results for the memory test are very consistent. The disk-to-disk tests vary more than the memory tests, with the most variance occurring in the 40 MB disk tests. This variance is again difficult to assess, and may be caused by a number of different operating system or I/O operations.

5.2 Simulation Study

In order to evaluate the performance of RMC in environments other than a local area network, we used the CSIM simulation package to implement a simulated network environment in which we could test the RMC protocol. The simulation was implemented using portions of the data structures from the Linux kernel, so that we could easily plug the RMC implementation into the simulation with only minor changes.

5.2.1 Simulation Environment

The architecture of the simulated environment is shown in Figure 5.5. In this figure, each box represents a CSIM process, of which there are three types: host processes, network interface processes, and router processes. The host processes are used to



(b) 1 Receiver (b) 2 Receivers

(b) 4 Receivers (b) 8 Receivers

Figure 5.4: Average throughput with minimum and maximum throughput values.

simulate each participating host. A host process controls the operation of the RMC protocol and underlying operating system on the host, as well as the sending or receiving application. Each host process is coupled with a network interface process, which handles incoming packets for the host and simulates the network delay associated with each packet. The router processes are used to simulate a simple network topology through which packets are routed before being delivered to the network interfaces. Each router is assigned a network speed, a queue size, and a loss rate.

The network simulation works as follows. At a given host, outgoing packets are constructed with a full RMC header and a partial IP header, and then passed to the closest router. Within a router, the packets are taken from the local queue, assigned

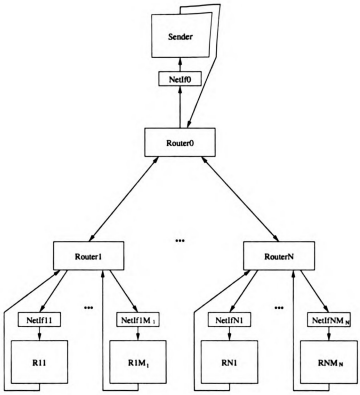
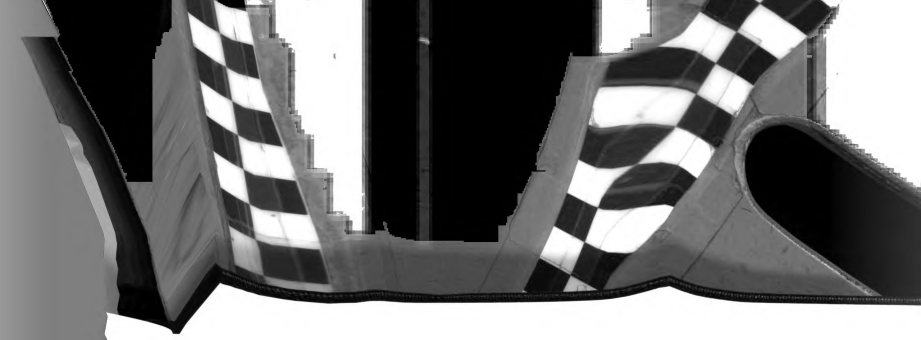


Figure 5.5: Simulation Architecture.

a delay according to the network speed, and passed on to the next router or to the appropriate network interface, as dictated by the IP destination. Multicast packets are duplicated within a router as necessary. At the network interface, packets are received one at a time, held for the assigned delay, and then passed to the host. At the host, incoming packets are passed to the RMC protocol, where normal processing continues. To simulate network loss, each router and network interface may be assigned a loss rate, which it uses to randomly drop packets.

To simulate the protocol, we imported the RMC protocol code directly from the Linux kernel into the simulation. We then brought in portions of data structures and functions from the Linux kernel that were used by the protocol. In order to simulate processing time at a host, we measured the time to complete RMC processing and lower layer network processing in the Linux kernel, and then introduced the



Group	*Delay*	*Loss Rate*
A	2 ms	0.1%
B	20 ms	0.5%
C	100 ms	2.0%

Table 5.2: Characteristic groups.

appropriate delays in the host process. For sending and receiving data of length l, the RMC delay was $(10 + .025 * l)\mu s$ and the lower layer delay was $150\mu s$.

In order to understand how RMC performs when hosts have different capabilities, we used the router processes to divide receivers into characteristic groups, where a characteristic group is defined by its network delay and loss properties. We divided the loss rate between the router process and each receiver's network interface process to simulate both correlated and uncorrelated loss. As reported in [33], most losses take place in the tail links of the network. The network backbone and the individual sites are mostly loss free. Thus, we set the loss parameters such that 90% of the loss was correlated and occurred at the router process and 10% of the loss was uncorrelated and occurred at the network interface process. Definitions of the characteristics groups that we used are listed in Table 5.2. Group A is intended to simulated a local environment, group C is intended to simulate a wide area environment, and group B simulates something between the two. These figures are similar to those reported in [34].

5.2.2 Simulation Results for "Fast" Receivers

We conducted two sets of simulation tests for 10, 25, and 100 receivers. In our first set of tests we assumed that the receivers had "fast" 300 MHz processors similar to those in the experimental tests and that the connecting network was 10 Mbps Ethernet. For this set of tests we ran five different tests using different combinations of the three

Test	*10 Receivers*	*25 Receivers*	*100 Receivers*
1	All in A	All in A	All in A
2	All in B	All in B	All in B
3	All in C	All in C	All in C
4	8 in B, 2 in C	20 in B, 5 in C	90 in B, 10 in C
5	2 in B, 8 in C	5 in B, 20 in C	10 in B, 90 in C

Table 5.3: Simulation tests.

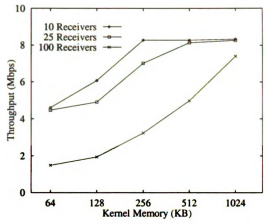
characteristic groups. As shown in Table 5.3, for tests 1, 2, and 3, the receiver group consisted of the single characteristic groups A, B and C, while tests 4 and 5 used receivers from characteristic groups B and C in opposite proportions.

Figure 5.6 shows the throughput results for tests 1, 2, and 3. In these tests we see that kernel buffer size becomes an important factor in determining throughput. Because the network delay is larger than in a local area network, the sender is required to buffer data for a longer time. This means that once the send window is full, the sender must wait for longer periods of time before buffer space is freed. An increase in the number of resent packets further slows packet release since packets must be held for the minimum time from the last time they are sent. We also observe a definite decrease in throughput from 10 to 25 receivers and from 25 to 100 receivers. This decrease is most likely due to the NAK activity. Unlike the experimental study, in the simulation study we introduce a constant network loss. Because we do not use global NAK suppression, the number of NAKs that arrive at the sender increases linearly with the number of receivers and with the loss rate. The sender does a slow start on the receipt of each NAK, which greatly affects the throughput, especially with high loss rates.

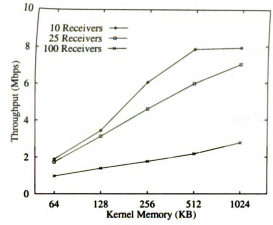
Despite the high loss rates, RMC is still able to achieve over 6 Mbps for up to 100 receivers in group A and up to 25 receivers in group B. In addition, the efficiency that is gained by using reliable multicast over reliable unicast is large, even when the throughput is low.

10

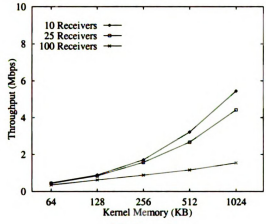
Throughput (Mbps)



(a) Test 1



(b) Test 2

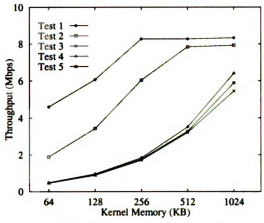


(c) Test 3

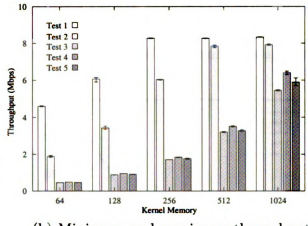
Figure 5.6: Simulation Throughput.

In Figures 5.7, 5.8, and 5.9, we present the throughput results for all 5 test cases for 10, 25, and 100 receivers, respectively. In each figure, the (a) subfigure gives the average throughput results and the (b) subfigure shows the minimum and maximum throughputs along with the average. In addition, the (c) subfigure shows the total number of NAKs that arrive at the sender and the (d) subfigure shows the total number of rate requests that are generated. In each test the throughput generally increases with kernel buffer size. Further, for each receiver group size, the best performance occurs in test 1, which represents a local environment. Test 1 is followed in throughput by test 2, which simulates a metropolitan area network. Test 3, which is

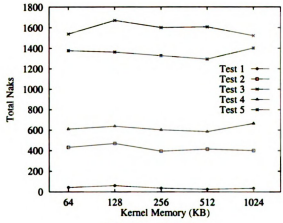
the wide area network, yields the lowest throughput. For test 4, where the receiver group is primarily composed of medium area receivers, and test 5, where the receiver group is primarily composed of wide area receivers, the throughput values are close to that of the wide area environment. This shows that RMC adapts to the least capable receivers in the multicast group.



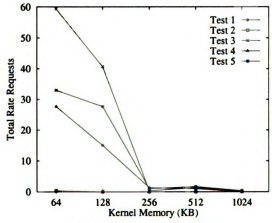
(a) Average throughput.



(b) Minimum and maximum throughput.



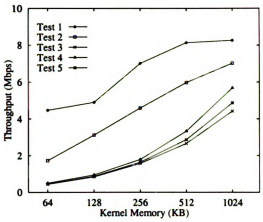
(c) Total NAKs.



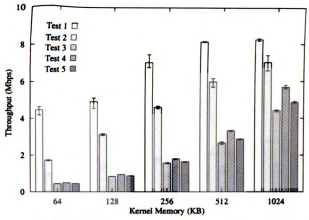
(d) Total rate requests.

Figure 5.7: Results for 10 fast receivers.

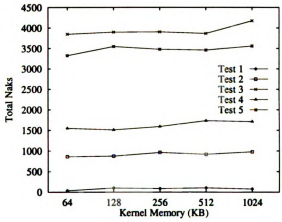
Looking at subfigures (c) and (d) from Figures 5.7, 5.8, and 5.9, we can analyze the feedback behavior from this set of simulation tests. We see that the number of NAKs remains constant across different buffer sizes, and is proportional to the number of receivers and loss rate of the test. The number of rate requests, on the other hand, is



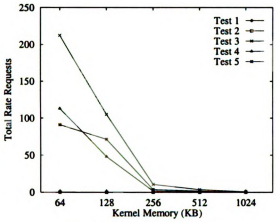
(a) Average throughput.



(b) Minimum and maximum throughput.



(c) Total NAKs.



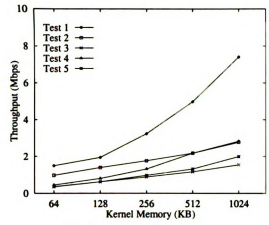
(d) Total rate requests.

Figure 5.8: Results for 25 fast receivers.

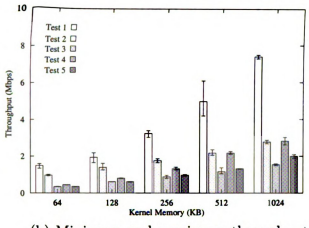
large for 64 KB and 128 KB buffers, but is small or insignificant for buffers of size 256 KB and larger. In these tests NAKs most likely had the effect of lowering an entire curve, while rate requests lessened throughput only for small buffers. In all cases, the number of NAKs is much larger than the number of rate requests, suggesting that the overall effect of rate requests was minimal.

5.2.3 Simulation Results for "Slow" Receivers

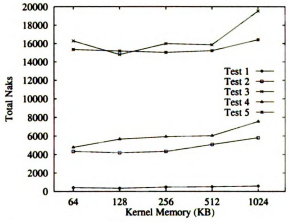
In the second set of simulation tests, we introduced "slow" 75 MHz processors while still using a 10 Mbps Ethernet network. Slow processors were defined to be two



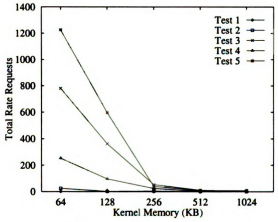
(a) Average throughput.



(b) Minimum and maximum throughput.



(c) Total NAKs.



(d) Total rate requests.

Figure 5.9: Results for 100 fast receivers.

generations old and were four times as slow as the fast processors used in the previous set of simulation tests. For this set of tests, we focused on characteristic groups B and C, making a subset of the processors slow. We performed three different tests for groups of 10, 25, and 100 receivers. The three tests are summarized in Table 5.4. Test 6 is similar to test 2 in the previous set of tests, in that all the processors belong to group B. In test 2 all of the processors are fast, while in test 6 all of the processors are slow. In test 7, the processors are all still in group B, but 8 of them are fast and only 2 are slow. Test 8 is similar to test 4, where 8 processors belong to group B and 2 processors belong to group C. In test 8, however, the group B processors are fast

Test	*10 Receivers*	*25 Receivers*	*100 Receivers*
6	All in B, all slow	All in B, all slow	All in B, all slow
7	All in B 8 fast and 2 slow	All in B 20 fast and 5 slow	All in B 90 fast and 10 slow
8	8 fast in B 2 slow in C	20 fast in B 5 slow in C	90 fast in B 10 slow in C

Table 5.4: Simulation tests.

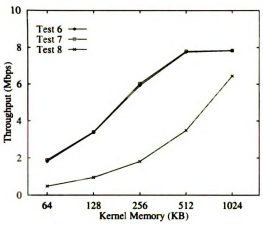
and the 2 group C processors are slow.

The test results for 10, 25 and 100 receivers are shown in Figures 5.10, 5.11, and 5.12, respectively. In each figure, the (a) and (b) subfigures show the throughput, the (c) subfigure shows the NAK feedback, and the (d) subfigure shows the rate request feedback.

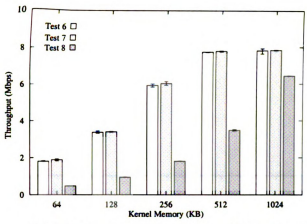
For the 10 receiver case as shown in Figure 5.10, the results for tests 6 and 7, where all processors are from group B, are very close. Further, the results for tests 6 and 7 are similar to those for test 2, shown in Figure 5.7. We see the same similarities between tests 8 and 4, which mix the receivers between groups B and C. These results suggest that simply introducing slower processors into the group do not significantly affect the protocol performance.

The same observations that were made for the 10 receiver case can be made for the 25 and 100 receiver cases. As shown in Figure 5.11(a), the difference in the results between tests 6 and 7 for the 25 receiver cases is slightly more, but still not significant. Similarly, the feedback activity for tests 6 and 7 is similar to the feedback activity for test 2 and the feedback activity for test 8 is similar to the feedback for test 4. Again, introducing slow receivers has little effect on the protocol.

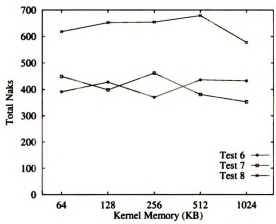
We see a larger effect of introducing slow processors in the 100 receiver tests, shown in Figure 5.12. While the throughput is limited by the high packet loss, there is more of a difference in the throughput values between tests 6 and 7 than in the 10 and 25 receiver case. In addition, as shown in Figure 5.12(d), the number of rate



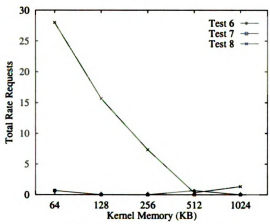
(a) Average throughput.



(b) Minimum and maximum throughput.



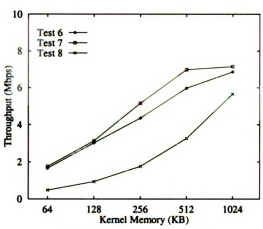
(c) Total NAKs.



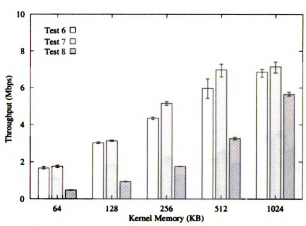
(d) Total rate requests.

Figure 5.10: Results for 10 slow receivers.

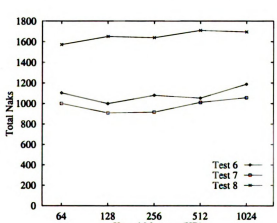
requests that are being generated in test 6 is much larger than the number being generated in test 2, shown in Figure 5.9(d). Thus, the effect of slow processors is clearly more when the number of slow receivers is greater. In addition, in test 6 with 128 KB of memory, there is a significantly larger number of rate requests than for the other cases. It is difficult to determine the cause of the high number of rate requests in this case. For the case, the capabilities of the receivers may have been somehow negatively affected by the increase in buffer space at both the sender and at the receivers, causing a strange surge in the number of rate requests.



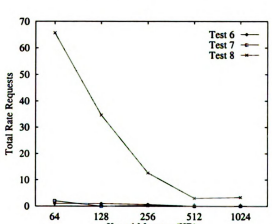
(a) Average throughput.



(b) Minimum and maximum throughput.

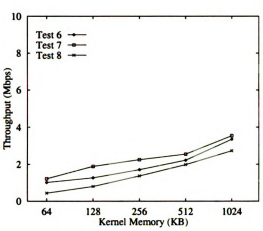


(c) Total NAKs.

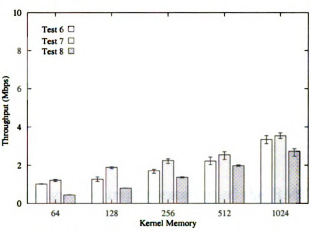


(d) Total rate requests.

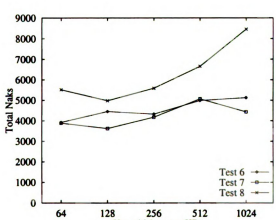
Figure 5.11: Results for 25 slow receivers.



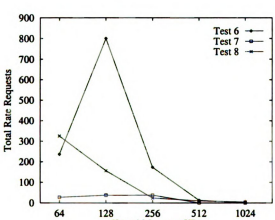
(a) Average throughput.



(b) Minimum and maximum throughput.



(c) Total NAKs.



(d) Total rate requests.

Figure 5.12: Results for 100 slow receivers.

Chapter 6

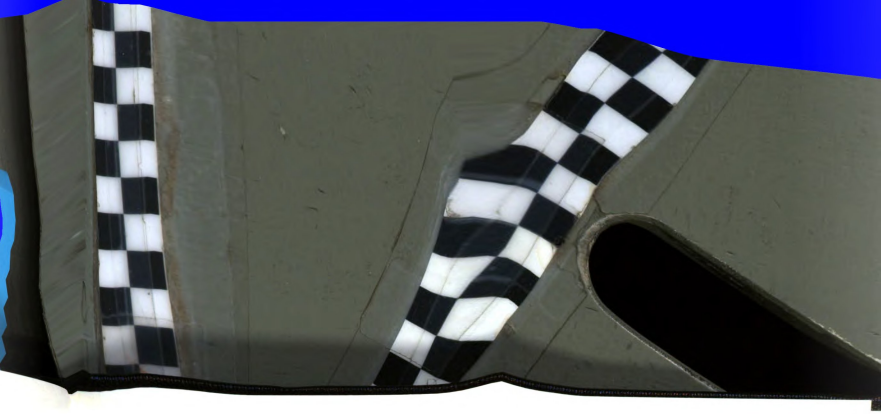
Conclusions and Future Work

In this research project we explored the design and implementation of the RMC reliable multicast protocol. The goal of the project was to design a protocol that is both scalable and fits well into an operating system kernel. We considered three issues in the RMC design: reliability, connection management, and flow control. In each aspect of the design, the interaction between the receivers and the sender is minimized. RMC uses NAK-based reliability and supports anonymous group membership. To allow for limited buffer space in combination with pure NAK-based reliability, RMC uses a combination of window-based and rate-base flow control. The window-based control manages the finite buffer space, while the rate-based control manages how quickly data within the window is transmitted. We have implemented the RMC protocol in the Linux kernel network stack. The implementation is based on IP multicast and provides a BSD socket interface to user-level applications.

We studied the performance of the RMC protocol through both experimentation and simulation. Performance tests in the Linux kernel revealed that RMC performs well for up to 8 receivers in a local area network, where packet loss is minimal. We used a network simulation to test the performance of RMC in both local area and wide area

network environments for up to 100 receivers. Results of the simulation tests show that RMC is scalable up to 100 receivers in a local area network environment, where loss rates and network delay are low. Although throughputs are lower in a wide area network environment with higher loss rates, the RMC protocol still provides good performance and is able to accommodate a large number of NAKs, even without global suppression. Moreover, the efficiency that is gained by using reliable multicast over reliable unicast is very large.

There are several aspects of RMC that may be improved. First, we may want to design a mechanism by which the sender could estimate the available network speed across the connection. The sender could then use this estimate to as an absolute limit on the transmission rate. Secondly, the protocol could be better adapted to fairly handle large variations in the capabilities of the receiver group. Currently, RMC adapts the transmission to the least capable receiver. To make the transmission more fair for all receivers, the sender could classify the receiver group according to their capabilities, and then transmit the data at different rates for each group. Other improvements for RMC include adding global NAK suppression to make the protocol more scalable in networks with higher loss rates and adding an option for maintaining explicit group membership.



Bibliography

[1] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of SIGCOMM '95*, August 1995.

[2] Peter Danzig. Flow Control for Limited Buffer Multicast. *IEEE Transactions on Software Engineering*, 20(1):1–12, January 1994.

[3] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64, February 1995.

[4] Gregory Pfister. *In Search of Clusters.* Prentice Hall, 2nd edition, 1997.

[5] Andrew S. Tanenbaum. Experiences with the Amoeba Distributed Operating System. *Communictions of the ACM*, 33:46–63, December 1993.

[6] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, December 1996.

[7] Robel Barrios and P. K. McKinley. WebClass: A multimedia web-based instructional tool. Technical Report MSU-CPS-98-26, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, August 1998.

[8] I. M. I. Habbab, M. Kavehrad, and C. W. Sundberg. Protocols for very high-speed optical fiber local area networks using a passive star topology. *Journal of Lightwave Technology*, LT-5(12):1782–1794, December 1987.

[9] M. Loper. Distributed interactive simulation (DIS) requirement for multicast. Multipeer/Multicast Workshop, Compendium of Solicited Papers, sponsored by U.S. Army Program Manager for Trai ning Devices and the Institute for Advanced Simulatio n and Training, August 1991. Draft presented at Orlando, Florida.

[10] Donald P. Brutzman, Michael R. Macedonia, and Michael J. Zyda. Internetwork infrastructure requirements for virtual environments,. In *Annual Symposium of the Virtual Reality Modeling Language (VRML 95)*, December 1995.

[11] IEEE. 802.3: Carrier sens multiple access with collision detection, 1985.

[12] Vicki Johnson and Marjory Johnson. Introduction to IP Multicast Routing. IP Multicast Initiative White Paper, 1997. Available at http://www.ipmulticast.com/community/whitepapers/introrouting.html.

[13] S. Deering, C. Partridge, and D. Waitzman. Distance Vector Multicast Routing Protocol. Internet RFC 1075, November 1988.

[14] J. Moy. Multicast Extensions to OSPF. Internet RFC 1584, March 1994.

[15] Steven Deering et. al. Protocol independent multicast version2, dense mode specification. IETF Draft, May 1997.

[16] T. Ballardie. Core Based Trees (CBT) Multicast Routing Architecture. Internet RFC 2201, September 1997.

[17] S. Deering, D. Estrin, D. Farinacci, A. Helmy, D. Thaler, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. Internet RFC 2362, June 1998.

[18] Michael R. Macedonia and Donald P. Brutzman. MBone provides audio and video across the Internet. *IEEE Computer*, April 1994.

[19] B. Fenner. Internet Group Management Protocol, Version 2. Internet RFC 2236, November 1997.

[20] W. T. Strayer, editor. *Xpress Transport Protocol Specification, Revision 4.0.* XTP Forum, March 1995.

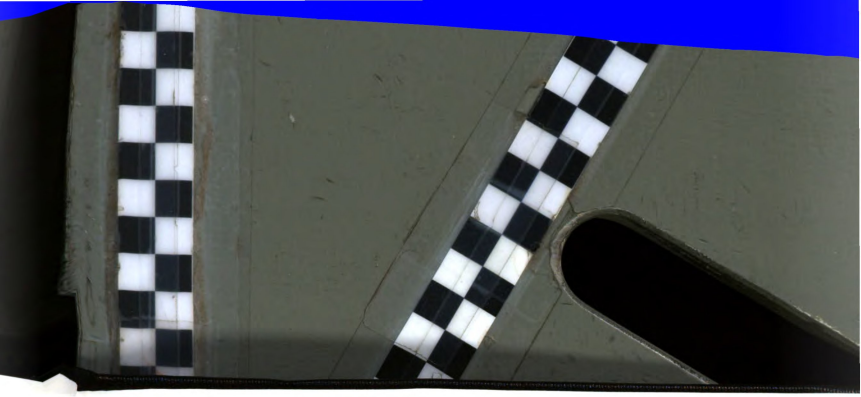
[21] Rajesh Talpade and Mostafa H. Ammar. Single Connection Emulation (SCE): An Architecture for Providing a Reliable Multicast Transport Service. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, June 1995.

[22] Alex Koifman and Stephen Zabele. RAMP: A Reliable Adaptive Multicast Protocol. In *Proceedings of IEEE INFOCOM*, pages 1442–1451, March 1996.

[23] Sanjoy Paul, Krishan K. Sabnani, John C. Lin, and Supratik Bhattacharyya. Reliable Multicast Transport Protocol RMTP. *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[24] Rajendra Yavatkar, James Griffioen, and Madhu Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *Proceedings of the ACM Multimedia '95 Conference*, November 1995.

[25] Tetsuo Sano Teruji Shiroshita, Osamu Takahashi, and Masahide Yamashita. Monitoring-based Flow Control for Reliable Multicast Protocols and its Evaluation. In *Proceedings of IEEE International Performance, Computing, and Communication Conference*, pages 403–409, February 1997.



[26] Brian N. Levine and J.J. Garcia-Luna-Aceves. A Comparison of Reliable Multicast Protocols. *accepted to appear in ACM Multimedia Systems Journal*, 1998.

[27] University of Southern California Information Sciences Institute. Transmission Control Protocol. Internet RFC 793, September 1981.

[28] P. Karn and C. Partridge. Estimating Round-Trip Times in Reliable Transport Protocols. In *Proceedings of SIGCOMM '87*, August 1987.

[29] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. In *Proceedings of SIGCOMM '88*, August 1988.

[30] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, 1996.

[31] David A. Rusling. *The Linux Kernel*. 1997. Available at ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/tlk-0.8-2.ps.gz.

[32] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP*, volume III. Prentice Hall, Englewood Cliffs, NJ, 1993.

[33] Don Towsley, Jim Kurose, and Sridhar Pingali. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. *IEEE Journal on Selected Areas in Communication*, 15(3):398–406, April 1997.

[34] Michael S. Borella and Gregory B. Brewster. Measurement and Analysis of Long-Range Dependent Behavior of Internet Packet Delay. In *IEEE INFOCOM 1998*, pages 497–504, April 1998.