Master's Thesis

Title

Design and Implementation of the Flexible Multicast Protocol

Supervisor

Professor TOKORO, Mario
Associate Professor AMANO, Hideharu

Computer Science,

Graduate School of Science and Technology, Keio University

1997 March

89624670

LAOHAKULVATHIT, Pornchai

Summary of M. S. Thesis

Major	Student No.	SURNAME, Firstname
Computer Science	89624670	LAOHAKULVATHIT, Pornchai

Title

Design and Implementation of the Flexible Multicast Protocol

Group Communication is becoming more and more popular in global networks. New applications and protocols are under development to support cooperative work and information dissemination in world wide, distributed systems. Most of them are based on a communication form in which a single source transmits data to a group of multiple receivers (1-to-n).

Multicasting provides an efficient way of disseminating data from a sender to a group of receivers. Instead of sending a separate copy of the data to each individual receiver, the sender just sends a single copy to all the receivers. The technique used for distributing the data to individual receivers is transparent to the sender.

The multicast support in TCP/IP protocol suite is becoming an increasingly important technology area for those group-based applications. The underlying delivery mechanism for IP multicast is presently the User Datagram Protocol (UDP) or raw IP packets. At present, these mechanisms provide a "best-effort" delivery service. Best effort implies that IP packets are treated with equal weight, and packets may be occasionally be delayed lost, duplicated, or delivered out of order. An increasing variety of group communication applications are being developed in which a consistent and/or reliable data delivery of all or a subset of data packets is a critical performance factor.

In this study, we describes the design and implementation of a multicast transport protocol called FMP. FMP provides the application specific reliability of multicast data transfer from one sender to a group of receivers. FMP achieves reliability by using a retransmission strategy that combines aspects of the traditional techniques go-back-n and selective-retransmit. To avoiding the feedback implosion problem, receivers send feedback to group periodically based on timer, and not only the sender but the other receivers that have the lost packets can retransmit that lost packets. This implementation of FMP has been shown to provide high throughput performance on Local Area Networks (LAN). For two or more destinations a single LAN, FMP provides higher throughput than any other protocol that does not use multicast or broadcast technology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

As memory, storage, reliability, and bandwidth increase, decentralization of data storage increases. So the motivation for developing multicast is that there are applications that want to send a packet to more than one destination host.

Multicasting provides an efficient way of dissemination data from a sender to a group of receivers. Instead of sending a separate copy of the data to each individual receiver, the sender just sends a single copy to all the receivers. The technique used for distributing the data to individual receivers is transparent to the sender. In fact, there is an underlying *multicast delivery system* which forwards the data from the sender to every receiver. Conceptually, the multicast delivery system forms a multicast tree connecting the sender and the receivers, with the sender as the root node and the receivers as the leaf nodes (see figure 1.1). Data generated by the sender flows through the multicast tree, traversing each tree edge exactly once. When receivers join or leave a multicast group, the multicast tree is dynamically reconfigured.

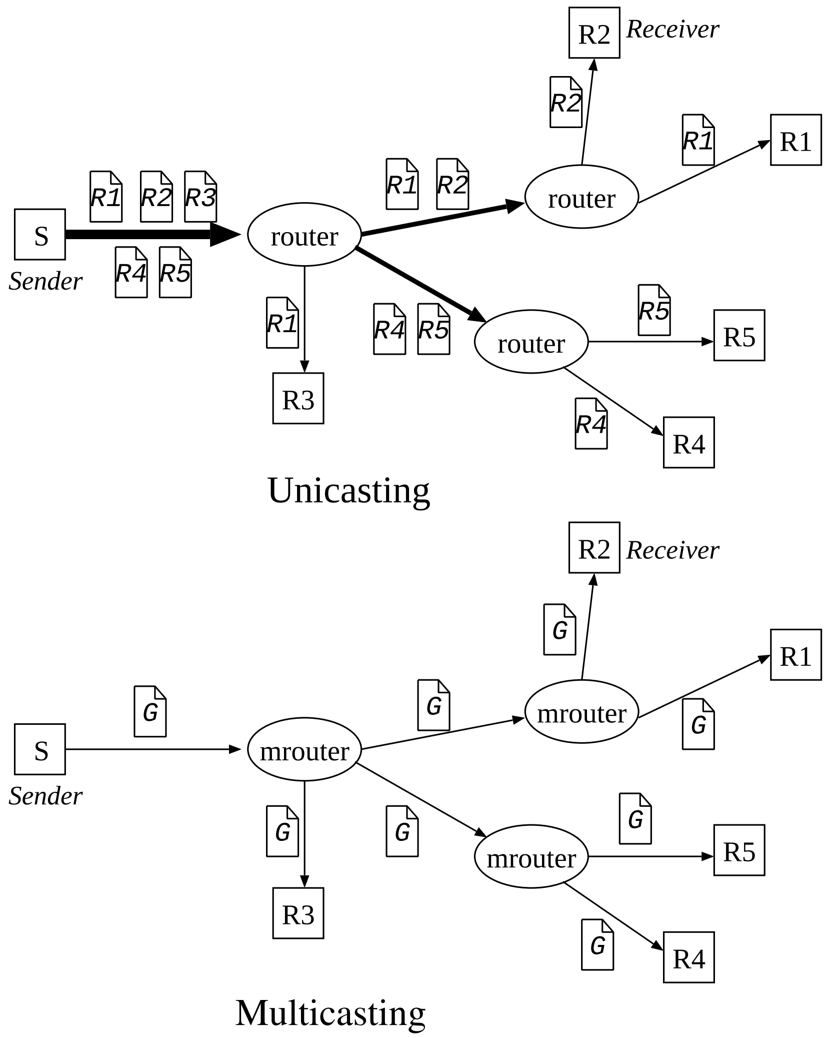

Figure 1.1: Comparison of point-to-point unicast and multicast data flow

Multicasting of real-time multimedia information has recently been receiving a great deal of attention [17]. Also, researchers have demonstrated multicasting real-time data over the Internet using a multicast delivery system called MBone [28, 14]. Since most real-time applications can tolerate some data loss but cannot tolerate the delay associated with retransmissions, they either accept some loss of data or use forward error correction for minimizing such loss. The main objective of multicast protocols for transporting real-time data is to guarantee quality of service by bounding end-to-end delay at the cost of reliability. In contrast, the objective of our protocol is to guarantee *application specific reliability* at the expense of delay.

This introduces new and complex issues that must be dealt with. Reliability, resiliency, ordering of messages, and fault-tolerance of communications are all low-level concerns of application developers that are introduced by the decentralization of computation into distributed systems. There are several multicast applications which require sequenced and lossless delivery of data. Such applications cannot tolerate data loss, but can tolerate delays due to retransmissions. For example, a publisher distributing books/journals electronically to its subscribers, or a financial institution disseminating market data to its subscribers. For such applications, a separate TCP [10] connection can be established between the sender and each receiver, delivering every packet individually to each receiver. However, such a scheme is very inefficient because every packet needs to be duplicated as many times as the number of receivers. For example, in figure 1.1, when the sender uses TCP to transmit a packet to each receiver, the same packet traverses path between the source and the first router fives times, each packet's destination is for each receivers. However, when the sender *multicasts* a packet to all receivers, the packet traverses every path only once to destination group address.

The lack of reliable broadcast, or reliable multicast, primitives has introduced a host of unnecessarily complex, and inefficient schemes for providing application specific fault-tolerance, reliability, and resiliency. Although addressing the difficult issues, usually these schemes are not reusable in other applications because the closely knit connection between the application and its communication mechanisms. In many cases these are so closely interconnected that one is not distinguishable from the other.

This virtually eliminates interoperability and can seriously effect application evolution. Distributed application should have a robust transport primitive that provides reliable delivery, ordering of messages, and fault-tolerance, thus unburdening the application developer and allowing concentration on the application development itself. The Flexible Multicast Protocol (FMP) is such a primitive. FMP provides a application specific reliable degree on top of an unreliable multicast service.

1.2 Research Objectives

The research objectives of this investigation are:

- Co-design the Flexible Multicast Protocol.
- Implement the design of the protocol and experiment with features to provide a continually evolving and comprehensive protocol.

1.3 Organization of the Thesis

Chapter 2 discusses some background concepts and provides a basic introduction to IP multicast and MBone technology, and discusses retated works. Chapter 3 presents the protocol description, design, and a representation as a state transition diagram. Chapter 4 presents an overview of the implementation of the protocol and the issues and lessons learned. Chapter

5 presents some performance results obtained by the implementation. Finally, Chapter 6 discusses some conclusions about the protocol design, and implementation. In addition, Appendix A presents the packet formats of the protocol, and Appendix B lists the header files in this implementation.

Chapter 2

Background and Related Works

This chapter presents a background to IP Multicast technology and to other related works.

2.1 IP Multicasting

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

IP multicasting provides two services for an application: [25]

1. Delivery to multiple destinations. There are many applications that deliver information to multiple recipients: interactive conferencing and dissemination of mail or news to multiple recipients, for example. Without multicasting these types of services tend to use TCP today (delivering a separate copy to each destination). Even with multicasting, some of these applications might continue to use TCP for its reliability.

2. Solicitation of servers by clients. A diskless workstation, for example, needs to locate a bootstrap server. Today this is provided using a broadcast (BOOTP), but a multicast solution would impose less overhead on the hosts that don't provide the service.

When the destination subset is less than the entire set of all possible destinations, some addressing scheme must be used to represent the subnet, called the multicast group. When a host desires to receive multicast traffic destined for a certain multicast group, the host should, in effect, join the multicast address. This addressing is an Internet standard [22].

An IP multicast address is a Class D address (see figure 2.1), and is always use as destination address. A *multicast group address* is the combination of the high-order 4 bits of 1110 and the multicast group ID. The valid range for IP multicast addresses expressed in dotted decimal notation is 224.0.0.0 to 239.255.255.255.

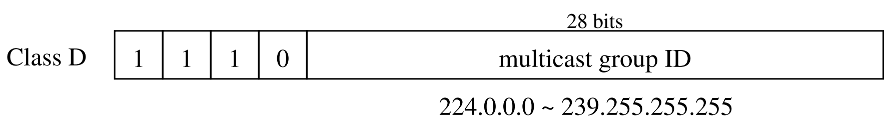

Figure 2.1: Format of a class D IP address

The set of hosts listening to a particular IP multicast address is called a *host group*. A host group can span multiple networks. Membership in a host group is dynamic – hosts may join and leave host groups at will. There is no restriction on the number of hosts in a group, and a host does not have to belong to a group to send a message to that group.

Some multicast group addresses are assigned as well-known addresses by the IANA (Internet Assigned Numbers Authority). These are called *permanent host groups*. This is similar to the well-known TCP and UDP port numbers. Similarly, these well-known multicast addresses are listed in the latest Assigned Numbers RFC. Notice that it is the multicast address of the group that is permanent, not the membership of the group.

For example, the 224.0.0.0 address is reserved and can not be used by any multicast group. In addition, address 224.0.0.1 means "all systems on this subnet," and 224.0.0.2 means "all routers on this subnet." The multicast addresses between 224.0.0.2 to 224.0.0.255 inclusive are reserved for routing protocols and low-level topology protocols. This leaves the address set of 224.0.1.0 to 239.255.255.255 opens for general use.

The IP multicast interface is currently supported at the TCP/IP layer of hosts that have been configured to support IP multicast technology[1]. This layer provides an address and a protocol port. The address is specified as an IP multicast address and the port is an abstraction that allows multiple destinations within a given host to be distinguished. The underlying transport layer for IP multicast is either the User Datagram Protocol (UDP) [11] or the raw IP datagram protocol. Both of these protocols employ "best-effort" on transmission. That is to say that the datagram is not guaranteed to arrive intact at the destinations, and that no ordering guarantees are given relative to other datagrams. UDP however does incorporate a checksum to detect damaged packets. IP datagrams do not incorporate this checksum. Immediately the need for a reliable protocol should be apparent. With an unreliable and unordered base transport mechanism forming the bottom layer, some sort of protocol is needed to provide reliability to those applications that demand it.

2.1.1 Multicast Routing

The IP multicast model can be expanded to encompass extended LANs and internet works by modifying the routers that connect the LANs together. Normal IP routers do not support multicasting. The multicast traffic does not leave the local network via the IP router unless the router supports IP multicasting. Commercial router hardware is quickly upgrading to support IP multicasting. As an ad hoc solution, many network providers are using "tunneling" to provide routing of multicast traffic. Tunneling is a scheme that involves encapsulating an IP multicast packet inside a regular IP unicast packet and sending the packet to another multicast capable router. This is supposed to be done by a dedicated host on the local network or subnet. However, currently most multicast routing is being done by hosts that serve other purposes in addition to performing multicast routing. A multicast router, or *mrouter*, may have serveral tunnels pointing to various other networks or subnets. When a destination mrouter receives a multicast packet, it will strip the encapsulation information from the packet and multicast the packet out onto the local network. The destination mrouter may also forward the packet along its own tunnels, thereby extending the range of the packet to other networks. (see figure 2.2)

Multicast routers use one of two routing protocols for routing multicast packets to other mrouters. The first is the Distance Vector Multicast Routing Protocol (DVMRP). This protocol is the one implemented by the majority of mrouters at this time. Its operation is detailed in an Internet standard [4]. This protocol maintains topological information by means of a routing protocol based on the Routing Information Protocol (RIP) [2], but enhances the protocol by implementing a multicast forwarding algorithm.

The second multicast routing protocol, MOSPF [8], is being used in a commercial router that supports IP multicasting. MOSPF is the multicast extension to the Open Shortest Path First (OSPF) routing protocol [9]. This routing protocol does not need to use tunnels and does not send more than one copy over a link.

[1]Current IP Multicast Implementations:

- shipping product
  Sun(Solaris 2.x), SGI(IRIX), x86(Microsoft Windows95, BSDI, NetBSD, FreeBSD, Linux), Apple(Open Transport), Proteon, Alantec, Cisco, Bay, 3Com
- patches available
  Sun(SunOS 4.x), DEC(Ultrix, OSF-1), HP(HP-UX), IBM(AIX)

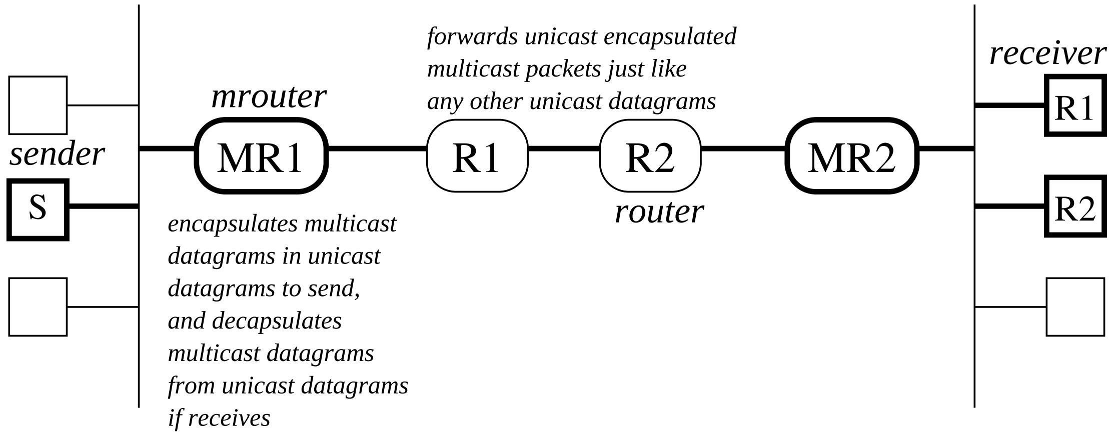

Figure 2.2: Tunneling

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

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

The pruning approach is currently being tested on a limited scale in some DVMRP mrouters (see figure 2.3). In this way only branches of the routing tree that have members that are part of multicast address receive the multicast packets for that group. Other mrouters, except MOSPF, only trim leaves from the routing tree and allow all traffic to be forwarded down branches that have no interested hosts. Thus some other mechanism is needed to restrict the scope of a multicast. This is done through placing lifetime restrictions on packets and lifetime thresholds on each of an mrouters tunnels. The basic premise is that only packets with a Time-To-Live (TTL) greater than 1 should be able to leave the local network or subnet in which they were created. The mrouter on that local network also decides, based on the TTL of the packet and the threshold value of its tunnels, whether the packet is forwarded out through any of its tunnels. Metrics associated with each tunnel decrement the TTL of the packet as it is transmitted across. Higher TTL packets therefore have a much larger scope than smaller TTL packets because the TTL can stay larger than most threshold values as the packet travels among subnets. The Internet Engineering Task Force (IETF) has imposed these suggested TTL conventions on scope:

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

Table 2.1: TTL Thereshold Scope

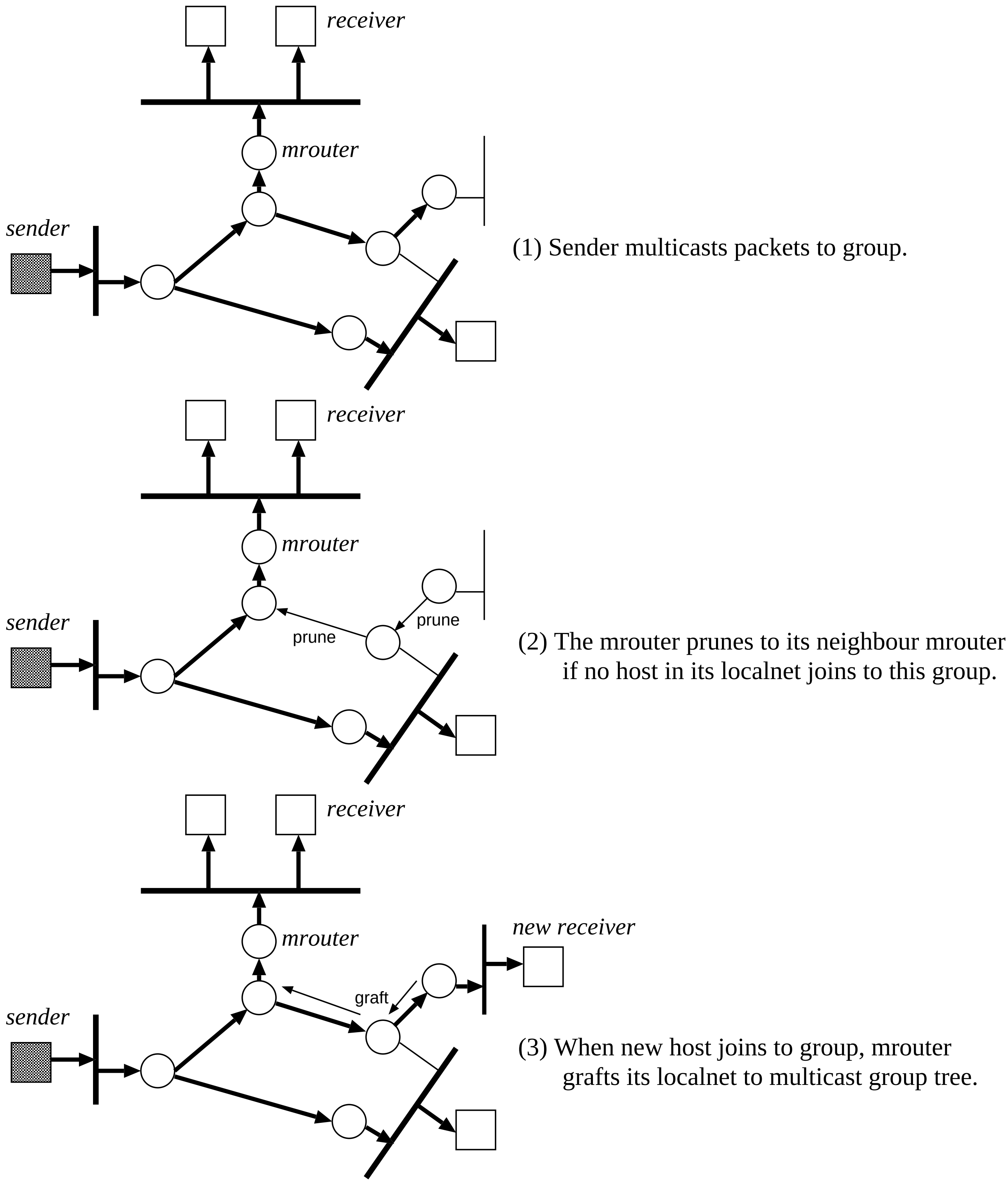

Figure 2.3: Multicast Routing — DVMRP

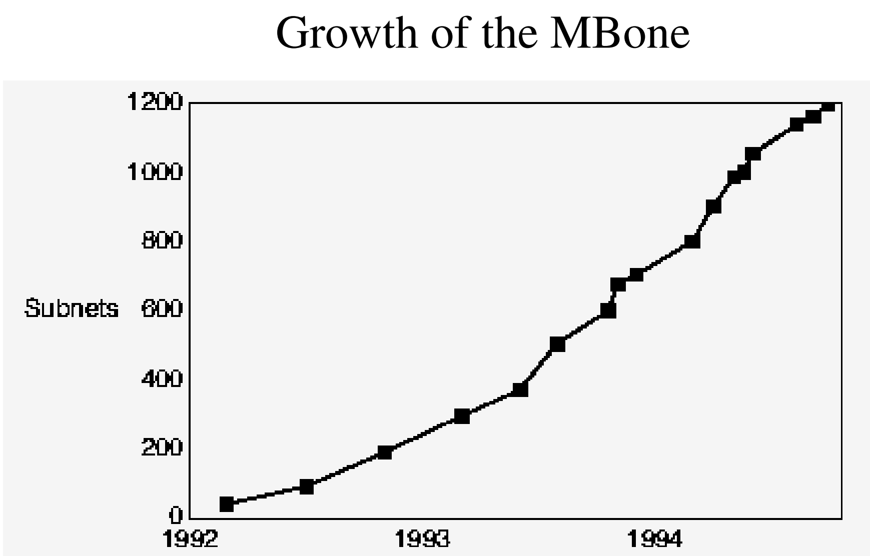

Figure 2.4: The growth curve of the MBone

2.1.2 Internet Multicast Backbone

The virtual network of interconnected multicast routers forms the topology of one of the Internets fastest growing resources, the Internet Multicast Backbone or MBone[28, 14, 6]. The MBone is currently of global proportions with more than 20 countries supporting connections. (see figure 2.4 and 2.5)

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

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

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

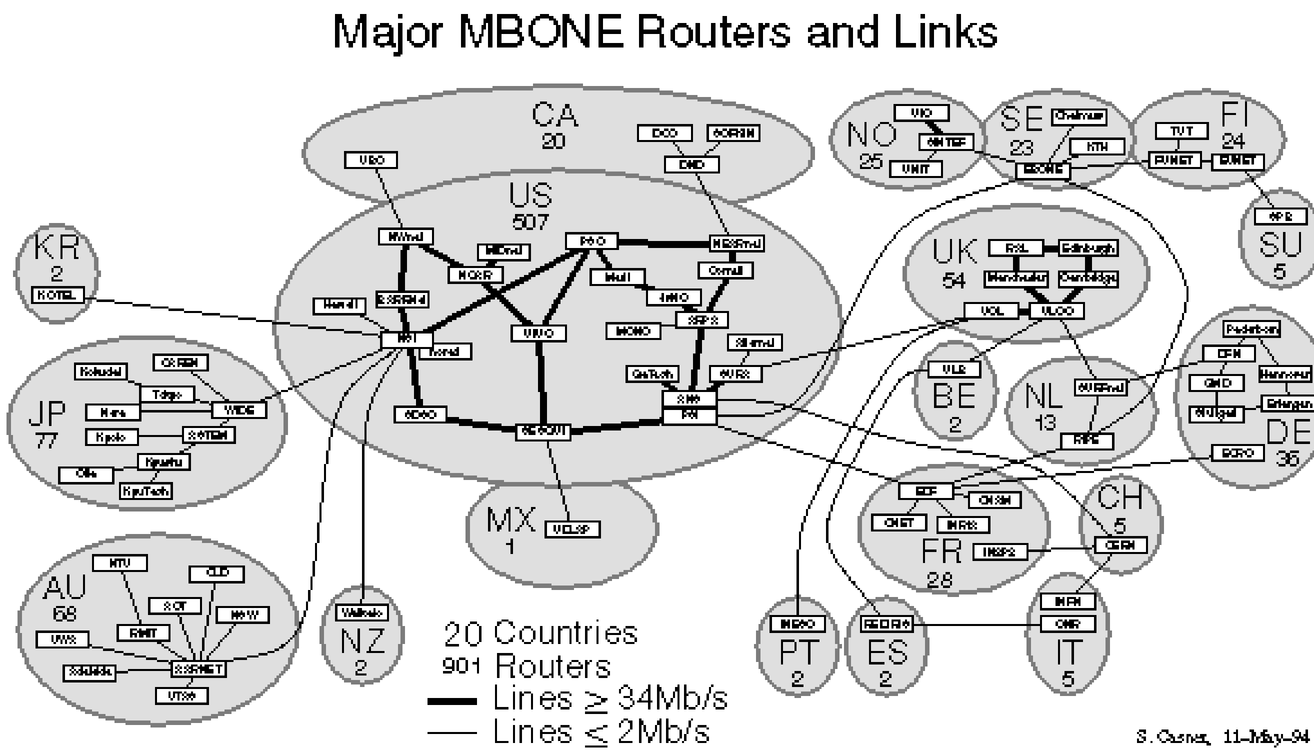

Figure 2.5: The Topology of the MBone

2.2 Provide Reliability to Multicast Communication

Traditionally, two schemes are primarily used to provide reliability on top of unreliable communications media. *sender-initiated* approach, and *receiver-initiated* approach.

2.2.1 Sender-Initiated Approach (ACK)

The sender-initiated approach places the responsibility for providing reliable multicast on the sender, which maintains state information on all receivers to which it is multicasting. This is accomplished by having receivers return positive acknowledgments (ACKs) for every correctly received packet, and having the sender use timers to detect potential packet losses. As the number of receivers becomes very large, the multicast sender is overwhelmed with return messages of its receivers. This fact is known as *sender implosion*[1].(see figure 2.6) The effect of implosion is twofold. Firstly, the large number of return messages results in processing overhead at the sender and, therefore, delays data communications. Secondly, a tremendous amount of control units may cause an excess of both, bandwidth and bufferspace, which in turn causes additional message losses.

2.2.2 Receiver-Initiated Approach (NACK)

The receiver-initiated approach, shifts most of the responsibility for reliable data delivery to the receivers. Packets are stamped sequence numbers and requesting retransmission of the packets corresponding to the gaps. This approach is not suitable to provide a full-reliable communica-

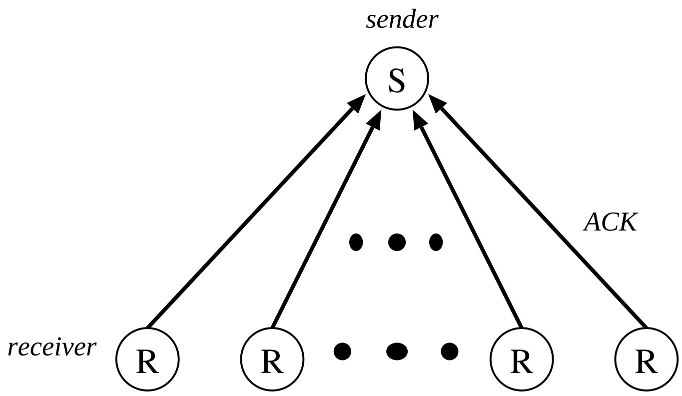

Figure 2.6: A basic diagram of a sender-initiated approach (ACK)

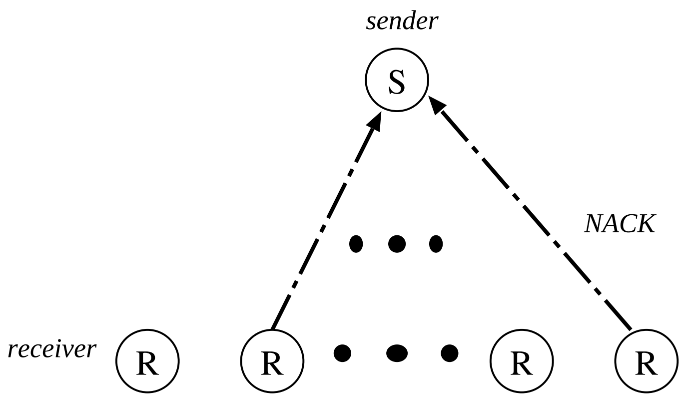

Figure 2.7: A basic diagram of a receiver-initiated approach (NACK)

tion service, because the information that a packet has been received is never propagated back to the sender, the senders in these protocols do not ever know for certain that a destination has received a packet. In addition, a lost packet will not be detected until another packet is received successfully, which may take a long time if the packet is the last to be sent to a group for a while. Furthermore, flow or rate control always require some kind of status exchange between sender and receivers. This will lead to sender implosion even in the case of receiver-initiated approach. (see figure 2.7)

2.3 Related Works

Several error correction schemes have been proposed to improve reliability of multicast communication in the Internet [26]. Much of the existing works on reliable multicast concern the design and implementation of sender-based (ACK) and/or receiver-based (NACK) error recovery. In [21] S. Pingali et al. have provided a quantitative demonstration of the superiority of receiver-based approaches over sender-based approaches.

2.3.1 Reliable Multicast Protocol

The Reliable Multicast Protocol (RMP) [26] provides a totally ordered, reliable multipoint transport protocol on top of an unreliable multicast service.

Besides using multicast where available, RMP extened the Reliable Broadcast Protocol (RBP) [7] with a name service that advertises the existence of multicast groups, and a flow and congestion control mechanism. RMP's name service uses the RMP protocol to manage its distributed database. RMP's group membership protocol allows sites to join or leave a group and updates the members' group membership view accordingly.

RMP's flow and congestion control mechanism uses the algorithms Van Jacobson developed for TCP [10]. Each sender in a group keeps a sliding window regulated by retransmission timers, NACKs form receivers, and positive ACKs from the token site. A sender uses slow-start to increase its window size by 1 packet every time an ACK is received. When a retransmission timer expires signaling congesion, the sender exponentially reduces its window size. To implement flow control, a RMP receiver multicasts a NACK to the group (instead of unicasting it to the token site) when it detects a lost packet. When a sender receives a NACK, it treats it just like an expired timer and backs off exponentially.

2.3.2 Scalable Reliable Multicast

The Scalable Reliable Multicast (SRM) [23] is an example of the receiver-based approaches. Wb, the distributed whiteboard tool presented in [23], uses the Scalable Reliable Multicast (SRM) transport protocol to reliably distribute data among all participating sites. Participants multicast NACKs to request retransmission of lost data, which can be answered by any member that has the information. To avoid generating multiple copies of retransmitted data, retransmissions are multicast to the group. To further reduce the implosion problem, a site waits a random period of time before sending a NACK or retransmitting data, and suppresses its own transmission in case it hears it from another member of the group.

A new site joins an ongoing whiteboard conference by announcing its presence with a join message. Current members update their view with new site. The new site's view of the group is gradually updated through periodic session messages generated by current members when the group stays quiet for a certain period of time. Session messages are also used to measure RTT between members, which are used to set the suppression timers.

2.3.3 Reliable Adaptive Multicast Protocol

The Reliable Adaptive Multicast Protocol (RAMP) [13] was developed to operate in an all-optical, circuit-switched, gigabit network environment. RAMP is layered on top of a network-layer multicast protocol, such as IP multicast, and provides NACK based reliable delivery. RAMP is a receiver-initiated based protocol that selectively unicasts or multicasts a retransmission, depending on the number of received NACKs.

2.3.4 Adaptive File Distribution Protocol

The Adaptive File Distribution Protocol (AFDP) [12] provides a reliable file distribution service on top of UDP. AFDP senders, or *publishers* send data to AFDP *subscribers*. A special subscriber in a group, or the groups's *secretary*, is responsible for managing group membershipo, authorizing publishers, and determining the appropriate transmission mode to be used. While AFDP uses multicast as its preferred transmission mode, AFDP packets can also be transmitted using broadcast, or unicast, whenever multicast is not available.

To achieve reliability, AFDP uses NACKs to request retransmission of lost packets. Senders retransmit lost packets to the entire group. AFDP's rate-based flow control mechanism slows down senders every time they receive a NACK. Senders gradually increase their transmitting rate after successful transmissions.

AFDP allows sites to join and leave a distribution group by contacting the group's secretary, but does not provide a recovery mode to handle site failures or network partitions.

2.3.5 Log-Based Receiver-Reliable Multicast

The Log-Based Receiver-Reliable Multicast (LBRM) [5] uses a logging server to log all packets transmitted by the source and to retransmit lost packets. The source keeps data that has already been sent until it gets an ACK from the loggin server. LBRM uses NACKs between receivers and the logging server and ACKs between the logging server and the source.

Logging servers may be organized in a hierarchy. Secondary logging servers act as a proxy between their clients (local receivers) and the primary logging server. They perform local recovery if they happen to have the packet lost by the local receivers, or request the lost packet form the primary log server on behalf of its clients. Logging servers decide wheterh they should multicast or unicast a lost packet based on the number of ACKs they got.

2.3.6 Others

Yavatkar [18] proposes error recovery that is driven by receivers who use a combination of restricted negative acknowledgment with NACK suppression and periodic positive ACKs.

Other approaches arrange receivers in a virtual tree hierarchy with the sender at the root. The Reliable Multicast Transport Protocol (RMTP) [15] represents a multi-level hierarchical approach in which leaf receivers periodically send status messages to the Designated Receiver (DR) of their subgroup. DRs in turn, send their status periodically to the higher layer DRs. Lost data packets are always requested from a higher lever DRs, not making use of local retransmissions between neighboring receivers.

All of the above work assumes that there is only a single multicast group, to which all the receivers belong.

McCanne [20] have proposed the use of multiple multicast groups for flow and congestion control, but not for error recovery. In [23] the possibility of using separate multicast groups for defining "local groups" for local recovery has been suggested. Holbrook [5] proposes the use of separate retransmission channels for error recovery as future work. Kasera et al. [19] examine an approach that involves the use of multiple multicast groups for reducing receiver processing costs.

Chapter 3

Design of the Flexible Multicast Protocol

This chapter discusses the design of FMP and presents the protocol itself. First the primary features and assumptions of FMP are introduced. Second, the protocol description is described. Next, the data structures and their operations are detailed.

3.1 Primary Features and Assumptions

3.1.1 Data Propagation Method

In this study, FMP multicasts packets using IP multicast via UDP, and unicasts (some retransmission) packets via UDP.

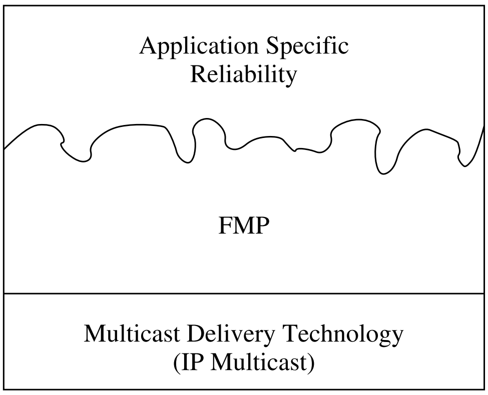

Figure 3.1: FMP Architecture

3.1.2 Application Data Unit (ADU)

FMP emerged the concept of *Application Layer Framing (ALF)* [3] because a realization that applications did not want a single transport layer to perform everything for the application. The ALF design principle suggests that data should be transmitted on the network in units which are idempotent and thus can be utilized by the application independently of other application data units. This is exactly what is required for multicast based share applications, so that loose consistency may be maintained, and data can be presented to the user as soon as it is available. Permitting this heterogeneity is essential for applications to scale. The application can then handle reliability and consistency issues as it sees appropriate depending on the application context.

We assume that applications access data as their *application data unit (ADU)*, and that an ADU is the smallest semantic unit that can be processed independently. The size of ADU may vary from application to application or from connection to connection. In this study, we assume that the ADU size is agreed upon by applications and the transport protocol, and remains fixed during a particular session. An ADU is the transport protocol data unit on retransmission will be based; the size of a packet is also determined on a session-by-session basis.

3.1.3 Application Reliability Requirements

We further assume that the applications are sophisticated enough to understand their error tolerance requirements. Multicast applications require varying degrees of reliability. Application reliability requirements can be loosely defined as follows: [16]

- Best effort reliability is similar to that which is provided by the UDP-based IP multicast delivery schemes most commonly present today. No reliable delivery is guaranteed.
- Absolute reliability is the most familiar requirement. It states that all packets in a session be reliably delivered to the receivers. This is the form of reliability that is commonly supported by TCP at the transport layer for unicast sessions.
- Bounded latency requires that each packet adheres to a specified lifetime over which the data is useful to the receiver. This is defined as an upper bound on its delivery latency. Packets arriving outside this timeframe are discarded. The common application requiring bounded latency is a realtime video stream. Each packet has a “playback” time and any packet not meeting this deadline is discarded.
- Most recent reliability is reliable transmission where only the most recent data of a particular parameter is of interest. A simple example would be a service that provides reliable stock updates. If a particular stock update is lost, and a new update is received before a retransmission can occur, the old data is rendered useless. Thus, it is possible that the data may take on a value that is never known to some or all of the receivers. Most recent reliability is a common requirement for many data types in advanced distributed simulations and in situational awareness dissemination.

However, in case of network-related problems, such as network partitions, FMP does not guarantee reliability. In such cases, the receiving application is notified. In addition, FMP does not ensure reliable delivery of data to receivers that voluntarily or involuntarily (due to a crash or network partitioning) leave a multicast group. We assume the existence of *Session Manager* who is responsible for detecting such anomalies and taking necessary actions.

3.2 Protocol Description

3.2.1 Transmission Channels

FMP creates 2 channels for transmission; one for tranferring data, and the other one for transferring control information.

3.2.2 FMP Entities

FMP is organized around three entities: FMP Sender Process, FMP Receiver Process, FMP Retransmittable Receiver (RR) Process. RR is the FMP Receiver process that can retransmit lost data packets as the sender process. To be RR is selected by application configuration.

A FMP Process is the basic entity of a larger system that uses FMP to communicate. Multiple processes may exist on the same host. Under this requirement, it is necessary to uniquely identify each one. This is accomplished by taking the IP address of the host and combining that with the protocol data and control port that the process will use for communication. This combination is referred to as the *FMP Address*, and should be unique across the entire internetwork.

A FMP Process initialize the FMP according to the *FMP Configuration* specified by the application. Every FMP Processes are required to maintain the *FMP Address List*. Each entry in the list stores the information about the properties of the FMP processes in the same group. The FMP Address List contain at least 2 entries; one is for FMP Process itself, and the other one is for the Sender Process of the group.

3.2.3 Connection Establishment and Termination

A FMP connection is identified by a pair of endpoints: a source endpoint and a destination endpoint. The source endpoint consists of the sender's FMP Address; the destination endpoint consists of the multicast group address and a port number. Each FMP connection has a set of associated connection parameters (see table 3.2). FMP assumes that the application is responsible for providing the sender and the receiver(s) with the associated connection parameters. FMP uses default values for any connection parameter that is not given.

Once the application has provided the sender and the receivers with the session information, receivers initialize the connection control block and stays in an unconnected state; the sender starts transmitting data. On receiving a data packet from the sender, a receiver goes from the unconnected state to the connected state and starts emitting feedbacks[1] at the feedback time interval. (feedback timeout is dynamically adjusted based on the round-trip time (RTT) measured from a receiver to its sender to minimize unnesscessary retransmissions; see section 3.2.5 for a detailed description)

Connection termination is timer based. After the sender transmits the last data packet, the sender goes into quit waiting state, starts a quit waiting timer. When the timer expires, the sender deletes all state information associated with the connection (i.e. it deletes the connection's control block). The quit waiting time interval is at least twice the lifetime of a packet in an internet. Any feedbacks from receivers arrived in this time interval will reset the timer to its initial value. A normal receiver deletes its connection control block and stops emitting feedbacks when it has correctly received all data packets. The retransmitable receiver (RR) behaves like a normal receiver except that it deletes its connection control block only after the quit waiting timer expires.

Because receivers periodically emit feedbacks after connection establishment and the time interval between consecutive feedbacks is much smaller than the quit waiting time, the sender uses feedback receptions during the quit waiting time interval to deduce whether all receivers have received all data. If it does not receive any feedbacks during the quit waiting time interval, the sender assumes either all receivers have received every packets, or something exceptional has happened that prevents the receivers from sending feedbacks. Possible "exceptional" situations include: network partition and receivers voluntarily or involuntarily leaving the multicast group. As described in section 3.1, FMP assumes that the application is responsible for detecting such situations and taking nesscessary actions.

In addition to normal connection termination, the control reset packet can be used to terminate connections. For example, when FMP detects that the sender has aborted before data transfer is complete, it uses the control reset packet to inform all the receivers to close the connection.

3.2.4 Reliability Mechanisms

FMP uses *receiver-initiated approach*; it places the responsibility of ensuring sequenced, lossless data reception on each individual receiver. Thus, the sender is relieved of the burden of tracking the status of each receiver. FMP uses retransmission for provide reliability. FMP's retransmission mechanism is based on a modified version of the SMART [24] retransmission mechanism to work with multicast. SMART is a retransmission strategy that combines aspects of the traditional techniques go-back-n and selective-retransmit.

The key idea in SMART is to build the bit-mask of correctly received packets at the sender. Each ACK therefore carries two pieces of information: the cumulative ACK (CACK), as in

[1] We use the term "feedback" instead ACK or NACK. See 3.2.4 for more details.

standard go-back-n, and the sequence number of the packet that caused the ack to be initiated. Thus, the second piece of information not only allows the sender to know which packets have been correctly received, but also allows the sender to correctly infer which packets have been lost, and to take corrective action. That is, when a source receives an ACK, it compares the CACK with the sequence number of the packet that caused the acknowledgment. If the two are not the same, the ACK must be a duplicate ack, and the source guesses that all the packets in the sequence space between the CACK and the packet being acked are lost, and places them in the transmission queue for subsequent retransmission. Because the ACK in SMART idea acts 2 roles, as ACK when the CACK equals the sequence number fo the packet that caused the acknowledgment, and if it is not equal as NACK. We will use the term *feedback* instead ACK or NACK in this study.

This idea works with unicast well, but when uses with multicast, the feedbacks from a large number of receivers may overwhelm the sender (i.e., the feedback implosion problem described in section 2.2 still exists). Furthermore, large number of feedback packets destined for the group and retransmitted data packets generated from the sender may congest the sender's neighboring routers and local networks. In order to avoid this situation, FMP uses policy like SRM [23] that allows any other receivers that received that feedbacks and has the lost ADU to be able to retransmit too. We modified SMART idea by adding feedback timer at each receivers for avoiding the feedback implosion problem, and change the second piece of information in SMART's ACK packet, the sequence number of the packet that caused the acknowledgment to the last sequence number of the gap. For example, see figure 3.2.

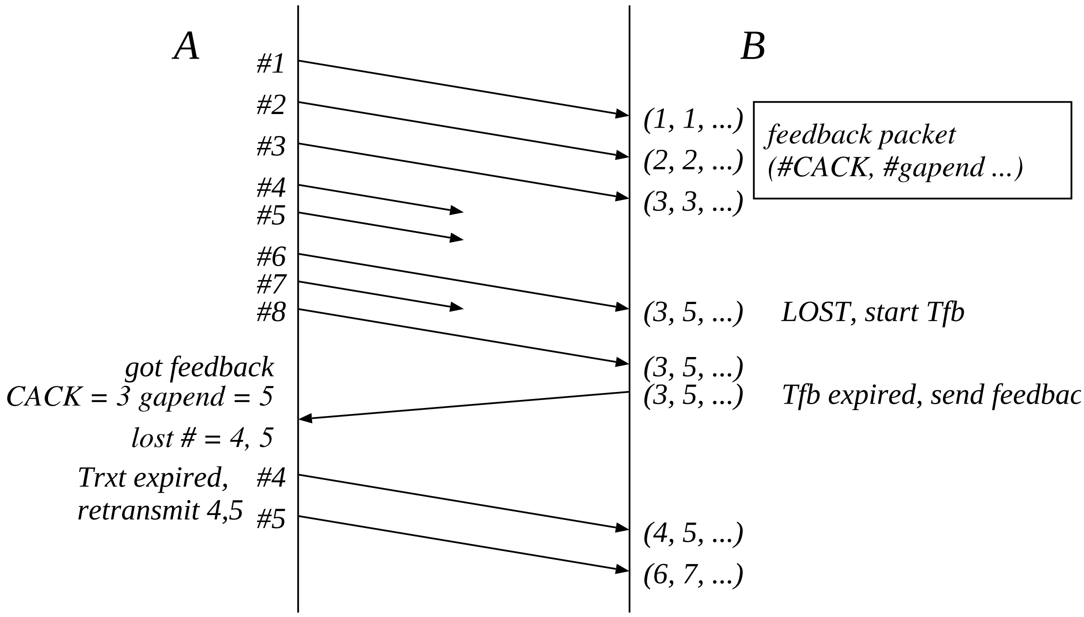

Figure 3.2: FMP Lost ADU Detect Mechanism

Every time each receiver received the data packets, it resets its feedback timer, and if in that interval it didn't get any feedbacks from other receivers that indicate to have the same feedback status (eg. CACK, the lost packet sequence number to retransmission if any) it multicasts the feedback to group with TTL specified in application configuration. Anyway the receivers double the retransmission timer to wait for the repair. Thus, avoid the duplicated feedbacks to overload the sender, and other receivers can check whether now they too lost the packets or not from this feedback.

Upon received the feedback, the sender and RRs check whether the retransmission is need or not, and which packet sequence number need to be retransmited, then start the retransmit timeout. If the retransmit packet is retransmited by the others, then cancel the retransmission. Normal receivers check whether that they have lost some packets or not.

FMP uses 2 modes for retransmission; unicast, and multicast. In order to determine whether a packet must be retransmitted by unicast or by multicast, the sender or RRs have to wait for retransmission timeout interval to collect retransmission requests from from several receivers. This may introduce a delay in delivering the requested packets to the waiting receivers. However, there is an option for the application to set the retransmission timeout to 0. In that case, the sender or RRs will immediately process each received feedbacks and retransmit packets using unicast. There is an obvious trade-off here between efficiency and latency. If the number of receivers that need retransmission more than threshold, then FMP uses multicasting to retransmit, else FMP uses unicasting to retransmit. If FMP uses multicasting for retransmission, the retransmit packet is multicasted with minimal TTL first for avoiding unnesscessary processing duplicated one at the FMP Processes that didn't lost the packet.

3.2.5 Measuring RTT and Calculating the Feedback and Retransmission Timeout

The receivers in FMP send feedback control packets periodically to group. If these feedbacks are sent too frequently, the network may end up with control packets or with retransmitting the same packet multiple times without knowing if the first retransmitted packet was received correctly by the receivers.

In order to prevent such redundant retansmissions, FMP uses RTT measurement method based on [23]. In addition to determine whether any packets are lost or not, the sender and receivers use the feedback control packet to estimate the one-way distance between nodes. The feedback control packets for that group include a source FMP Address, and a timestamp. The feedback packet timestamps are used to estimate the host-to-host distances needed by the retransmission mechanisms.

Assume that host A sends a feedback packet FB_1 at time t_1 and host B receives FB_1 at time t2. At some later time t_3, host B generates a feedback FB_2, marked with (A, t_1, Δ) where $\Delta = t_3 - t_2$. Upon receiving P_2 at time t_4, host A check if the FMP Address is it or not, if so then it can estimate the latency from host B to host A as $(t_4 - t_1 - \Delta)/2$, or equivalently, as $((t_4 - t_3) + (t_2 - t1))/2$. Note that while this estimate does not assume synchronized clocks, it does assume that paths are roughly symmetric.

3.2.6 Data Cache

FMP allows receivers to join an ongoing session at any time and still receive the entire data reliably if it requires. In order to provide this feature, the senders and the retransmitable receivers (RRs) in FMP need not to buffer the entire file during the session, because FMP include the ALF concept in design, so when the request ADU is not in received ADU buffer, the sender or RRs can rebuild from the original source (sender), or from the saved ADU (RRs). This allows receivers to request for the retransmission of any transmitted ADU. A one-level caching mechanism is used in FMP. The most recent ADU are cached in memory, and the other ADUs are extracted from the source file when necessary. In-memory cached packets contain application data as well as protocol headers to improve efficiency. The buffered data size at

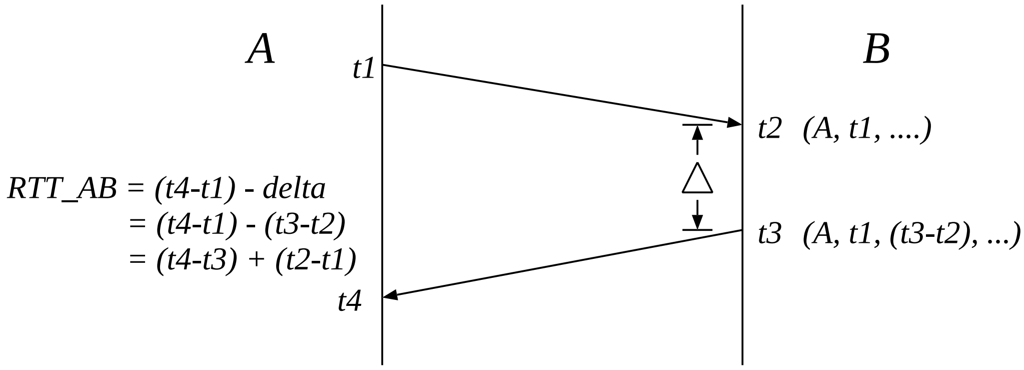

Figure 3.3: RTT Measurement

the sender and the RRs can be reduced by determining how much data can be discarded based on the context of the application.

3.2.7 Independent State

The state information maintained at each multicast participant is independent of the number of participants. Therefore, when a receiver join or leaves a multicast group, it does not affect the state information of the sender or other receivers. FMP requires the sender and every receivers maintain state table of itself and the sender, they will may maintain the state of others when they received the control packets from others.

In this study, FMP makes no assumption about the processing power or the link bandwidth at each receiving node. It uses windowed flow control with congestion avoidance to avoid overloading slow receivers and links with low bandwidth. It also allows users to set an upper bound on the sender's data transmission rate. We assume that the Session Manager is responsible for choosing and setting the maximum transmission rate. To cope with receivers with varying speed, FMP takes a conservative approach by using the progress of slow receivers to determine how much more data needs to be transmitted by the sender. However, techniques to prevent a slow receiver from slowing down faster receivers are being investigated.

3.3 The Data Structures

3.3.1 FMP Packet Types

FMP is a packet based system. This means that each message sent is a sigle packet. Fragmentation and reassembly of large messages into a stream is the task of higher level interfaces.

FMP creates 2 channels for transmission; one for tranferring data, and the other one for transferring control information. Each packet type contains header. Only control packet type has subtype so its header contains control type. The exact format of each packet is shown in Appendix A. Table 3.1 lists the packet types defined in FMP. Their function will be described in the following sections.

Packet Types	Description
`DATA_ORIG`	original data packet
`DATA_RXT`	retransmitted data packet
`CTRL_FB`	feedback packet
`CTRL_EOADU`	end of application data unit
`CTRL_RST`	reset (terminate) connection

Table 3.1: FMP Packet Types

3.3.2 FMP Configuration

The FMP configuration that can be set by application is divided into 3 parts: group configuration, host configuration, and application configuration. See table 3.2 for the lists of configurations and descriptions.

Configuration	Description
Group Configuration	
grp_name	parent group name or IP address
grp_serv	group data service port
grp_ttl	TTL used for sending messages
Host Configuration	
host_name	interface name or IP address
is_RR	need to function as retransmittable receiver or not
Application Configuration	
input_fp	input file pointer
input_func	input function
output_fp	output file pointer
output_func	output function
$ADUSIZE$	application data unit length
numadubuf	number of ADU buffers
numadu	number of ADU for tolerance check
maxadulost	max allowed ADU lost per numadu
maxaduburst	max allowed ADU burst
ttl_minfb	mininum TTL used for sending feedback
T_{FB}	time interval to send `CTRL_FB`
T_{EOADU}	time interval after sending `CTRL_EOADU` and no more `CTRL_FB`s receive

Table 3.2: FMP Configuration

3.3.3 FMP Entity Status Information Table

FMP requires each FMP process to maintain table that store the latest status informations of the other processes in the group. At least 2 entries are required; the entry for FMP process itself, and the entry for the sender process of the group. The sender and RR processes need to maintain more entries for using those status informations in FMP reliable mechanism. Table 3.3 lists the status informations and descriptions.

Information	Description
Group Configuration	
grp_id	entry ID for this group status (library internal use)
grp_ttl	TTL used for sending data to group
grp_maddr	multicast group's data channel address (IP address + port)
data_msfd	data send/recv socket file descriptors
grp_caddr	multicast group's control channel address (IP address + port)
ctrl_msfd	ctrl send/recv socket file descriptors
Host Configuration	
host_id	entry ID for this host status (library internal use)
hmaddr	this host's data channel address (IP address + port)
hcaddr	this host's control channel address (IP address + port)
rtt	estimated RTT to this host
lastseen	timestamp when received previous packet from this host
fb	feedback control packet for this host [2]
eoadu	eoadu ctrl pkt for this host [3]

Table 3.3: FMP Entity Status Information

3.3.4 ADU Buffers

This ADU buffers are used by the sender or RRs for retransmit when needs. The buffers may need for buffering unorder ADU for normal receivers if they require to get ADU in order.

3.3.5 Retransmission Queue

When the sender or RRs detect that the retransmit is need it insert the ADU sequence number need to retransmission to, the detected timestamp, the requestor's FMP address and the number of requests to this queue.

3.4 FMP State

Table 3.4 shows the FMP state, and figure 3.4 shows FMP state transition diagram.

[3] Used only if this entry stores the sender process information. FMP is designed first to support many-to-many, but in this study we support only 1-to-many only.

FMP State	Description
UNCONNECT	initial state for FMP Processes
CONNECTED	ADU transfer state
ADU_LOST	some ADUs are lost
ADU_RXT	there are entries in retransmit queue
QUIT_WAIT	all ADU had transmitted, waiting for closing connection
QUIT	connection is closed

Table 3.4: FMP State

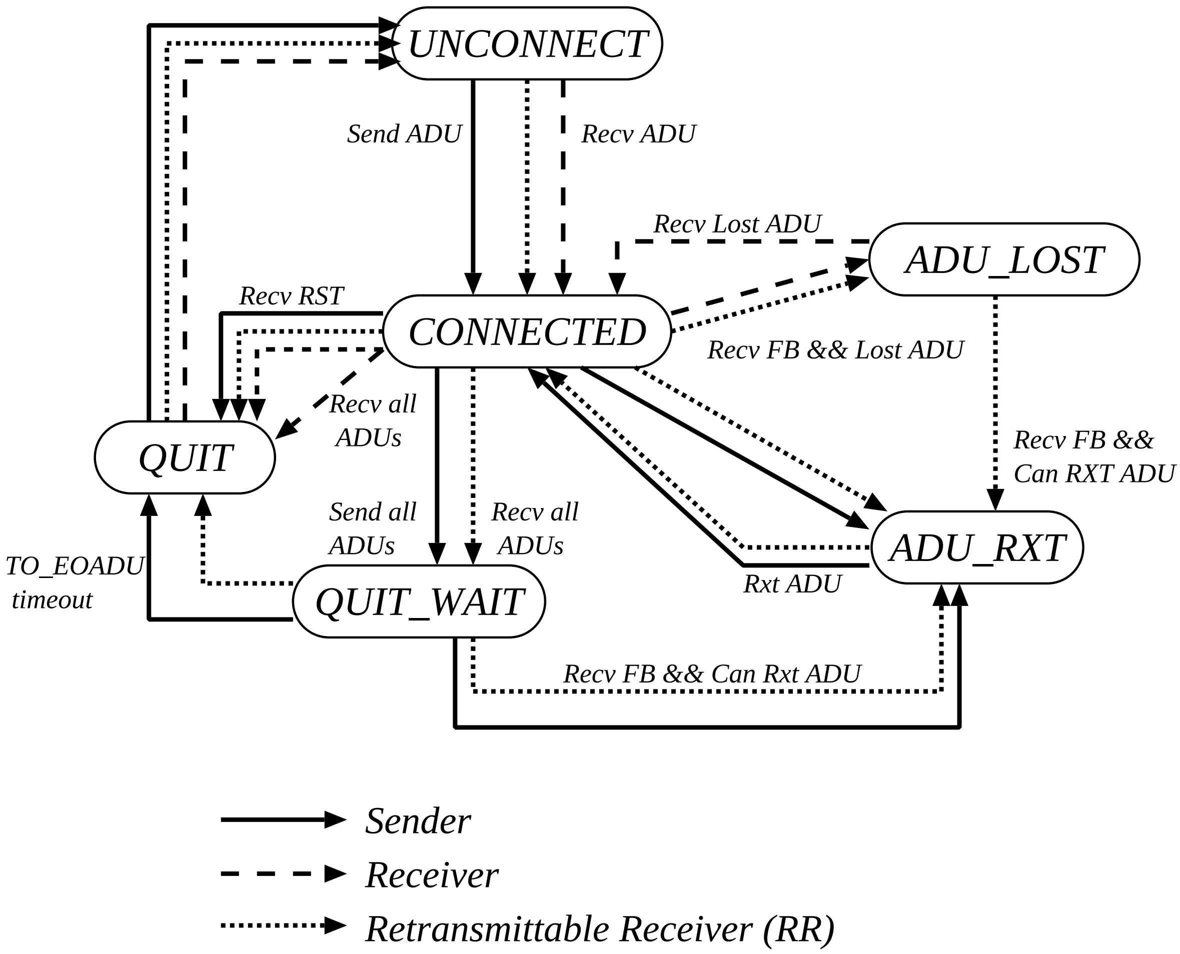

Figure 3.4: FMP State Transition Diagram

Chapter 4

Implementation of the Flexible Multicast Protocol

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

4.1 Major Implementation Decisions

4.1.1 Implementing Protocols in the User Level

Traditional protocols have been implemented as monolithic entities in the operationg system kernel or in a single trusted user-level server. The reasons for this have primarily been concerns over security and performance. Recently other important issues, such as ease of code maintenance, debugging, customization, and the need to have concurrent, multiple operating protocols, have prompted many to implement protocols in the more managable user level [27]. Typically these implementations have taken the form of user-level libraries of protocols which can be linked into the application. It has been shown that traditional protocols can be implemented in user space and suffer no major degradation in performance. With these facts in mind, we implemented multicast packet forwarding and FMP protocol processing at user level. Thus, multicast packets travel on UDP. A user level protocol process implements the FMP protocol. Application level receivers and senders use UDP to communicate with the FMP protocol process.

4.1.2 Event-Driven Control

Because FMP was to be implemented in the user-level as a library for use in applications, a control paradigm had to be chosen to reflect the diverse ways in which develpers implement their own distributed applications. The paradigm chosen was an event-driven approach. FMP relies on events occurring and states being transitioned to and out of on specific event conditions, thus the event-driven approach for the implementation seemed very close and natural. FMP requires application to register its input and output functions when the protocol is initialized. Then FMP will constructs a control loop and read input and send output to application functions. When the transmission is finished the control is returned to application.

This FMP implementation has been implemented under 4.4BSD (FreeBSD 2.2.5) and SVR4 (Sun Solaris 2.5) UNIX systems. In order to implement retransmit timers and alarms, the software signal facilities[1]had to be used. This imparts the only non-control sequence restriction that the implementation imposes on the developer. The program using FMP must use FMP to set any alarms or timers it may need. It should not set up its own signal trap for the alarms. This would cause unknown behavior for protocol operation.

4.2 The FMP Internal Data Structure

4.2.1 FMP Property Data

The FMP Property contains every configurations about the protocol, and the current protocol state. For details see data structure `FMP_PROP` in Appendix B.2.1. The FMP protocol property is divided to 3 subproperties:

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

- FMP Process Property Table
  The FMP Process Property Table contains informations of FMP processes active in the session. For details of one entry in this table see data structure `FMP_HOSTPROP` in Appendix B.2.1.
- FMP Group Property
  The FMP Group Property contains informations of FMP Destination Group of the session. For details see data structure `FMP_GRPROP` in Appendix B.2.1.
- FMP Application Property
  The FMP Application Property contains informations of application that use FMP. For details see data structure `FMP_APPROP` in Appendix B.2.1.

4.2.2 FMP Send and Receive ADU Buffers

The buffer space for ADUs is also preallocated and pooled for use as is needed. ADU buffers have their fixed header set to default values when the ADU is placed back in the pool. This is to ensure that the vital information in that header is either valid because it is filled or it is invalid because the default values were still left after it was pulled from the pool for use. For details see data structure `FMP_ADUBUF`, `FMP_ADUBUFLIST` in Appendix B.2.3.

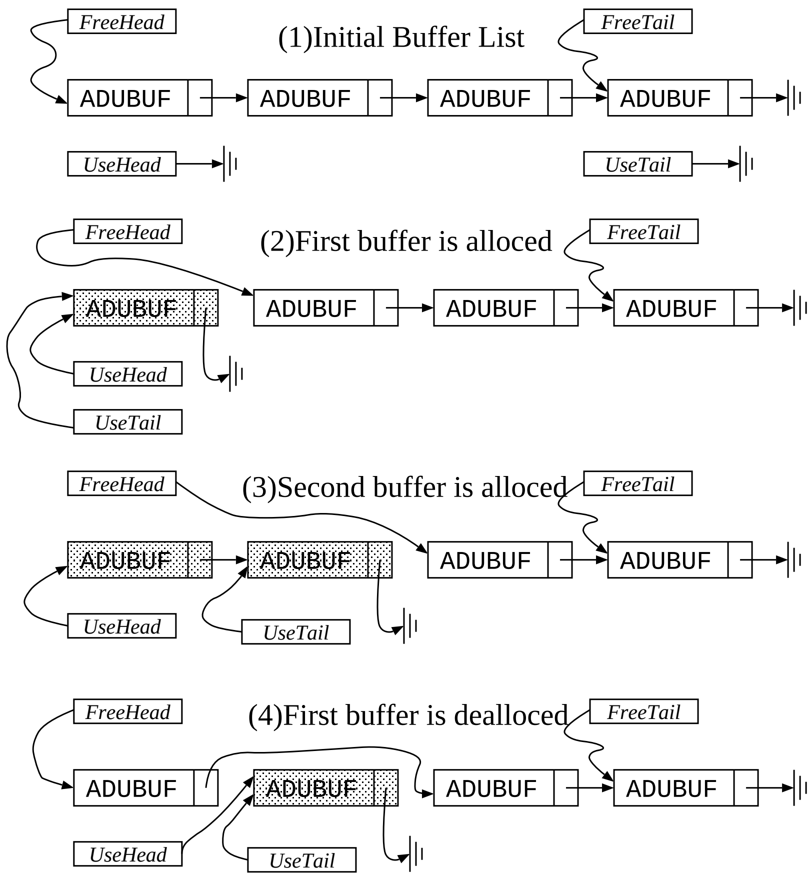

Figure 4.1: ADU Buffer List Allocation and Deallocation

4.3 The FMP External Data Structure

This section describes the data structure that application that use FMP needs.

4.3.1 FMP Protocol Configuration

The FMP Protocol Configuration contains every configurations about the protocol. For details see data structure `FMP_CFG` in Appendix B.1.1. This configuration is divided to 3 subconfigurations:

- FMP Host Configuration
  The FMP Host Configuration contains informations of the local FMP processes in the session. For details see data structure `FMP_HOSTCFG` in Appendix B.1.1.
- FMP Group Configuration
  The FMP Group Configuration contains informations of FMP Destination Group of the session. For details see data structure `FMP_GRPCFG` in Appendix B.1.1.
- FMP Application Configuration
  The FMP Application Configuration contains informations of application that use FMP. For details see data structure `FMP_APPCFG` in Appendix B.1.1.

4.3.2 FMP Statistics

The FMP Statistics data structure contains every FMP statistics that application can access via global variable `fmp_stat`. For details see data structure `FMP_STAT` in Appendix B.1.1.

4.4 The FMP Application Programming Interface

- `int fmp_init(FMP_CFG *`*fmp_cfg*`)`
  The `fmp_init()` initialize the FMP protocol according to a FMP protocol configures in *fmp_cfg*. (see section 4.3.1 for `FMP_CFG` details)
- `int fmp_input(int `*infd*`, FILE *`*infp*`, int *`*saduseq*`, char *`*sbuf*`, int `*sendbytes*`)`
  The application that use FMP need to provide input function that have same prototype as `fmp_input()`. `fmp_input()` will be called when the application want to send ADU with arguments:
  - *infd* : The input UNIX file descriptor.
  - *infp* : The input UNIX file pointer.
  - *saduseq* : The pointer to ADU sequence number of this ADU filled by application.
  - *sbuf* : The buffer that the application fills the ADU in.
  - *sednbytes* : Number of bytes send.
- `int fmp_output(FMP_ID `*fromid*`, int `*raduseq*`, char *`*rbuf*`, int `*recvbytes*`)`
  The application that use FMP need to provide output function that have same prototype as `fmp_output()`. `fmp_output()` will be called when the ADU is received with arguments:
  - *fromid* : The FMP Process ID of the FMP sender process.
  - *raduseq* : The ADU sequence number of this received ADU.

  - *rbuf* : The ADU itself.
  - *recvbytes* : Number of bytes received.

- `void fmp_prtstat()`
  `fmp_prtstat()` will display the FMP collected statistics. It should be called after the session has finished. (see section 4.3.2 for details)

- `void fmp_loop()`
  After calling `fmp_init()` for initialize FMP protocol, the application calls `fmp_loop()`, then the control is moved to FMP until the data transfer is completed or there are any errors detected.

Chapter 5

Performance Results

This chapter presents the performance achieved by this FMP implementation. The first section presents performance observed on a 10 Mbps Ethernet. The last section briefly discusses performance issues as have been observed on a portion of the WAN.

5.1 LAN Performance

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

All tests were run on a Sun Microsystems SPARCstations-5, a Sun Microsystems UltraSPARC-II running Solaris 2.5 and x86 machines running FreeBSD 2.2.5 operating system on a 10Mbps Ethernet. Table 5.1 shows the details of host specifiction, and figure 5.1 shows the network topology used in this test. The network was operating under light load before and after the tests. Each test constituted of ten trials and the average of the ten trials is shown. Throughput measurements were taken by timing transfer of 5 megabytes of data and does not include packet headers, the ADU size is 8192 bytes. Table 5.2 shows four test configurations separated with number of receivers per one sender. No fair comparison with other reliable multicast protocols can be made using these tests as performance numbers form other protocols have been collected on varying types of platforms and operating systems.

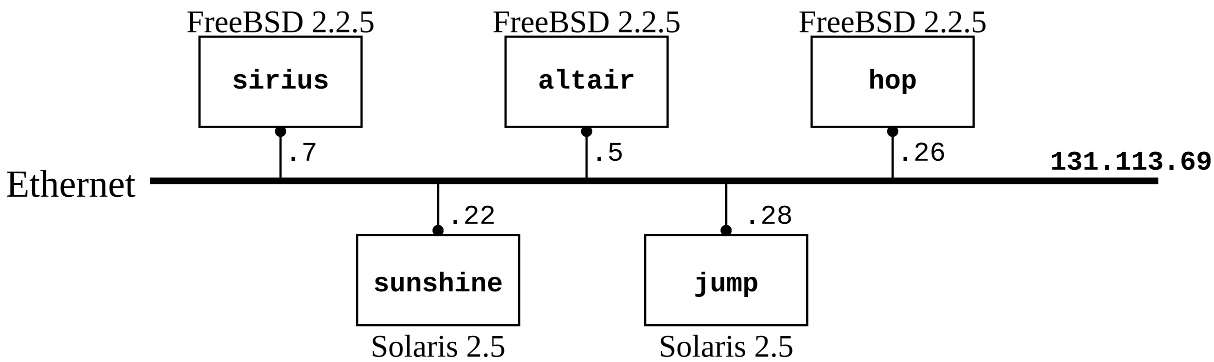

Figure 5.1: Test Network Topology

Hostname	IP address	OS	CPU	Memory
sunshine	131.113.69.22	Solaris 2.5	UltraSPARC-II (296 MHz) (x2)	512MB
jump	131.113.69.28	Solaris 2.5	SparcStation-5 (85 MHz)	96MB
sirius	131.113.69.7	FreeBSD 2.2.5	Pentium Pro (299.15-MHz)	64MB
altair	131.113.69.5	FreeBSD 2.2.5	Pentium Pro (199.43-MHz)	192MB
hop	131.113.69.26	FreeBSD 2.2.5	AMD Am5x86	32MB

Table 5.1: Test Hosts Specificition

Number of Sender:Receiver(s)	Sender	Receiver(s)
1:1	hop	sirius
1:2	hop	sirius, altair
1:4	hop	sirius, altair, sunshine, jump
1:8	hop	sirius(2), altair(2), sunshine(), jump(2)

Table 5.2: Test Configuration

5.1.1 Test Results

FMP – 1 Sender : 1 Receiver

No.	Throughput (KB/s)	Aggregate Throughput (KB/s)
1	551.688	551.688
2	555.981	555.981
3	560.900	560.900
4	555.178	555.178
5	547.274	547.274
6	566.423	566.423
7	555.849	555.849
8	565.137	565.137
9	557.866	557.866
10	550.444	550.444
Average	556.674	556.674

Table 5.3: Test Results, FMP – 1 Sender : 1 Receiver

Non-multicast – 1 Sender : 1 Receiver

No.	Throughput (KB/s)	Aggregate Throughput (KB/s)
1	836.177	836.177
2	830.749	830.749
3	837.244	837.244
4	838.462	838.462
5	819.370	819.370
6	804.717	804.717
7	837.881	837.881
8	708.199	708.199
9	808.228	808.228
10	830.826	830.826
Average	815.185	815.185

Table 5.4: Test Results, Non-multicast – 1 Sender : 1 Receiver

FMP – 1 Sender : 2 Receivers

No.	Throughput (KB/s)		Aggregate Throughput (KB/s)
	sirius	altair	
1	545.192	545.133	1090.325
2	565.599	565.532	1131.131
3	559.356	559.334	1118.690
4	544.736	544.769	1089.505
5	554.945	554.880	1109.825
6	561.527	561.545	1123.072
7	571.782	571.512	1143.294
8	543.424	543.371	1086.795
9	549.880	549.825	1099.705
10	568.881	568.805	1137.686
Average	556.532	556.471	1113.003
	556.501		

Table 5.5: Test Results, FMP – 1 Sender : 2 Receivers

Non-multicast – 1 Sender : 2 Receivers

No.	Throughput (KB/s)	Aggregate Throughput (KB/s)
1	428.727	857.454
2	394.505	789.010
3	432.550	865.099
4	320.690	541.381
5	325.505	651.011
6	425.372	850.744
7	267.826	535.652
8	337.702	675.404
9	440.493	880.987
10	437.463	874.926
Average	381.083	752.167

Table 5.6: Test Results, Non-multicast – 1 Sender : 2 Receivers

FMP – 1 Sender : 4 Receivers

No.	Throughput (KB/s)				Aggregate Throughput (KB/s)
	sirius	altair	sunshine	jump	
1	533.444	533.391	533.582	533.603	2134.020
2	533.892	533.836	534.020	534.060	2135.808
3	559.626	559.569	559.778	559.863	2238.836
4	545.733	545.675	545.865	545.918	2183.191
5	555.946	555.894	556.092	555.796	2223.728
6	551.272	551.217	551.419	551.421	2205.329
7	556.896	528.623	557.041	556.227	2198.787
8	559.902	559.842	560.043	560.052	2239.839
9	549.120	549.071	549.262	549.205	2196.658
10	569.884	568.829	570.038	570.088	2278.839
Average	551.572	548.595	551.714	551.623	2203.504
	556.501				

Table 5.7: Test Results, FMP – 1 Sender : 4 Receivers

Non-multicast – 1 Sender : 4 Receivers

No.	Throughput (KB/s)	Aggregate Throughput (KB/s)
1	227.270	909.081
2	222.370	889.479
3	229.767	919.068
4	204.942	819.768
5	204.406	817.625
6	226.080	904.321
7	199.628	798.513
8	205.850	823.402
9	205.039	820.154
10	201.154	804.614
Average	212.651	850.603

Table 5.8: Test Results, Non-multicast – 1 Sender : 4 Receivers

FMP – 1 Sender : 8 Receivers

No.	Throughput (KB/s)								Aggregate Throughput (KB/s)
	sirius		altair		sunshine		jump		
	1	2	1	2	1	2	1	2	
1	542.631	542.654	542.582	542.608	542.971	542.971	442.512	328.116	4027.045
2	519.400	519.498	519.411	519.387	519.411	519.575	519.534	498.736	4134.952
3	558.874	558.856	558.797	558.821	558.995	559.012	557.927	559.037	4470.319
4	531.505	531.394	531.437	531.324	531.379	531.566	530.939	531.593	4251.137
5	505.017	505.032	504.968	504.989	505.165	505.157	382.739	344.614	3757.681
6	495.400	495.421	495.352	495.376	495.383	495.533	494.055	495.857	3962.377
7	445.279	376.376	445.240	445.258	445.401	445.401	444.135	445.777	3492.867
8	545.689	545.664	545.610	545.639	545.812	545.813	544.561	545.830	4364.618
9	538.570	538.553	538.500	538.524	538.712	538.517	536.893	537.876	4306.145
10	462.137	480.109	503.001	502.909	502.980	503.124	479.864	501.204	3935.328
Average	514.450	509.356	518.490	518.484	518.621	518.667	493.316	478.864	4070.247
	508.781								

Table 5.9: Test Results, FMP – 1 Sender : 8 Receivers

Non-multicast – 1 Sender : 8 Receivers

No.	Throughput (KB/s)	Aggregate Throughput (KB/s)
1	108.996	871.966
2	86.073	690.108
3	86.263	690.108
4	86.576	692.612
5	84.621	676.969
6	82.473	659.788
7	85.274	682.193
8	85.890	687.121
9	83.545	687.121
10	84.470	675.757
Average	87.418	701.374

Table 5.10: Test Results, Non-multicast – 1 Sender : 8 Receivers

5.1.2 LAN Aggregate Throughput

Aggregate throughput is the throughput the application sees. It is computed by taking the amount of data sent from all of the senders and multiplying it by the number of destinations. Figure 5.2 shows aggregate throughput for FMP with a variety of configurations. The application specific reliable was *Non-Loss Ordered.*

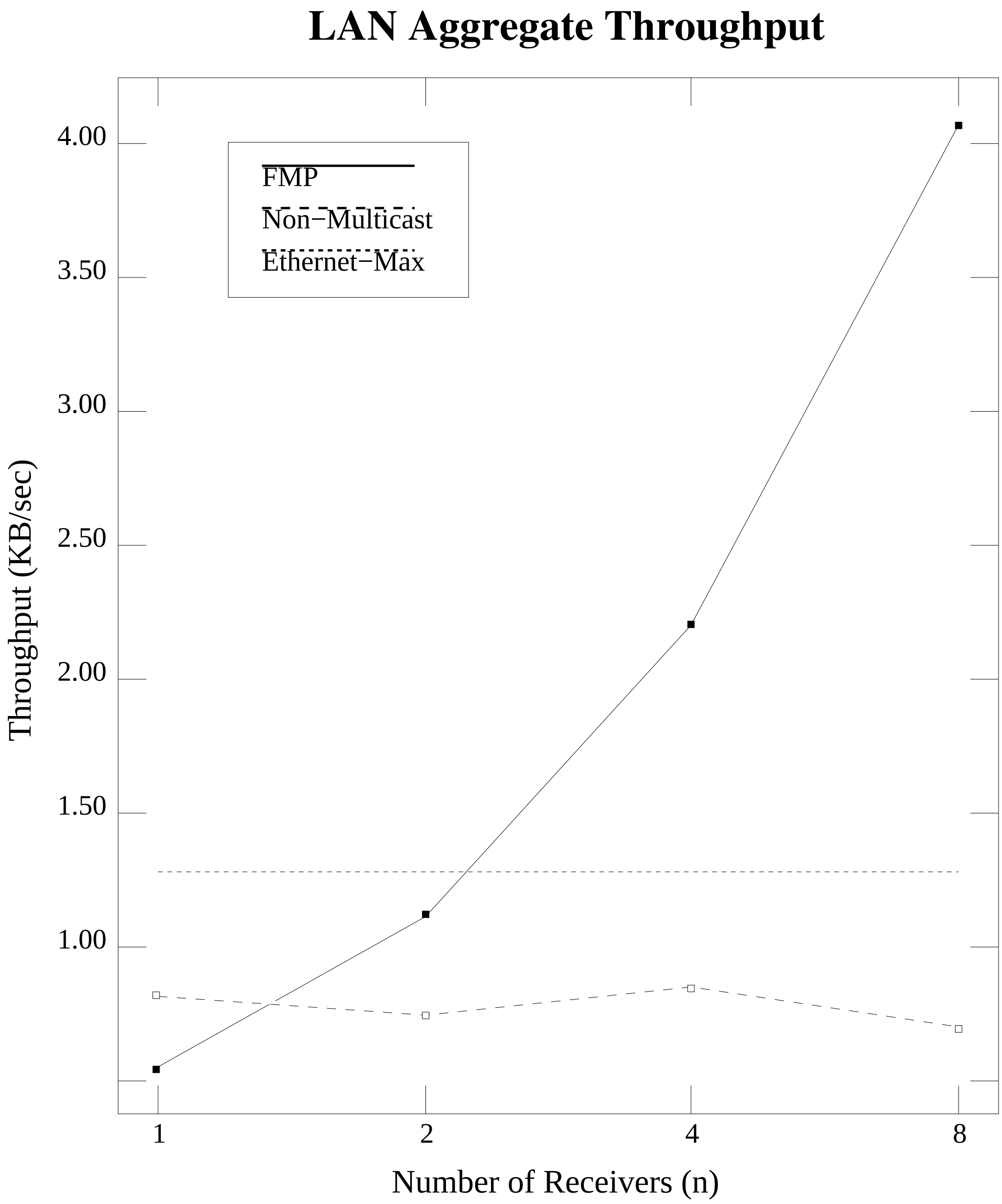

Figure 5.2: LAN Aggregate Throughput

This is the case that most applications use totally ordered multicasting for, and it is actually the worst case for the throughput of the protocol because of the load on the sender. It is not possible for a non-multicast or non-broadcast scheme to break the Ethernet throughput boundary shown in figure 5.2.

5.1.3 LAN Throughput

Figure 5.3 shows a LAN throughput. This is the aggregate throughput divided by the number of destinations. The sender is not counted as a destination in these tests. The non-multicast maximum throughput is shown. This drops off as a factor of $1/N$. This limitation is a fundamental limitation of the network. As can be seen, FMP stays roughly constant as the number of destinations increases.

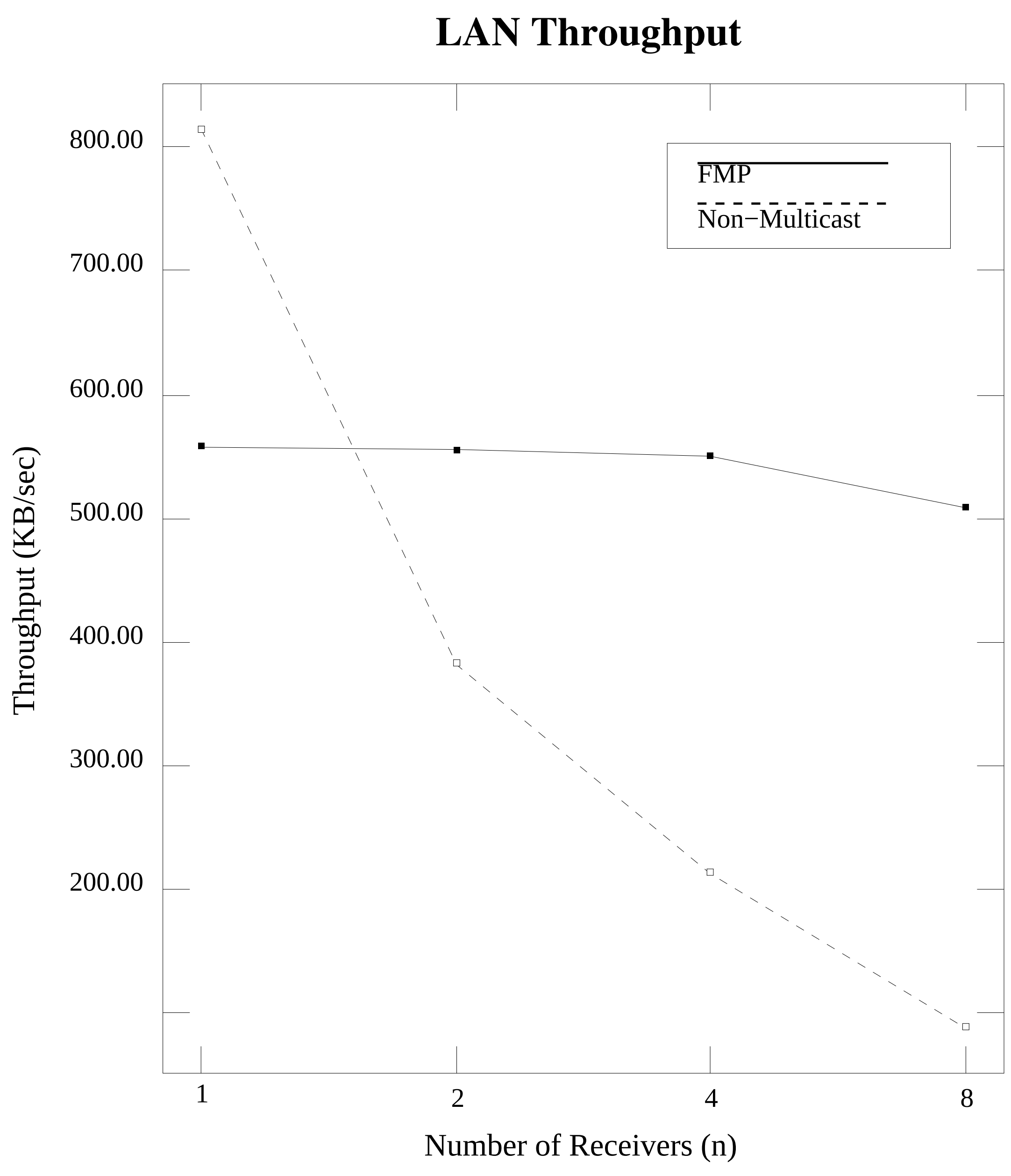

Figure 5.3: LAN Throughput

Chapter 6

Conclusions

This chapter draws some conclusions about FMP design, implementation. It presents the future directions of FMP.

6.1 Conclusions

Presented in this study are the Flexible Multicast Protocol, the design aspects of the protocol, the implementation notes for the protocol implementation. To summarize:

- Concurrent design and implementation allows implementation restrictions to be made clearer and handled earlier than normal software life-cycle models.
- Monolithic implementations of network protocols in operationg system kernels impose undue load on the operating system and jeopardize protocol evolution. Alternatives, such as user-level implementations, are increasingly becoming very usable and preferable.
- We observed the application layer framing (ALF) trend in multicast protocol design: more recent protocols target the requirements of specific applications. So the reliability requrements can be chosen by the application. The cost associated with such flexibility is very reasonable.
- Although our current protocol implementation is specifically tailored to multicast file tranfer, the protocol framework is generally applicable to a widfe variety of other applications.
- Because of their many-to-many communication model, reliable multicast protocols can have considerable negative performance impact on the underlying network. They can generate large amounts of data as well as control traffic, which may cause network congestion and eventually lead to congestion collapse. Furthermore, multicast protocols may try to inject packets at a rate, which the network, given its current state, cannot tolerate. These negative performance effects become more acute as multicast groups size grows. Therefore, existing and emerging multicast protocols should implement effective congestion control mechanisms allowing them to be efficient, yet sharing available bandwidth with other competing traffic.
- Many applications need or design support for reliable multicast. Experience with this design shows, however, that individual applications may have widely different requirements of multicast reliablity. Instead of designing a generic reliable multicast protocol to meet the most stringent requirements, this work has resulted in a simple, robust, and flexible reliable multicast algorithm that meets a minimal reliability definition of delivering all data to all group members, leaving more advanced functionalities, whenever needed, to be handled by individual applications.
- Flow and congestion control of multicast protocols deserves and requitres serious research, especially in the WAN environment.

6.2 Future Directions

6.2.1 Design Directions

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

6.2.2 Implementation Directions

The FMP implementation also holds a large amount of unanswered questions. The implementation at this stage in design and developemnt is fairly efficient, but imposes a large amount of overhead to processes that use it. Direct reduction of this overhead is desirable. One avenue that is under consideration to accomplish this is, to use dynamic libraries of some UNIX operating systems. This concept also brings up some questions about how to efficiently handle multiple sites per Internet host.

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

Acknowledgements

Appreciation and respect are given to the members of the Academic Examining Committee, Professor TOKORO Mario, and Associate Professor AMANO Hideharu. They have allowed the author freedom to examine and work on such an intensely demanding project and provided noting but encouragement. And many thanks to the admins and members of AMANO lab for providing the author about the research environments.

The author wishes to thank Dr.Raverdy P.G. at SONY computer science lab, NARITA Tomio, SAWATARI Ryusuke, the author's seniors at TOKORO lab for their many useful advices, guidances and comments about this research.

As immeasureable amount of appreciation and gratitude go to the Tokio Marine and Fire Insurance Company, Limited, especially MIYAKAWA Morihisa, MIWATA Masanobu, and SUGIMOTO Minako for their scholarship and supports for the author's 2 years studying here.

Next, the author would like to thank SHIMIZU Hisashi, the teacher at Internation Center of this campus, and TEZUKA Chizuko, the student counsellor for their encouragement, suggestions for the author, MURATA Minori, the author's Japanese language teacher for many readings and corrections of the author's Japanese papers.

The author would also like to thank to his Thai and foreigner friends at this campus for their encouragemnet, enjoy, friendliness. In particular to BOONSUPTHIP Boonwas and SHIAO Chien-tsair for their support and kindness during the author's student life here.

Finally and foremost, the author would like to thank his family and relatives in Thailand for their encouragement, understanding, endurance, and unquestionable support and love. Without them this would not have been possible.

February 1998
LAOHAKULVATHIT, Pornchai

Appendix A

FMP Packet Format

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

A.1 FMP Fixed Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  FMP Version  | Data/Ctrl Port|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   FMP Packet Send Timestamp                   |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           FMP Packet                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

FMP Version: Specifies the version of the FMP protocol. The current version is 1. Any packet received with an invalid version should be discarded.

Data/Ctrl Port: Specifies the data address port if the FMP packet is control packet, or the control address port if the FMP packet is data packet.

FMP Packet Send Timestamp: Specifies the FMP packet send time. This information will be used when in feedback control packets to let the source calculate the RTT to the receiver that send the feedback.

A.2 FMP ADU Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|   Data Type   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               FMP Address of Original Sender                  |
+                                                               +
|             (IP address + data port + ctrl port)              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ADU Sequence Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  ADU Length   |              Data                              ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Data Type: Specifies the data packet type. The valid types are:

Value	Data Packet Type
1	Original ADU
2	Retransmitted ADU

FMP Address of Original Sender: The address of FMP process that original send the ADU.

ADU Sequence Number: Specifies the ADU sequence number. The ADU sequence number counter starts at 0. The first ADU packet the source sends has a sequence number of 0. Because sequence number are 32 bit length, they have a valid range of 0 to $2^{32} - 1$. And because this range is finite, all arithmetic and comparison with these numbers must be modulo 2^{32}. FMP requires that no more than 2^{32} packets can be created over a period equal to the maximum lifetime of a datagram packet in the network.

ADU Length: Specifies the length of the ADU. Normally all ADU except the last one has the same value.

Data: The data to be delivered.

A.3 FMP Control Header

A.3.1 FMP Control Fixed Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Control Type  |      Control Info Length      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Control Info                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Control Type: Specifies the control packet type. The valid types are:

Value	Control Packet Type
0-1	Reserved
2	Feedback
3	End of ADU
4	Reset

Control Info Length: Specifies the length of the Control Info part.

FMP Feedback Control Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                FMP Address of Original Sender                 |
+                                                               +
|              (IP address + data port + ctrl port)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               ADU Cumulative ACK Sequence Number              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Last ADU Sequence Number of the Gap             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              FMP Address of Last Feedback Sender              |
+                                                               +
|              (IP address + data port + ctrl port)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Delta Time Interval                      |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

FMP Address of Original Sender: The address of FMP process that the Feedback is for.

ADU Cumulative ACK sequence number: Specify the cumulative ADU sequence number of this FMP process. This information will be use in FMP reliability mechanism (see section 3.2.4 for details).

Last ADU Sequence Number of the Gap: Specify the last ADU sequence number of the Gap. This information will be use in FMP reliability mechanism (see section 3.2.4 for details).

FMP Address of Last Feedback Sender: The control address of FMP process that last sent the Feedback. This information will be use in FMP RTT measurement mechanism (see section 3.2.5 for details).

Delta Time Interval: Specifies the time interval between the time that received last feedback. This information will be use in FMP RTT measurement mechanism (see section 3.2.5 for details).

FMP End of ADU Control Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              FMP Address of Last Feedback Sender              |
+                                                               +
|              (IP address + data port + ctrl port)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   End of ADU Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

FMP Address of ADU Source: The address of FMP process that send ADUs.

End of ADU Sequence Number: Specify the last ADU sequence number.

FMP Reset Control Header

The control type Reset has no control information.

Appendix B

FMP Header Files

B.1 FMP External Header Files

B.1.1 fmp.h

```
/*
 * fmp.h - Definitions for FMP. (application use only)
 */

#ifndef _FMP_H_
#define _FMP_H_

#include    <sys/time.h>
#include    <sys/types.h>

#include    "fmp_types.h"
#include    "fmp_consts.h"
#include    "fmp_error.h"

#include    "boolean.h"
#include    "dbg_utils.h"

/* FMP transport layer protocol depentdent host address. */
#ifdef  FMP_UDP
#include    "fmp_udp.h"
#else
/* don't work for other protocols now */
#endif
#define FMP_ADDRSIZE        (sizeof(FMP_ADDR))

/* FMP group configuration */
typedef struct fmp_grpcfg {
    char    *fmp_grpname;   /* parent group name or address */
    char    *fmp_grpserv;   /* group data service port */
    int     fmp_grpttl;     /* TTL used for sending messages */

    BOOL    fmp_sendonly;   /* if true, don't recv any mesg from anyone */
    BOOL    fmp_metoo;      /* if true, recv my sent mesg too */
} FMP_GRPCFG;
#define FMP_GRPCFGSIZE  (sizeof(FMP_GRPCFG))

/* FMP host configuration */
typedef struct fmp_hostcfg {
    char    *fmp_hostname;  /* interface name or address */
    BOOL    fmp_isRR;       /* want to be RR or not */
} FMP_HOSTCFG;
#define FMP_HOSTCFGSIZE (sizeof(FMP_HOSTCFG))

/* FMP application configuration */
typedef struct fmp_appcfg {
    FILE    *fmp_infp;          /* input file pointer */
    int     (*fmp_input)        /* input routine */
#ifndef FREEBSD
            (int, FILE *, int *, char *, int);
#else
            __P((int, FILE *, int *, char *, int));
```

```
#endif
    FILE    *fmp_outfp;         /* output file pointer */
    int     (*fmp_output)       /* output routine */
#ifndef FREEBSD
            (FMP_ID, int, char *, int);
#else
            __P((FMP_ID, int, char *, int));
#endif

    int     fmp_adulen;         /* application data unit length */
    int     fmp_ackmapsize;     /* ack bitmap size */
    int     fmp_nadubuf;        /* number of ADU buffers use */

    int     fmp_fbto;           /* feed back timeout (sec) */
    int     fmp_numadu;         /* number of ADU for tolerance check */
    int     fmp_maxadulost;     /* max allowed ADU lost per fmp_numadu */
    int     fmp_maxaduburst;    /* max allowed ADU burst */

    int     fmp_quitwait;       /* Quit waiting time (sec) */
} FMP_APPCFG;
#define FMP_APPPCFGSIZE (sizeof(FMP_APPCFG))

/* FMP protocol configuration from application */
typedef struct fmp_cfg {
    FMP_HOSTCFG     *fmp_hostcfg;
    FMP_GRPCFG      *fmp_pgrpcfg;
    FMP_GRPCFG      *fmp_grpcfg;
    FMP_APPCFG      *fmp_appcfg;
} FMP_CFG;
#define FMP_CFGSIZE         (sizeof(FMP_CFG))

/* FMP protocol configuration from application */
typedef struct fmp_hostinfo {
    FMP_ID      fmp_hostid;
    char        *fmp_hostname;
    FMP_ADDR    fmp_hostaddr;       /* host's FMP address */
} FMP_HOSTINFO;
#define FMP_HOSTINFOSIZE    (sizeof(FMP_HOSTINFO))

/* FMP Statistics */
typedef struct fmp_stat {
    struct timeval  fmp_alltime;
    struct timeval  fmp_sendtime;
    struct timeval  fmp_recvtime;
    double      fmp_sthput;
    double      fmp_rthput;
    double      fmp_asthput;
    double      fmp_arthput;

    int     fmp_nhosts;

    /* all #byte send/recv not include data/ctrl HDR */
    int     fmp_adu_nsend;          /* include # resend */
    int     fmp_adu_nrecv;
    int     fmp_adu_nresend;
```

```
    int     fmp_adu_ndup;
    int     fmp_adu_ndisc;

    int     fmp_data_nbytesend;     /* include #byte resend */
    int     fmp_data_nbyterecv;
    int     fmp_data_nbyteresend;
    int     fmp_data_nbytedup;
    int     fmp_data_nbytedisc;

    int     fmp_ctrl_npktsend;
    int     fmp_ctrl_npktrecv;
    int     fmp_ctrl_nbytesend;
    int     fmp_ctrl_nbyterecv;
} FMP_STAT;
#define FMP_STATSIZE    (sizeof(FMP_STAT))

FMP_STAT    fmp_stat;

/* constant */
#define FMP_EOD     -2

/* error status */
#define FMP_ERR     -1

/* function prototype */
int fmp_init(FMP_CFG *);
void fmp_close();
int fmp_send(char *, int);
int fmp_recv(char *, int);
FMP_HOSTINFO * fmp_gethostinfo(FMP_ID);
void fmp_loop();

/* global variables */
extern FMP_STAT     fmp_stat;

#endif
```

B.1.2 fmp_consts.h

```
/*
 * fmp_consts.h - Definitions for FMP constants.
 */

#ifndef _FMP_CONSTS_H_
#define _FMP_CONSTS_H_

#define FMP_VER_1           1           /* FMP ver 1.0 */
#define FMP_CURRENT_VER     FMP_VER_1   /* the current version of FMP */

#define FMP_MAXHOSTNAME     64
#define FMP_MAXSERVNAME     32

/*
 * FMP data packet send/recv buffer structure  (all are in network byte order)
 */
```

```
#define FMP_MAXADUBUF   4096
#define FMP_MINADUBUF   8

#define FMP_DEFAULT_FBTO    300     /* feedback timeout 5 min (300 sec) */
#define FMP_DEFAULT_MAXFBTO 30000   /* feedback timeout 30000 msec */
#define FMP_MAXQUITWAIT     60      /* Quit waiting timeout (sec) */
#define FMP_SLEEP_EOADU     1       /* sleep time before sending eoadu */

#define FMP_IDLE_TIME       FMP_SLEEP_EOADU

#define FMP_NOID        -1      /* no FMP ID specified */
#define FMP_EOD         -2      /* end of data */

#define FMP_NOADUSEQ    -1      /* no FMP ADUSEQ specified */
#define FMP_INITADUSEQ  0       /* initial FMP ADUSEQ */

#define FMP_INTR_BUFOK      1

#endif
```

B.1.3 fmp_error.h

```
/*
 * fmp_error.h - Definitions for FMP error, error values.
 */

#ifndef _FMP_ERROR_H_
#define _FMP_ERROR_H_

#define    FMP_OK                0
#define    FMP_ERROR            -1

/* data */
#define FMP_EDATA_DISCARD       -2         /* discard packet */
#define FMP_EDATA_DISCARDME     -3         /* discard my packet */
#define FMP_EDATA_NOSBUF        -4         /* no send data buffer */
#define FMP_EDATA_NORBUF        -5         /* no recv data buffer */
#define FMP_EDATA_NOTINBUF      -6         /* not found data in buffer */
#define FMP_EDATA_DUP           -7         /* discard dup packet */

/* status */
#define FMP_STATUS_OK            -10
#define FMP_STATUS_QUITING       -11
#define FMP_STATUS_QUITOK        -12
#define FMP_STATUS_NOTQUIT       -13

/* function prototypes */
void fmp_perror(char *, int);
char *fmp_strerr(int);

#endif
```

B.1.4 fmp_types.h

```
/*
```

```
 * fmp_types.h - Definitions for FMP data types.
 */

#ifndef _FMP_TYPES_H_
#define _FMP_TYPES_H_

typedef int     FMP_ID;            /* FMP ID number type (> 0) */

/* FMP transport layer protocol depentdent host address. */
#ifdef    FMP_UDP
#include  "fmp_udp.h"
#else
/* don't work for other protocols now */
#endif
#define   FMP_ADDRSIZE        (sizeof(FMP_ADDR))

#endif
```

B.1.5 fmp_udp.h

```
/*
 * fmp_udp.h - Definitions for FMP using UDP protocol.
 */

#ifndef _FMP_UDP_H_
#define _FMP_UDP_H_

#include    <sys/types.h>
#include    <netinet/in.h>

#include    "udp_mcast.h"

/*
 * Maximum UDP datagram size now is 65507.
 * (IP_DGRAM_SIZE - IP_HDR_SIZE - UDP_HDR_SIZE = 65536 - 20 - 8)
 * But this value is depend on the implementation of src and desc.
 * For now use NFS default UDP size.
#define UDP_MAXSIZE     8192
 */
#define ETHERNET_MTU        1500

/*
 * For easiest, now use max adusize depend on the mtu of Ethernet media.
#define MEDIA_MTU           4152
 */
#define MEDIA_MTU           8248

#define IP_HDR_SIZE         20
#define UDP_HDR_SIZE        8
#define FMP_PKTMAXSIZE      (MEDIA_MTU - IP_HDR_SIZE - UDP_HDR_SIZE)

/* use IP address and port for FMP_ADDR */
typedef struct fmp_addr {
    struct  in_addr     fmp_addr;       /* host/group address */
    u_short             fmp_dport;      /* data port */
```

```
    u_short             fmp_cport;      /* ctrl port */
} FMP_ADDR;

/* convert mode use in address conversion function */
#define C_AD        1   /* addr + dport convert mode */
#define C_AC        2   /* addr + cport convert mode */
#define C_DONLY     3   /* only dport convert mode */
#define C_CONLY     4   /* only cport convert mode */

#endif
```

B.1.6 udp_mcast.h

```
/*
 * udp_mcast.h - Definitions for IP mulitcast via UDP.
 *
 * $Id: udp_mcast.h,v 1.6 1997-12-11 22:17:58+09 meng Exp $
 */

#ifndef _UDP_MCAST_H_
#define _UDP_MCAST_H_

#include    <stdio.h>

#include    <sys/types.h>       /* many C defs and data type defs */
#include    <sys/socket.h>      /* socket struct defs */
#include    <netinet/in.h>      /* Internet family */

#include    <net/if.h>

#ifdef FMP_BSD
#include    <sys/ioctl.h>
#endif

#ifdef SOLARIS
#include    <sys/sockio.h>
#endif

#include    "sock_utils.h"
#include    "dbg_utils.h"

/* structure for multicast socket descriptors */
typedef struct msfd {
    int     msendfd;        /* socket for sending multicast packet */
    int     mrecvfd;        /* socket for receving multicast packet */
} MSFD;

/* function prototypes */
int mcast_open(MSFD *, char *, char *, char *, int);
int set_mcast_ttl(int, u_char);
int get_mcast_ttl(int, u_char *);
int join_mcast_grp(int, struct in_addr, struct in_addr);
int leave_mcast_grp(int, struct in_addr, struct in_addr);
int mcast_loopback(int, u_char);
int set_mcast_if(int, struct in_addr);
```

```
int mcast_able(int);

#endif
```

B.2 FMP Internal Header Files

B.2.1 fmp_internal.h

```
/*
 * fmp_internal.h - Definitions for FMP. (internal use only)
 */

#ifndef _FMP_INTERNAL_H_
#define _FMP_INTERNAL_H_

#include    <stdio.h>          /* standard input/output library defs */
#include    <sys/types.h>      /* many C defs and data type defs */
#include    <sys/time.h>
#include    <stdlib.h>
#include    <string.h>
#include    <unistd.h>

#include    "dbg_utils.h"
#include    "bit_utils.h"

#include    "fmp.h"
#include    "fmp_utils.h"
#include    "fmp_error.h"

#define FMP_VERSION     FMP_CURRENT_VER     /* the current version of FMP */

/* FMP state */
#define FMP_STATE_UNCONNECT     1
#define FMP_STATE_CONNECTTED    2
#define FMP_STATE_QUITWAIT      3
#define FMP_STATE_QUIT          4
#define FMP_STATE_LOST          5
#define FMP_STATE_SENDWAIT      6

/* FMP packet type */
typedef u_char          FMP_PKTYPE;
#define FMP_DATAPKT     1
#define FMP_CTRLPKT     2

/* FMP Feedback definition */
#define FMP_MAXACKMAP   8192        /* Max ACK map buffer */
#define FMP_ACK         ON
#define FMP_NOACK       OFF

typedef struct fmp_ackmap   {
    int     fmp_mapsize;        /* bitmap size in bit */
    int     fmp_baseseq;        /* base ADU sequence number */
    int     fmp_basemap;        /* base map number */
    u_char  *fmp_map;           /* bitmap */
```

```
} FMP_ACKMAP;

/* FMP group property */
#define FMP_MAXGRP      4   /* Max group entries in group property table */

typedef struct fmp_grprop {
    FMP_ID      fmp_grpid;
    FMP_ID      fmp_hostid;
    char        *fmp_grpname;
    char        *fmp_grpserv;
    int         fmp_grpttl;     /* TTL used for sending data */
    double      fmp_datarel;    /* reliable deadline in usec */

    FMP_ADDR    fmp_addr;       /* multicast group's FMP address */
    MSFD        fmp_datamsfd;   /* data send/recv socket file descriptors */
    MSFD        fmp_ctrlmsfd;   /* ctrl send/recv socket file descriptors */

    BOOL        fmp_sendonly;   /* true if don't want to recv mesg */
    BOOL        fmp_metoo;      /* if true, recv my sent mesg too */
} FMP_GRPROP;
#define FMP_GRPROPSIZE  (sizeof(FMP_GRPROP))

/* FMP packet fixed header */
typedef struct fmp_hdr {
    u_char      fmp_ver;        /* FMP version */
    u_short     fmp_dcport;     /* FMP data or ctrl port */
    struct timeval  fmp_tstmp;  /* FMP data send timestfmp (net byte order) */
} FMP_HDR;
#define FMP_HDRSIZE     (sizeof(FMP_HDR))

#define FMP_HP(fmp_pkt)     ((FMP_HDR *) &((fmp_pkt))[0])

#include    "fmp_data.h"
#include    "fmp_ctrl.h"

/* FMP Host property */
#define FMP_MAXHOST     16      /* Max host entries in host property table */

typedef struct fmp_hostprop {
    FMP_HOSTINFO    fmp_hostinfo;
#define fmp_hid     fmp_hostinfo.fmp_hostid
#define fmp_hname   fmp_hostinfo.fmp_hostname
    FMP_ADDR        fmp_addr;       /* host's FMP address */

    struct timeval  fmp_rtt;
    struct timeval  fmp_lastseen;

    u_char      fmp_minttl;
    u_char      fmp_maxttl;
    u_char      fmp_curttl;

    int         fmp_needsfb;    /* need to send feedback */
    FMP_ADUSEQ  fmp_maxaduseq;  /* maximum ADU seq got */
    FMP_FB      fmp_fb;         /* feedback ctrl pkt for this host */
    FMP_EOADU   fmp_eoadu;      /* eoadu ctrl pkt for this host */
```

```
} FMP_HOSTPROP;
#define FMP_HOSTPROPSIZE    (sizeof(FMP_HOSTPROP))

/* FMP application property */
typedef struct fmp_approp {
    int     fmp_infd;           /* input file descriptor */
    FILE    *fmp_infp;          /* input file pointer */
    int     (*fmp_input)        /* input routine */
#ifndef FREEBSD
        (int, FILE *, int *, char *, int);
#else
        __P((int, FILE *, int *, char *, int));
#endif

    int     fmp_outfd;          /* output file descriptor */
    FILE    *fmp_outfp;         /* output file pooutter */
    int     (*fmp_output)       /* output routoute */
#ifndef FREEBSD
            (FMP_ID, int, char *, int);
#else
            __P((FMP_ID, int, char *, int));
#endif

    int     fmp_adulen;         /* application data unit length */
    int     fmp_maxadulost;     /* max allowed ADU lost per fmp_numadu */
    int     fmp_fbto;           /* feed back timeout (msec) */

    FMP_ACKMAP  fmp_ackmap;
    int     fmp_nadubuf;        /* number of ADU buffers use */

    int     fmp_quitwait;       /* quiting waiting time */
} FMP_APPROP;
#define FMP_APPROPSIZE  (sizeof(FMP_APPROP))

/* FMP protocol property */
typedef struct fmp_prop {
    FMP_ID      fmp_pgrpid;
    FMP_ID      fmp_grpid;
    FMP_ID      fmp_hostid;
    FMP_APPROP  fmp_approp;
    char        fmp_curstate;
    BOOL        fmp_isRR;

    TIMER       fmp_alltimer;
    TIMER       fmp_sendtimer;
    TIMER       fmp_recvtimer;

    int         fmp_pid;
    int         fmp_event;
    int         fmp_rxtqlen;
} FMP_PROP;
#define FMP_PROPSIZE    (sizeof(FMP_PROP))

#define FMP_MYHID       (fmp_prop->fmp_hostid)  /* my host id */
#define FMP_MYADDR      (fmp_hostbl[FMP_MYHID]->fmp_addr)   /* my host addr */
```

```
/* function prototypes */
void fmp_initrand();
FMP_ID fmp_inithost(FMP_HOSTCFG *);
FMP_ID fmp_initgrp(FMP_ID, FMP_GRPCFG *);
/*
FMP_IP * fmp_gethostpropbyid(FMP_ID);
 */
FMP_ID fmp_gethostidbyaddr(FMP_ADDR *);
FMP_ID fmp_newhostprop(FMP_ADDR *);

int fmp_mcast_open(FMP_GRPROP *);
int fmp_mcast_sendto(FMP_PKTYPE, FMP_GRPROP *, char *, int);
int fmp_mcast_recvfrom(FMP_PKTYPE, FMP_GRPROP *, FMP_ID *, char *, int);
int fmp_mcast_close(FMP_GRPROP *);
char * fmp_prtaddr(FMP_ADDR *);
char * fmp_hdr_prt(FMP_HDR *);
void fmp_calstat();
void fmp_prtstat();

/* global variables */
extern FMP_ADDR     orig_grp_addr;  /* orginal multicast group's addr */
extern FMP_ADDR     pgrp_addr;      /* multicast parent group's addr */
extern FMP_ADDR     grp_addr;       /* multicast group's addr */
extern FMP_ADDR     if_addr;        /* interface's addr */

extern FMP_PROP     *fmp_prop;                    /* FMP's property */
extern FMP_GRPROP   *fmp_grptbl[FMP_MAXGRP + 1];  /* FMP group property table */
extern FMP_HOSTPROP *fmp_hostbl[FMP_MAXHOST + 1]; /* FMP host property table */
extern int      fmp_nhost;      /* number of hosts in host prop table */

extern int      wantoclose;     /* last aduseq of data */

#endif
```

B.2.2 fmp_ctrl.h

```
/*
 * fmp_ctrl.h - Definitions for FMP control packet. (internal use only)
 */

#ifndef _FMP_CTRL_H_
#define _FMP_CTRL_H_

#include    "fmp.h"

/* FMP control type */
typedef u_char          FMP_CTRLTYPE;
#define FMP_CTRLACK     1
#define FMP_CTRLFB      2
#define FMP_CTRLEOADU   3
#define FMP_CTRLRST     4

/* FMP control header definition */
typedef struct fmp_ctrlhdr {
```

```
    FMP_CTRLTYPE    fmp_ctype;  /* FMP control type */
    u_short         fmp_len;    /* FMP ctrl length (not include fmp_ctrlhdr) */
} FMP_CTRLHDR;
#define FMP_CTRLHDRSIZE (sizeof(FMP_CTRLHDR))

#define FMP_CTRL_HP(fmp_pkt)    ((FMP_CTRLHDR *) &((fmp_pkt))[FMP_HDRSIZE])
#define FMP_CTRL_DP(fmp_pkt)    ((u_char *) &((fmp_pkt))[FMP_HDRSIZE + FMP_CTRLHDRSIZE])

/* FMP control feedback definition */
typedef struct fmp_fb {
    u_char          fmp_got1stpkt;  /* true, if got some data */
    FMP_ADDR        fmp_srcaddr;    /* address of this host */
    FMP_ADUSEQ      fmp_cack;       /* cumulative ACK */
    FMP_ADUSEQ      fmp_gapend;     /* seq gap end seq  */
    struct timeval  fmp_tss;        /* time that sender sent data */
    struct timeval  fmp_trr;        /* time that receiver received data */
    struct timeval  fmp_trs;        /* time that receiver send feedback */
} FMP_FB;
#define FMP_FBSIZE  (sizeof(FMP_FB))

/* FMP control feedback definition */
typedef struct fmp_eoadu {
    FMP_ADDR        fmp_srcaddr;    /* mesg address of this host */
    FMP_ADUSEQ      fmp_eoaduseq;   /* End of ADU seq  */
} FMP_EOADU;
#define FMP_EOADUSIZE   (sizeof(FMP_EOADU))

/* FMP retransmission queue entry */
typedef struct fmp_rxt {
    FMP_ADUSEQ      fmp_gapbeg;     /* gap start ADU seq  */
    FMP_ADUSEQ      fmp_gapend;     /* gap end ADU seq  */
} FMP_RXT;
#define FMP_RXTSIZE     (sizeof(FMP_RXT))
#define FMP_MAXRXTENT       8

#define FMP_MAXADURXT       50
#define FMP_MINADURXT       5

/* function protypes */
int fmp_ctrinit();
int fmp_ctrlhdr_fill(FMP_CTRLHDR *, FMP_CTRLTYPE, int);

int fmp_sendctrl(FMP_ID, FMP_CTRLTYPE, char *, int);
int fmp_sendack(FMP_ID, FMP_ADUSEQ);
int fmp_sendfb(FMP_ID, FMP_ID);
int fmp_sendeoadu();

void fmp_mapack(int, int);

int fmp_recvctrl(FMP_ID, FMP_ID *);
int fmp_procack(FMP_ID, FMP_ADUSEQ *);
int fmp_procfb(FMP_ID, FMP_FB *);

int aduseq2bmpid(FMP_ADUSEQ);
int fmp_gotack(FMP_ADUSEQ);
```

```
void fmp_prt_ctrlpkt(char *, u_char *);
void fmp_prt_ctrlhdr(char *, FMP_HDR *, FMP_CTRLHDR *);
void fmp_prt_ackpkt(u_char *);
void fmp_prt_fbpkt(FMP_FB *);

void fmp_prtackmap(FMP_ACKMAP *, int, int);

#endif
```

B.2.3 fmp_data.h

```
/*
 * fmp_data.h - Definitions for FMP data packet. (internal use only)
 */

#ifndef _FMP_DATA_H_
#define _FMP_DATA_H_

#include    "fmp_types.h"
#include    "fmp_error.h"
#include    "fmp_consts.h"

#include    "fmp_internal.h"

/* FMP data type */
typedef u_char          FMP_DATATYPE;
#define FMP_DATAORIG    1       /* original data packet */
#define FMP_DATARXT     2       /* retransmission data packet */

/*
 * FMP data packet header structure
 */
typedef struct fmp_datahdr {
    FMP_DATATYPE    fmp_dtype;      /* FMP data packet type */
    FMP_ADDR        fmp_orgsource;  /* FMP address of original source */
} FMP_DATAHDR;
#define FMP_DATAHDRSIZE (sizeof(FMP_DATAHDR))

/*
 * FMP application data unit header structure
 */
typedef short   FMP_ADUSEQ;     /* FMP ADU sequence number type */
typedef u_short FMP_ADULEN;     /* FMP message length type */

typedef struct fmp_aduhdr {
    FMP_ADUSEQ  fmp_aduseq;     /* FMP adu seq number (net byte order */
    FMP_ADULEN  fmp_adulen;     /* FMP data length (net byte order) */
} FMP_ADUHDR;
#define FMP_ADUHDRSIZE      (sizeof(FMP_ADUHDR))

#define FMP_ADUDATAMAXSIZE  (FMP_PKTMAXSIZE - FMP_HDRSIZE - FMP_DATAHDRSIZE - FMP_ADUHDRSIZE)

/* head pointer finding macros */
#define FMP_DATA_HP(fmp_pkt)    ((FMP_DATAHDR *) &((fmp_pkt)[FMP_HDRSIZE]))
```

```
#define FMP_ADU_HP(fmp_pkt)     ((FMP_ADUHDR *) &((fmp_pkt)[FMP_HDRSIZE + FMP_DATAHDRSIZE]))
#define FMP_ADU_DP(fmp_pkt)     (&((fmp_pkt)[FMP_HDRSIZE + FMP_DATAHDRSIZE + FMP_ADUHDRSIZE]))

/* FMP default ADU buffer values */
#define FMP_MAXADUBUFTIME       1   /* max time adubuf in adubuflist (sec) */

typedef struct fmp_adubuf {
    FMP_ID          fmp_srcid;      /* ADU's original source FMP ID */
    struct timeval  fmp_tstmp;      /* send/recv timestfmp */
    FMP_ADUHDR      fmp_aduhdr;     /* adu header */
    u_char          *fmp_adudata;   /* adu data */
    struct fmp_adubuf   *next;
} FMP_ADUBUF;
#define FMP_ADUBUFSIZE  (sizeof(FMP_ADUBUF))

typedef struct fmp_adubuflist {
    FMP_ADUBUF  *usehead;       /* head pointer of use list */
    FMP_ADUBUF  *usetail;       /* tail pointer of use list*/
    int         nusebuf;        /* number of buffers in use list */
    FMP_ADUBUF  *freehead;      /* head pointer of free list */
    FMP_ADUBUF  *freetail;      /* tail pointer of free list */
    int         nfreebuf;       /* number of buffers in free list */
    FMP_ADUBUF  **list;         /* buffer list */
} FMP_ADUBUFLIST;

/* function prototypes */
int fmp_datainit();
int fmp_datapkt_init(int);
int fmp_adubuflist_init(FMP_ADUBUFLIST *, int);
int fmp_datahdr_fill(FMP_DATAHDR *, FMP_DATATYPE, FMP_ID);

int fmp_adubuflist_empty(FMP_ADUBUFLIST *);
FMP_ADUBUF *fmp_adubuf_alloc(FMP_ADUBUFLIST *, int);
FMP_ADUBUF *fmp_adubuf_find(FMP_ADUBUFLIST *, FMP_ID, FMP_ADUSEQ);
FMP_ADUBUF *fmp_adubuf_findbef(FMP_ADUBUFLIST *, FMP_ID, FMP_ADUSEQ);
int fmp_adubuf_free(FMP_ADUBUFLIST *, FMP_ID, FMP_ADUSEQ);

int fmp_sendadu(FMP_ADUSEQ, char *, int);
int fmp_senddata(FMP_ID, FMP_DATATYPE, FMP_ID, FMP_ADUHDR *, char *);

int fmp_recvadu(FMP_ID *, FMP_ADUSEQ *, char *, int);
int fmp_recvdata(FMP_ID, FMP_ID *, FMP_ADUHDR *, char *);

void fmp_upfb(FMP_ID, struct timeval *, FMP_ADUSEQ);

#ifdef DEBUG
void fmp_prtdatapkthdr(char *, FMP_HDR *, FMP_DATAHDR *, FMP_ADUHDR *);
void fmp_prtadubuf(FMP_ADUBUF *);
void fmp_prtadubuf2(FMP_ADUBUFLIST *, FMP_ID, FMP_ADUSEQ);
#endif

/* global variables */
extern FMP_ADUBUFLIST   fmp_sendbuflist;
extern FMP_ADUBUFLIST   fmp_recvbuflist;
```

```
#endif
```

B.2.4 fmp_utils.h

```
/*
 * fmp_utils.h - Definitions for FMP utility functions.
 */

#ifndef _FMP_UTILS_H_
#define _FMP_UTILS_H_

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/time.h>

#include    "dbg_utils.h"
#include    "time_utils.h"

void itoa(int, char *);

double fmp_rtt(struct timeval *, struct timeval *, struct timeval *, struct timeval *);
void fmp_tstmp(struct timeval *);

void hton_tval(struct timeval *);
void ntoh_tval(struct timeval *);

#endif
```

B.3 FMP Utility Header Files

B.3.1 bit_utils.h

```
/*
 * bit_utils.h - Definitions for bitmap manipulate functions.
 */

#ifndef _BIT_UTILS_H_
#define _BIT_UTILS_H_

#include    <sys/types.h>
#include    <stdio.h>
#include    <stdlib.h>

#include    "dbg_utils.h"

#define NPB     8           /* number of bits per u_char */

#define ON      0x1
#define OFF     0x0

#define bit_i(bmp, i)   ((bmp)[i / NPB] >> (NPB - 1 - (i % NPB)) & 0x1)

u_char *bit_alloc(int);
```

```
int bit_ison(u_char *, int);
int bit_isoff(u_char *, int);

u_char bit_set(u_char *, int, u_char);
u_char bit_inv(u_char *, int);

u_char bit_on(u_char *, int);
u_char bit_off(u_char *, int);

#endif
```

B.3.2 boolean.h

```
/*
 * boolean.h - Definitions for boolean data type.
 */

#ifndef _BOOLEAN_H_
#define _BOOLEAN_H_

#include    <sys/types.h>

/* boolean data type and value */
typedef u_char      BOOL;

#ifndef TRUE
#define TRUE    1
#endif

#ifndef FALSE
#define FALSE   0
#endif

#endif
```

B.3.3 dbg_utils.h

```
/*
 * dbg_utils.h - Definitions for debug utility functions.
 */

#ifndef _DBG_UTILS_H_
#define _DBG_UTILS_H_

#include    <stdarg.h>
#include    <stdio.h>
#include    <time.h>
#include    <sys/time.h>
#include    <errno.h>

#include    "boolean.h"

#define hdr_print(fp, hdr, str)     fprintf((fp), "%s %s: ", (hdr), (str))

/* function prototypes */
```

```
void va_print(FILE *, const char *, va_list);

void warn(char *, const char *fmt, ...);
void debug(int, char *, const char *fmt, ...);
void inerr(char *, const char *fmt, ...);
void err(char *, const char *fmt, ...);

void log(const char *fmt, ...);
void vbose(const char *fmt, ...);

/* global variables */
extern int      errno;

extern FILE     *logfile;       /* log file pointer */
extern BOOL     debugflg;       /* view debug mesg or not */
extern BOOL     vboseflg;       /* verbose or not */
extern int      debuglevel;     /* debug level */

#endif
```

B.3.4 sock_utils.h

```
/*
 * sock_utils.h - Definitions for socket interface utility functions.
 */

#ifndef _SOCK_UTILS_H_
#define _SOCK_UTILS_H_

#include    <sys/types.h>       /* many C defs and data type defs */
#include    <sys/socket.h>      /* socket struct defs */
#include    <netinet/in.h>      /* Internet family */
#include    <arpa/inet.h>       /* for address conversion routine */
#include    <netdb.h>           /* for get network host entry routine */
#include    <stdio.h>           /* standard input/output library defs */
#include    <unistd.h>          /* for gethostname */

#include    "dbg_utils.h"

#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff  /* should be in <netinet/in.h> */
#endif

/* function prototypes */
int fill_inetsock(struct sockaddr_in  *, char *, char *, char *);

#ifdef  DEBUG
char    *prt_inetaddr(struct sockaddr_in *);
#endif

#endif
```

B.3.5 time_utils.h

```
/*
```

```
 * time_utils.h - Definitions for time manipulate functions.
 */

#ifndef _TIME_UTILS_H_
#define _TIME_UTILS_H_

#include    <stdio.h>

#include    <sys/types.h>
#include    <sys/time.h>
#include    <sys/resource.h>
#include    <unistd.h>
#include    <signal.h>

#include    "dbg_utils.h"

/* timer data structure */
typedef struct timer {
    struct timeval  start;  /* start time for real time */
    struct timeval  stop;   /* stop time for real time */
    struct rusage   ru_start;   /* start time for user & sys time */
    struct rusage   ru_stop;    /* stop time for user & sys time */
} TIMER;

#define t_tval2usec(tvalp)  (((double) (tvalp)->tv_sec) * 1000000.0 + (tvalp)->tv_usec)
/*
 * Fill millisec to struct timeval variable.
 */
#define t_ms2tval(tms, tval)    t_usec2tval(((tms) * 1000), (tval))

void t_init(TIMER *);

void t_start(TIMER *);
void t_stop(TIMER *);

void t_getutime(TIMER *, struct timeval *);
void t_getstime(TIMER *, struct timeval *);
void t_getrtime(TIMER *, struct timeval *);

void tv_sub(struct timeval *, struct timeval *, struct timeval *);
void t_usec2tval(long, struct timeval *);

int t_sleep(struct timeval *);
int t_alarm(int, struct timeval *);

#endif
```

Bibliography

[1] A.Erramilli and R.P.Singh. A Reliable and Efficient Multicast Protocol for Broadband Broadcast Networks. In *Proc. ACM SIGCOMM '87*, Aug. 1987.

[2] C.Hedrick. Routing Information Protocol. *RFC 1058*, June 1988.

[3] D.D. Clark and D.L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM'90*, pages 200–208. ACM, September 1990.

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

[5] Sandeep K. Singhal Hugh W. Holbrook and David R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proc. ACM SIGCOMM '95*, Aug. 1995.

[6] IP Multicast Initiative (IPMI). IP Multicast Backgrounder. In *http://www.ipmulticast.com/community/whitepapers/backgrounder.html*, 1997.

[7] J.M.Chang and N.F.Maxemchuk. Reliable broadcast protocols. 2(3):251–273, August 1984.

[8] J.Moy. Multicast Extensions to OSPF. *RFC 1584*, March 1994.

[9] J.Moy. OSPF Version 2. *RFC 1583*, March 1994.

[10] J.Postel. Transmission Control Protocol. *RFC 793*, September 1981.

[11] J.Postel. RFC 768: User Datagram Protocol. In *Request For Comments*, Aug. 1980.

[12] J.R.Cooperstock and S.Kotsopoulos. Why use a fishing line when you have a net? an adaptive multicast data distribution protol. In *To Appear in Proceedings of Usenix '96*, 1996.

[13] A. Koifman and S. Zabele. RAMP: A Reliable Adaptive Multicast Protocol. In *IEEE INFOCOM '96*, Mar. 1996.

[14] Y.Lepage K.Savetz, N.Randall. *MBONE:Multicasting Tomorrow's Internet.* IDG Books Worldwide, Inc., USA, 1996.

[15] John C. Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *IEEE INFOCOM '96*, Mar. 1996.

[16] J.Macker M.S.Corson. Reliable Scalable Multicast and a Self-Organizing Approach for Long-Lived Sessions. *IEEE Journal on Selected Areas in Communications*, January 1996.

[17] N.Shacham and J.S.Meditch. An Algorithm for Optimal Multicast of Multimedia Streams. *IEEE INFOCOMM '94*, June 1994.

[18] J.Griffioen R.Yavatkar and M.Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *Proc. ACM Multimedia '95*, Nov. 1995.

[19] J. Kurose S. Kasera and D. Towsley. Scalable Reliable Multicast Using Multiple Multicast Groups. In *Proc. ACM SIGMETRICS '97*, 1997.

[20] V. Jacobson S. McCanne and M. Vetterli. Receiver-driven Layered Multicast. In *Proc. ACM SIGCOMM '96*, Aug. 1996.

[21] Towsley S. Pingali and J. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *Proc. ACM SIGMETRICS '94*, 1994.

[22] S.Deering. Host Extension for IP Multicasting. *RFC 1112*, August 1989.

[23] S.McCanne C.Liu S.Floyd, V.Jacobson and L.Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, Nov. 1996.

[24] S.P.Morgan S.Keshav. SMART Retransmission: Performance with Overload and Random Losses. In *IEEE INFOCOM '97*, 1997.

[25] W.Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Publishing Company, USA, 1996.

[26] TASC. Reliable multicast protocols. In *http://www.tascnets.com/mist/doc/mcpCompare.html*, Aug. 1996.

[27] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing Network Protocols at User Level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.

[28] V.Kumar. *MBONE-Interactive Multimedia on the Internet.* New Riders Publishing, Indianapolis, Indiana, USA, 1996.