RELIABLE MULTICAST TRANSPORT PROTOCOL IMPLEMENTATION BUILDING BLOCKS

Tian Fang

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2004

Abstract

Reliable Multicast Transport Protocol Implementation Building Blocks

Tian Fang

In order to provide reliable IP multicast services, multiple Reliable Multicast Transport Protocols (RMTPs) have been developed. However, because there exist different definitions of "reliability", no single RMTP can meet all requirements of different applications. The research focus of RMTPs has changed from individual protocols to the protocol building blocks. This change of focus provides impetus for this thesis describing RMTP implementation building blocks.

The Meta-Transport Library (MTL) was designed to implement RMTPs. Nonetheless, due to its limits, the MTL is insufficient to be used as the basis for the RMTP implementation building blocks.

The RMTP implementation building blocks presented in this thesis provide a framework and a full set of common components that can be used to implement a wide range of RMTPs.

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. J. William Atwood, for his contribution in supervising this thesis. It was largely because of his careful supervision, insightful scholarly guidance, and financial support that made it possible for me to complete this thesis.

My special thanks are also due to my parents and my wife who in various ways helped me keep moving towards the completion of this thesis.

Contents

List of Figures

List of Acronyms

ACK	Acknowledge
ALC	Asynchronous Layered Coding
FEC	Forward Error Correction
FIFO	First In First Out
FSM	Finite State Machine
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPC	Inter-Process Communication
MBone	Multicast Backbone
MOSPF	Multicast Extensions to OSPF
MTL	Meta-Transport Library
NACK	Non-acknowledge
OS	Operating System
OSPF	Open Shortest Path First
PDU	Packet Data Unit
PIM-DM	Protocol Independent Multicast - Dense Mode
PIM-SM	Protocol Independent Multicast - Sparse Mode
RFC	Request for Comments
RMTP	Reliable Multicast Transport Protocol
RMTP-II	Reliable Multicast Transport Protocol - II

RMTWG	Reliable Multicast Transport Working Group
TCP	Transmission Control Protocol
TVLV	Type, Version, Length, and Value
UDP	User Datagram Protocol
XTP	Xpress Transport Protocol

Chapter 1

Introduction

1.1 Multicast vs. Unicast

Similar to the Transmission Control Protocol (TCP) [Pos81b] or the User Datagram Protocol (UDP) [Pos80], most network transport protocols only provide a unicast transmission service. Therefore, network hosts are only able to send data to one other node at a time. We describe this unicast transmission service as being point-to-point. If a host wants to transfer the same information to N other hosts (a situation referred to as one-to-many) with unicast, it must send the data N times.

Two better ways to transmit data from one source to many destinations are: provide a broadcast transport service, or provide a multicast service. With either a broadcast or a multicast transport service, a single node can send data to many destinations by making just a single call on the transport service.

In a broadcast transport service, a single node sends data to a specified broadcast address. All destinations in the same subnet will receive the data. No host can refuse to receive broadcast data.

For a multicast transport service, a single node sends data to a specified multicast session group. Those hosts who want to receive the data need to join that group before they can receive data specific to the group. Only hosts interested in the data will receive the data. Multicast is a receiver-based concept; so it is the receiver's responsibility to decide whether or not to join a particular multicast session group. The sender to a multicast session group knows nothing about receivers. Traffic to a group is delivered to all members of that multicast session group. It is not necessary for the sender to maintain a list of receivers. Only one copy of a multicast message will pass through any link in the network, and copies of the message will be made only where paths diverge at a router. Thus multicast yields many performance improvements and conserves bandwidth end-to-end. It is also more efficient than broadcasting one copy of the message to all nodes on the network, since many nodes may not want the message.

Many Internet applications are one-to-many or even many-to-many, where one or several sources want to send information to multiple receivers at once. Some examples are: distribution of stock prices to every individual, video and/or audio conferencing for remote meetings, replicating multiple databases and mirroring web site information. For these applications, which involve a single node sending information to multiple destinations, a multicast transport service is assuredly a better choice than a unicast transport service. By sending only a single copy of a message to multiple destinations who explicitly want to receive the message, multicast is much more efficient. For unicast, which requires the source to send a copy of a message to each individual requester, the bandwidth available to the sender limits the number of receivers.

Internet Protocol (IP) multicast services are based on raw IP [Pos81a] or on UDP. Multicast was introduced in 1989 in the Ph.D dissertation of Steve Deering, and then standardized as RFC 1112. Multicasting is described as follows: "the transmission of an IP datagram to a 'host group', a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same 'best-efforts' [*sic*] reliability as regular unicast IP datagram. The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time." [Dee89]

Deering's work led to the creation of multicast communication in the Internet and the creation of the Multicast Backbone (MBone) [Eri94]. Research into the routing of multicast packets within the MBone has been carried out in order to extend some existing unicast routing protocols. MOSPF [Moy94], Multicast Extensions to OSPF, uses the particular mechanisms of the Open Shortest Path First (OSPF) [Moy98] protocol to provide multicast. Protocol Independent Multicast - Sparse Mode

(PIM-SM) [EFH+98] and Protocol Independent Multicast - Dense Mode (PIM-DM) [DEF+03] have also been developed.

From the Packet Data Unit's (PDU) point of view, the only difference between a multicast IP packet and a unicast IP packet is the destination address. IP addresses have been divided into five classes according to address range. Class D address range (224.0.0.0–239.255.255.255) is reserved for the multicast purpose usage. Class A (0.0.0.0–127.255.255.255), B (128.0.0.0–191.255.255.255), and C (192.0.0.0–223.255.255.255) are reserved as unicast IP address classes. One multicast address represents one multicast session group, which includes one or more senders and zero or more receivers. A sender sends data to a multicast IP address as the destination address for that data. In order to receive the data sent to this group, receivers must join this multicast IP address group.

1.2 Reliable Multicast Transport Protocols

Since multicast packets are raw IP or UDP packets, they are unreliable. Reliable Multicast Transport Protocols (RMTPs) are designed to provide "TCP-like" multicast services.

In order to provide "reliable" transmission service on the basis of raw IP or UDP protocol, RMTPs need to develop various mechanisms pertaining to "reliability". Unlike for unicast, which is point-to-point transmission, for multicast the number of possible ways the packets can be lost or damaged is much larger. A definition of "reliable" transmission can vary for different application requirements.

- Best-effort reliability

  As its name indicates, this type of reliability provides the best-effort service. UDP is a good example to provide this kind of reliability. When a corrupted packet is received at the receiver, it may or may not be sent to the user. There is no necessity to have Acknowledge (ACK), Non-acknowledge (NACK), or resend mechanisms.

- Bounded-latency reliability

  Every packet is said to be "alive" only for a specified lifetime. Within this time, the data are considered useful to the user. Otherwise, they can be discarded.

A video or audio stream is the best example of an application using this kind of reliability.

- Most-recent reliability

  For this type of reliability, the most recent data should be guaranteed. An application using these data is only interested in the most up-to-date data. An example of such an application is a stock quote, wherein only the latest quote is considered useful.

- Receiver-centered reliability

  In this case, the receiver determines whether or not the data should be re-sent. Furthermore, the sender has no knowledge of the success of the delivery. It may be a receiver's responsibility to re-send the lost or damaged data to other receivers. A good example is satellite transmission, which has only downstream transmission. Since there is no upstream to ask the sender to do retransmission, receivers themselves have to be able to guarantee the reliability.

- Absolute reliability

  This type of reliability means that all the data should be received by all receivers, in order and completely, or none of the data should be used by any receiver.

Some RMTPs may provide more than one kind of reliability according to the protocol configuration. For example, the Reliable Multicast Transport Protocol - II (RMTP-II) [WBP$^+$98a, WBP$^+$98b] can provide bounded-latency reliability and receiver-centered reliability for different applications.

Besides defining what kind(s) of reliability can be supported by an RMTP, the primary challenge all RMTPs are facing is how to scale to a large number of receivers. Some protocols use a back-channel for recovery of lost packets, while others have found that coded data are very beneficial to achieve good throughput for numerous receivers. RMTPs can be broken into four families according to the above factors:

- NACK only

  SRM [FJM95] and MDP2 [AM99] only use NACKs to request retransmission of lost or damaged packets.

- Tree based ACK

  RMTP [LP96], RMTP-II and TRAM [KCWP00] have a transmission tree to aggregate ACKs. This can avoid ACK explosion for many receivers, achieving good scalability. Moreover, this tree can also be used to reduce the overflow from the sender to re-transfer the lost or damaged data. Some nodes on the tree may take control of the re-transmission data to receivers further down the tree.

- Asynchronous Layered Coding (ALC)

  It is used by [RV97] and [BLMR98]. By using a sender-based Forward Error Correction (FEC) method, no feedback from receivers or the network is required. Therefore, higher throughput can be achieved.

- Router assist

  Extra router software is required to do constrained negative acknowledgements and retransmissions. All protocol families described above can also be combined with this mechanism.

These protocol families are not precise and exclusive to each distinct protocol. Some protocols may use different mechanisms belonging to different classes. For example, RMTP-II uses both tree based ACK and NACK.

Another way to classify RMTPs is according to the number of senders [Atw]:

- 1-N Multicast

  This is the usual way of multicasting. Only one node within the group, the sender, is allowed to transmit the data. All others are the receivers or control nodes.

- M-N Multicast

  A limited subset of the nodes in the group may become the sender.

- N-N Multicast

  Any node within the group may become the sender.

1.3 Motivation

Because there are different definitions of "reliability" for different applications, there is no single RMTP that can meet the needs of all applications. Therefore, the Reliable Multicast Transport Working Group (RMTWG) in the Internet Engineering Task Force (IETF) has changed its research focus from individual protocols to building blocks for all RMTPs, referred to as "protocol building blocks". This change of focus provides impetus for this thesis describing RMTP implementation building blocks.

The Meta-Transport Library (MTL) was designed to implement RMTPs. However, due to its limits, the MTL is insufficient to be used easily as the RMTP implementation building blocks. This thesis presents a complete set of RMTP implementation building blocks that can be used to implement a wide range of RMTPs.

Chapter 2 summarizes the current research of the protocol building blocks and the MTL. The analysis of the requirements of the implementation building blocks is presented in chapter 3. Based on this analysis, chapter 4 illustrates the design for all implementation building blocks. A comparison between the method presented in this thesis and the MTL is shown in chapter 5.

Chapter 2

An Approach to the Design and Implementation of RMTPs

2.1 Protocol Building Blocks

A protocol building block represents a component common to various RMTPs. By combining protocol build blocks according to different requirements, different RMTPs can be constructed. Several Request for Comments (RFC) and Internet Drafts have been published concerning the protocol building blocks.

2.1.1 The Reliable Multicast Design Space for Bulk Data Transfer (RFC2887)

This RFC [HWK$^+$00] studies the following aspects of the reliable multicast design space for bulk data transfer:

- Application constraints

  Different applications have different requirement for the protocols. Application constraints include the definition of "reliability" and receiver set scaling.

- Network constraints

  Network administration, bandwidth, architecture, and assistant ability are network constraints.

- Good throughput mechanisms

  There are some mechanisms that can be used to achieve good throughput, including ACK-based, Tree-based ACK, NACK-based, Packet-level FEC and layered FEC mechanisms.

- Congestion control mechanisms

  Since the Internet provides best-effort service, end-systems are expected to apply congestion control by themselves. Sender-controlled one group, sender-controlled multiple groups, receiver-controlled one group, receiver-controlled layered organization, and router-based congestion control mechanisms can be used.

2.1.2 Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer (RFC3048)

Based on the analysis of reliable multicast design space for bulk data transfer in RFC 2887, a framework for the standardization of bulk-data reliable multicast transport is proposed in RFC 3048 [WVK+01]. This framework is based on the protocol building blocks, which are the common components for multiple protocol classes. The following protocol building blocks are recommended in the RFC:

- NACK-based reliability [ABHM03, ABHM04]
- FEC coding [LVG+02a, LVG+02b, LV04]
- Congestion control [WH03, LG02]
- Generic router support [Cal01, CST01]
- Tree configuration [KWC+01]
- Data security [WH01]
- Common headers
- Protocol cores [LGV+02a, LGV+02b, PLL+03, KWCT01]

2.2 Meta-Transport Library – MTL

The MTL [San96, San97] is a collection of C++ base classes developed at Sandia National Laboratories, California. This set of classes includes many common components used to implement RMTPs. Specific RMTP implementations can be derived from the MTL. Moreover, the architectures of RMTP implementations are also defined within the library.

2.2.1 MTL Use Case

When using the MTL as the basis for implementing an RMTP, the development is separated into two parts [Figure 1]

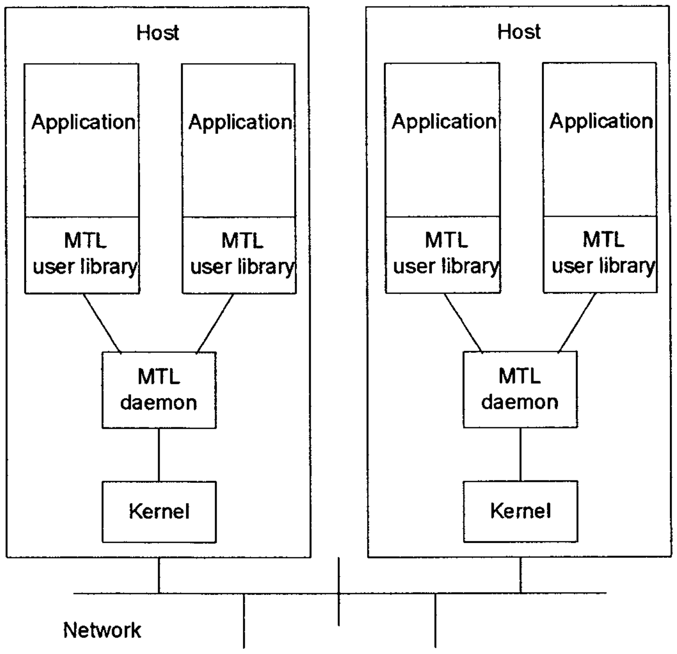


Figure 1: MTL Use Case

The core part of the RMTP implementation work is a stand-alone process, the MTL daemon, which implements all the functionalities of an RMTP. It communicates with other protocol nodes to exchange protocol information or other data, and it also handles all requests sent by any application that uses the RMTP for communication. The daemon hides most of the protocol details from the application.

The other part of an RMTP implementation will be combined into a library, which will be linked by any application that uses the RMTP. This part is referred as the

MTL user library. The purpose of the MTL user library is to encapsulate the details for communication between the application and the MTL daemon.

2.2.2 MTL Classes

- Packets and derived classes

  The information exchanged in a protocol is called a packet. The MTL packet class simply defines a byte array and some methods to manipulate this array. However, the MTL packet class is only an abstract class, and does not contain any knowledge about the kinds of contents found inside the array. In other words, no method provided by the MTL packet class will interpret the contents in the array. The array will be treated as a whole. Other RMTP specific classes provide the methods to insert or retrieve values from the packet.

  There are two supplemental classes in the MTL used for packets. One is the *packet_pool* class for memory management for packet objects and the other is the *packet_fifo* class for the queuing of packets prior to processing.

  The MTL packet class and the two supplemental classes are only used within the MTL daemon.

- Context and context management

  A context is defined in the MTL as a node in the protocol. It communicates both with other nodes in the protocol to exchange packets, and with an application to handle user requests and data. The MTL context class includes all node information used in the protocol, as well as methods for state transitions, packet parsing, and user request handling.

  The MTL context management class contains and maintains all contexts. It is also responsible for dispatching packets or user requests to their corresponding context.

  Any specific RMTP implementation has a subclass derived from the MTL context class. All protocol specific functions are provided by this subclass.

  Both the MTL context and context management classes are used within the MTL daemon.

- Buffer management

  An MTL buffer management class is used both within the MTL daemon and the MTL user library. It provides a data buffer and methods to read or write the buffer, and to move various markers within the buffer.

  Any single data buffer will be referred to and controlled by two buffer managers. One is in the user application and the other is in the context associated with the user.

- User interface

  The MTL user interface class is only used within the MTL user library. It provides a standard set of methods for the application to request service from the MTL daemon. Protocol- and service-specific access methods can be derived from this class.

2.2.3 Extended MTL

The protocol building blocks are created to construct different RMTPs to meet different requirements. It would be very useful to have implementation building blocks, which can be used to construct implementations of various RMTPs. The MTL was designed for this purpose. However, it was only used to implement the Xpress Transport Protocol (XTP) [C+92, S+95, S+98]. In order to validate that the MTL is suitable for the implementation building blocks, we used the MTL to implement another protocol, RMTP-II.

RMTP-II differs from XTP in many aspects. The differences that are related to the MTL are listed in the following:

- Transport layer access point

  In XTP, there is only one access point between the protocol and the transport layer. All users of XTP are using the same access point to communicate with other nodes in the network. However, in RMTP-II, every user may have multiple access points with the transport layer depending on the user's type.

- Transport layer protocol

  XTP can be implemented on top of either raw IP or UDP. The protocol specification only requires the underlying transport layer to provide framing. RMTP-II is built on top of UDP. UDP port numbers are used by the protocol in various places.

- Protocol packet

  Since XTP uses only one access point with the transport layer, destination identifiers are embedded inside protocol packets. RMTP-II relies on the source address, source port, destination address, and destination port of a protocol packet to identify the packet's destination. It is not necessary to parse the protocol packet to locate the destination.

- Node type in the protocol

  All nodes within RMTP-II are in a tree hierarchy. RMTP-II not only defines sender and receiver node types as XTP does, but also defines more node types: top node, aggregator, and designated receiver.

- Data sequence number

  The sequence number used for the data transmitted in XTP is byte based. However, RMTP-II uses the sequence number to identify an individual data packet.

Because of the above differences, in order to implement RMTP-II with the MTL, we extended the MTL to meet the new requirements.

The design of the MTL does not allow multiple classes derived from the *context* class. In order to support completely different node types in RMTP-II, a new class is created. Originally, the *context* class handles packets and user requests. After introducing the new class, the *context* class becomes only a wrapper that passes the packets or requests to the new class.

In order to support multiple access points with transport layer, all interfaces between the MTL and the transport layer were modified to include one more parameter, the access point.

Because RMTP-II relies on the addresses and ports of source and destination instead of the contents to dispatch the received packets, the dispatch mechanism inside the context manager is re-written.

With these modifications, we were able to implement RMTP-II with the extended MTL.

2.2.4 Inadequacies

During the implementation of RMTP-II, we found more inadequacies of the MTL.

- Insufficient support for multiple users in an application

  The MTL does not limit the number of users supported for an application. But if multiple users exist in an application, the MTL can only support a request from one user at a time. For example, if one user sends a request that takes a long time to complete, other users in the application are blocked from sending any request to the MTL daemon.

- No memory management mechanism

  The MTL does not provide any general memory management mechanism. When objects of a single type need to be allocated and freed often, the MTL pre-allocates a pool of the objects (i.e., the *packet* and *context* classes) in order to enhance efficiency and avoid memory fragmentation. Moreover, the MTL does not provide any mechanism to retrieve the statistics of memory usage.

- Inefficient timer management implementation

  In the MTL, a timer belongs to a *context*. In order to find the nearest timer, the context manager will request all contexts to return their timers. This is not an efficient implementation [Jia01].

Although we implemented RMTP-II with the extended MTL, the design of the MTL does not fully support the requirements of some RMTPs. For example, we created a new class to support different node types. But the new class's functions belong to the *context* class. It is redundant to have both the new class and the *context* class in the MTL, which is not suitable for every RMTP (i.e., XTP only needs the *context* class). Due to these problems and inadequacies, the MTL, or even the extended MTL, cannot be readily used as the building blocks for RMTP implementations. With the experience of implementing RMTP-II with the extended MTL, a complete set of implementation building blocks are presented in the following chapters.

Chapter 3

Analysis of RMTP Implementation Building Blocks

3.1 RMTP Implementation Framework

An RMTP is used to transfer data from one or many senders to many receivers using multicast. The RMTP implementation is the provider of this service. Including an entire RMTP implementation within an application would make the application heavy and difficult to write and debug. However, because of the complexity of implementation and difficulty in debugging, the RMTP implementation cannot be within the kernel either. Within this thesis, an approach to implement the RMTP in a separate process is developed. As shown in Figure 2, all applications are linked with a light RMTP application library to communicate with an RMTP process, which contains an RMTP implementation.

In this chapter, we will divide the design space for RMTP implementation building blocks into three parts. Section 3.2 discusses the building blocks in the RMTP application library and those used between the application and the RMTP process. The building blocks used by the RMTP process to communicate with the Operating System (OS) are addressed in section 3.3. Finally, the building blocks used in the RMTP process are analyzed in section 3.4.

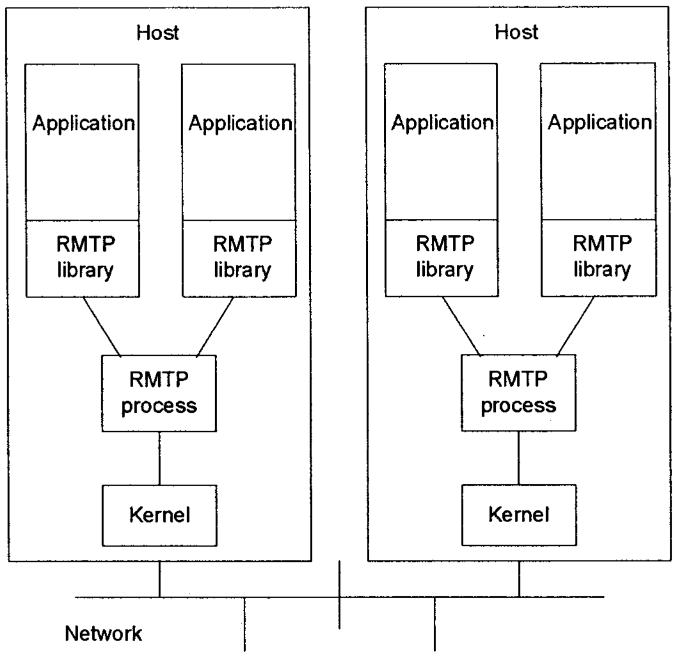


Figure 2: RMTP Implementation Architecture

3.2 Building Blocks in the RMTP Application Library

The implementation of an RMTP provides data transmission service to applications. Before actually sending or receiving data through an RMTP, an application must set up a multicast group session based on the RMTP specification. During the transmission of data some errors may occur. The RMTP process should be able to inform the application about those errors. At any time, an application may request the RMTP process to retrieve any information concerning the session. These requirements lead to the following building blocks.

3.2.1 User Building Block

The user building block defines the characteristics of a user in an application. A user represents an endpoint within a reliable multicast group session, which is a sender, a receiver, or a protocol node.

The user building block has the following requirements:

- It should be able to support different types of users.

- Each individual user should be able to send commands to the RMTP process.
- For a command that needs a long time to handle, the application should not be blocked because of the command. Asynchronous command handling should be supported by the user building block.
- A user should be able to receive notification messages from the RMTP process.
- Every user should be able to send and receive data independently and efficiently to and from the RMTP process.

3.2.2 Command Channel Building Block

This building block describes a command channel between a user in an application and the RMTP process. Each user has its own command channel. A user uses its command channel to send commands to, and receive results or notifications from the RMTP process.

When a command, a result, or a notification, which is referred to as a control message, is sent out by one side of the command channel, the other side should have some way to know of its arrival. It is the command channel's responsibility to inform the RMTP process or its user in this instance.

According to the architecture of the RMTP implementation presented earlier, the RMTP process is a stand-alone process. Users begin in applications that do not have a communication channel with the RMTP process. The command channel is the first channel that can be used to communicate between a user and the RMTP process. In this circumstance, the command channel building block must be able to establish the channel between a user and the RMTP process without any other means of communication.

After the channel is established, commands, results, or notifications through the channel must not be lost. Therefore, reliability is another requirement for command channels.

3.2.3 Data Channel Building Block

Similar to control messages, data blocks represent another kind of information that needs to be transferred between a user and the RMTP process. The command channel between a user and the RMTP process can be used for this purpose. However, because implementations of RMTPs provide data transferring service, there is a large amount of data transferred between a user and the RMTP process. Control messages may be blocked or delayed if they share the same channel with data. Having a separate data channel addresses this problem.

Since the data channel between a user and the RMTP process can be connected after the command channel is established, extra information can be exchanged between the user and the RMTP process to create the data channel. There is no need to require that the data channel can be created without any communication between a user and the RMTP process. This flexibility is not found in the command channel building block. Moreover, due to the command channel, a data channel is not required to be able to inform its user or the RMTP process about the new data put into the channel.

Data transferred through a command channel must not be lost. Reliability is also a requirement for data channels.

3.2.4 User Manager Building Block

An application may have multiple users simultaneously. Each user is able to receive notification from the RMTP process through its own command channel independently. Managing all users and dispatching notifications to the right user is the responsibility of the user manager building block. An application can use this building block to monitor any information from the RMTP process for all users.

An application is dynamically able to add a new user or remove an existing user from the user manager.

3.2.5 Data Buffer Building Block

When a sender wants to send data, the data has to be put into a packet as its payload and then be sent out. If the low-level transportation service is not reliable, for reliability, the data sent out should be stored in a buffer until the acknowledgement of the data reception is received. Once a packet arrives at the receiver, the payload of the packet will be extracted and placed into a buffer until an application requests it. The data buffer building block is designed to provide the functionality of a buffer to store the data.

Either receivers or senders may use data buffers. The requirements for the data buffer building block depend on whether it is used by a receiver or a sender. However, any data buffer has one provider and one consumer. As shown in Figure 3, for the data buffer used by a receiver (referred to as a receiver data buffer), the low-level transportation system is the provider of the data buffer, and a user in an application that wants to receive the data is the consumer. For the data buffer used by a sender (referred to as a sender data buffer), the user is the provider and the low-level transportation system is the consumer. A data buffer stores data after the provider provides it and before the consumer consumes it. For a receiver data buffer, once the consumer of the buffer consumes some data, there is no need to keep them in the buffer. The sender buffer must keep the data, even when the data is already consumed, until the data have been acknowledged. The data buffer building block should satisfy the different requirements for both sender and receiver.

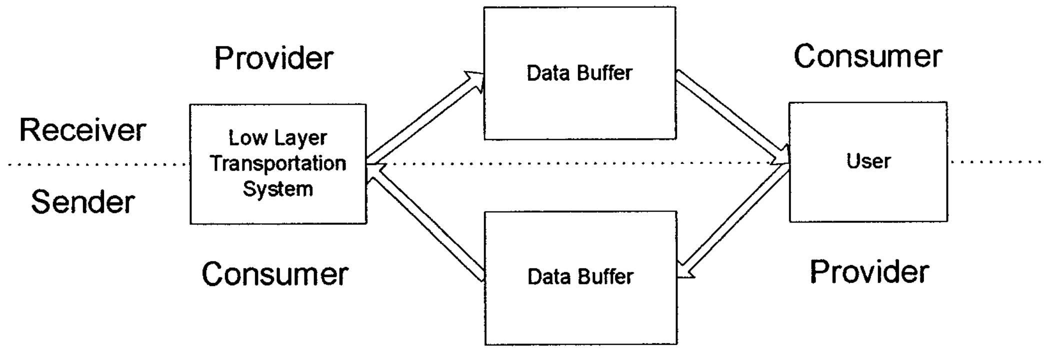


Figure 3: Data Buffer Usage

No matter where the buffer is used, the following should be considered when designing the buffer building block.

- Ordering

  The mechanism used by the data buffer to provide the data is First-In-First-Out (FIFO). The order of data provided to the consumer should be the same as the order when the data are provided by the provider to the data buffer. However, there is an exception when the provider selectively fills the data in the data buffer.

- Re-usability

No matter what size a data buffer is, it is possible to use up all available space in the data buffer. When the consumer consumes data, it should tell the data buffer to discard the data consumed. The space in the data buffer used by the previously consumed data should be reusable for future data.

- Selective filling

  The provider may provide data to the data buffer accompanied by a position where it should be put in the buffer. In this case, the data buffer may receive the data out of order. The data buffer should be able to store the out of order data and provide them to the consumer in the correct order. Since the provider decides the position of the data in the data buffer, there may be holes in the buffer where no data is provided. The data buffer should be able to provide the subset of data before any hole to the consumer.

3.3 Building Blocks for the Interface with OS

The RMTP process provides services to the application atop of the OS. This section addresses the APIs between the RMTP process and the OS.

3.3.1 Address Building Block

In order to transfer data, a user must specify a destination address. When some data arrive, a source address and/or a destination address are used to find the appropriate user, who is waiting for the data. The address building block is designed to encapsulate all functions applicable to an address.

RMTPs are based upon IP. Only raw IP and UDP can support multicast. The address building block needs to support them both.

Copying from one address to another or comparing one address with another should be provided by the address building block. Other auxiliary methods, such as setting or getting all attributes of an address, should also be available.

3.3.2 Delivery Building Block

The delivery building block is designed to send/receive data to/from the OS. It is used along with the address building block to provide transport functionality to an RMTP.

The delivery building block is conceptually located between the RMTP process and the IP stack, and is not visible to the users of RMTPs.

For the RMTP process, the delivery building block is the only access point for communicating with other RMTP endpoints in the protocol. Since only raw IP and UDP can be used for multicast, the delivery building block should include support for these two protocols.

Sending and receiving data are the two main methods provided by the delivery building block. Usually, one pointer to one buffer is passed as a parameter to these two methods. However, sometimes when sending data, an array of pointers to multiple buffers is preferred. The send method should support these two types of parameters.

Another set of methods provided by the delivery building block concern joining or leaving a multicast group session.

3.3.3 Packet Building Block

The delivery building block provides the transportation methods. However, it does not address how to store the data that are received or that are to be sent. The packet building block satisfies this requirement.

When an RMTP packet arrives, the delivery building block is used to receive it. Then, the packet building block is used to store it for later processing. The packet building block can also be used as an intermediate container when sending out data. The data received from a user cannot be sent out directly. For example, an RMTP packet header must be prepended to the data. The packet building block can be used to construct the full datagram for sending.

Although the packet building block is designed as a container for the real data, the buffer to hold the data can be created internally within the packet building block or passed from outside. The packet building block should support these two types of packet.

RMTPs use raw IP or UDP as a delivery service. Any RMTP PDU is encapsulated within a raw IP or UDP PDU. However, an instance of the packet building block only represents a part of the RMTP PDU. The OS constructs the raw IP or UDP packet that will be sent over a network.

The packet building block is also used as the base class for the packets of various RMTPs. However, due to different packet design in different RMTPs, the packet building block should not put any restrictions on the packet format. A packet is only a container of the real datagram received or sent. This suggests that the methods of the packet building block are not to set or to retrieve fields within the packet. These methods handle all data within the packet as a whole.

3.4 Building Blocks in the RMTP Process

In this section, we will discuss the building blocks used internally by the RMTP process. These are the core building blocks of an RMTP implementation.

3.4.1 Task Building Block

In previous sections, the delivery building block and the packet building block are introduced as the components to communicate with the OS. These two building blocks are used by the task building block.

A task is a function entity within the RMTP process. It represents an access point, which is ultimately used by a user in an application. At one time, a user in an application may have multiple communication links with other nodes in a multicast group session. The design of the task building block should be able to support multiple tasks for a user.

When a task is used to send out packets, the RMTP will specify the access point to be used by the user relevant to the circumstance. Even if there are multiple tasks created for a user, based on the protocol specification, the RMTP process is able to select the right task to do the sending. However, when a packet arrives, the task building block should allow the RMTP process to find the corresponding task(s) to receive it. It is worth pointing out that, in a multicast conversation, there may be multiple users waiting for the same packet. Therefore, there may be multiple corresponding tasks for one received packet.

Another problem to be addressed by the task building block is asynchronous processing. When a task is used for communication, the rate of arriving packets may be higher than that of packet reception by the user, or the rate of packet sending by the user may be higher than that of sending through the OS. The task building block needs to handle such a situation.

3.4.2 Task Manager Building Block

The task manager building block organizes all tasks. One main function of the task manager building block is to link all tasks and provide a mechanism for locating the appropriate task when a packet arrives. Another requirement for the task manager building block is to manage all delivery objects referred to by all tasks.

3.4.3 Node Building Block

The RMTP process may serve multiple users at once. In order to perform actions on behalf of different users, there should be an entity within the RMTP process to represent a user. This entity is defined as a node.

The requirements of the node building block are to receive and interpret the commands received from its user, to manage all tasks for commands, and to deliver the received data to a user.

3.4.4 Node Manager Building Block

The main functions of the node manager building block are the creation, maintenance, and deletion of nodes. It also must dispatch commands received from a user to its corresponding node within the RMTP process.

3.4.5 Timer Building Block

A timer is widely used in any protocol implementation, especially for reliable transport protocols. For example, when a sender sends out some data, a timer should be set for receiving an acknowledgement. If no acknowledgement is received before the timer has expired, an action, re-sending the data, should be performed. The action is referred to as the timer handle.

When we set different timers for different packets, which are sent at the same time, the values for the all re-send timers are the same. If multiple packets are lost, the corresponding timers will simultaneously fire and all packets will be resent at the same time. This will cause a network burst. In order to avoid this situation, a small amount of jitter can be added to a timer. The jitter added may be an absolute value, representing a time, or a percentage of the timer value. The jitter is used along with the timer value when setting a timer.

Once a timer is fired and its handle is executed, the timer attributes define whether to delete or keep the timer. Some timers operate only once. After they are fired, they should be deleted. Some timers should exist indefinitely. After they are fired, they should be reset for future timing. Both types of timer should be supported by the timer building block.

Because most UNIX systems are not real-time systems, not all timers can be fired at exactly the defined time. When resetting the timer after it has expired, it may be necessary to make the timer accurate for the next time expiration according to the requirement. The timer building block should provide a way for the user of the timer to specify this attribute.

3.4.6 Timer Manager Building Block

All timers created in the RMTP process will be managed by the timer manager building block. In addition to the methods used to manage all timers, there are two methods provided by the timer manager building block. The first is to find all expired timers and to execute their timer handles. The second is to return a time when the nearest timer will expire. The RMTP process can use this time to decide how long it should wait for any incoming packets or user commands [section 3.4.10].

3.4.7 Trace Building Block

It is difficult to analyze an RMTP or its implementation while in operation because of the high rate of data transfer. The trace building block is designed to output some readable messages to a terminal or a file, providing a way to do offline analysis. The readable messages are defined as trace information. The output of the trace information can be to a terminal, a file, or some other destinations. The trace building block should be flexible enough to support different destinations.

An instance of the trace building block should be able to support different features or modules. Each of them can be enabled or disabled independently and dynamically. A trace message will only be sent to a trace destination when the trace is enabled. A new instance of the trace building block should be able to inherit all configurations from an existing instance.

3.4.8 FSM Building Block

Almost all RMTPs use Finite State Machines (FSMs) as their fundamental abstraction. The FSM building block provides a general framework to create, define, and execute various state machines.

Given that different RMTPs have different state machines, all states, events, and transactions cannot be predefined in the FSM building block. The FSM building block should provide a way to register all states, events, and transactions in order to initialize an instance.

When an event occurs, the FSM building block should be able to automatically carry out the transaction based on the current state and all information registered during initialization.

3.4.9 Memory Manager Building Block

The purpose of a transport protocol implementation is to transport data from one endpoint to another. Before a packet is sent out by a sender, a buffer needs to be allocated to store the data received from the application; after a packet arrives at a receiver, a buffer needs to be allocated to store the packet until the packet is picked up by the application. In addition to the consumption of memory for packet buffers, there are many other consumers of memory, such as IP addresses, nodes, and other internal structures. The OS does provide the functions to allocate and free memory. However, it has three disadvantages:

1. Memory fragmentation

   Frequent memory allocation and deallocation are the common behaviors for any protocol implementation. The memory allocation and deallocation functions provided by the OS will cause memory gaps. After running the protocol a long time, a slightly larger memory allocation may fail even though the system has mainly small size memory fragments.

2. Efficiency

   Memory allocation and deallocation are provided by the OS as system calls, which require at least two context switches. The more frequently memory is allocated or freed, the slower the implementation.

3. Difficult to get memory usage

   As a protocol implementation, it is very important to know how memory is used. Knowing the usage of memory for some kinds of the components will help in analyzing the protocol implementation. With memory allocation and deallocation functions provided by the OS, it is possible to get the amount of the memory consumed by the protocol. However, a further breakdown of memory usage is unclear.

The memory manager building block is designed to solve these problems.

3.4.10 RMTP Process Building Block

As illustrated in Figure 2, an RMTP implementation is a stand alone process. It waits for the commands from users in applications and communicates with other endpoints in the protocol through the OS. The RMTP process building block provides the methods to set up and control the process.

An RMTP implementation can be executed in the foreground as a process or in the background as a daemon. When an RMTP implementation is a daemon, it must detach from its terminal. This ability is provided by the RMTP process building block.

When the RMTP process is being executed, its status should be retrievable. The RMTP process building block provides a way to dump its internal status. The trace building block operates differently from this dump function. The trace building block is used to provide information over a period of time while the dump function is used to provide a snapshot of all modules within the RMTP process at a certain time.

The RMTP process building block also provides a flow chart of the process. It cooperates with all other building blocks to construct a framework for an RMTP implementation.

Chapter 4

The Design of RMTP Implementation Building Blocks

Based on the preceding analysis of the requirements for the RMTP implementation building blocks, this chapter presents the design of all building blocks.

The mechanisms to process commands and notifications are addressed first in section 4.1 and 4.2. Afterwards, every building block previously specified is studied.

4.1 User-Daemon Command Processing

In this section, two mechanisms are defined allowing a user to send a command to and receive results from the RMTP process.

4.1.1 Blocking Command Processing Mechanism

In a blocking command processing mechanism, a user must wait for the reply from the RMTP process. Furthermore, the RMTP process cannot send the reply until it finishes executing the command. During the processing time, the application is not free to do other jobs.

As an example, consider an application that already includes several users who are receiving data. When a new user is created in the application and seeks to enter a multicast conversation, the RMTP process receives a join command from the user. The RMTP process then asks the other endpoints already within the multicast group session for approval of the membership. Processing this command may take a long time. During the time spent waiting for membership approval, the application is blocked and is not able to receive data from other users.

With this mechanism, after the user sends the command, the application will be blocked until it receives the result sent back from the RMTP process [Figure 4]. The details of sending commands and receiving results are transparent to the application. This command processing mechanism simplifies the implementation of an application.

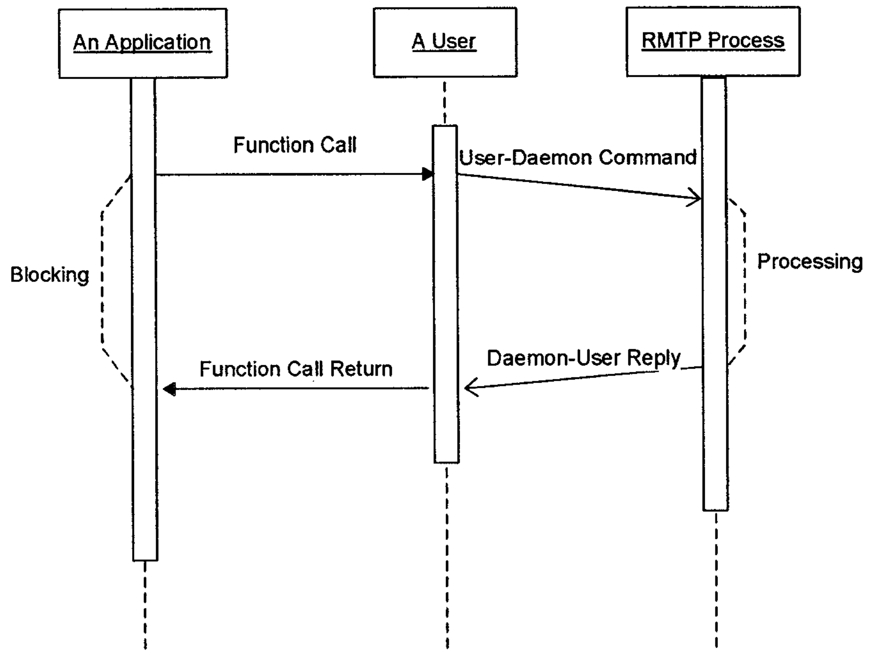


Figure 4: Blocking Command Processing Mechanism

- User-Daemon command format

  All user commands to the RMTP process are defined based on the protocol specification. A general command format is depicted in Figure 5.

```
                     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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Protocol   |    Version    |  Command ID   |     Length    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Parameters ...                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5: User-Daemon Command Format

Every RMTP is allocated a unique protocol ID, which is found in the first byte of the command packet. The version is the implementation version number so that the RMTP process is capable of handling different versions of the implementation even for the same protocol. The command ID is one byte with a range of 0–255. The length field defines the length of the parameters. Its value can be zero in the case where no parameters are needed. Since the size of the length field is one byte, the maximum length available for parameters is 255 bytes. The format of the parameters field is command specific.

- Daemon-User command reply format

  When the RMTP process receives a command from a user, the process decides what processing should be performed according to the protocol specification. A reply is sent back to the user with the result when processing is completed. The format of the reply is defined in Figure 6. The return code is one byte long. Its interpretation depends on the command. If additional information is needed in the reply, a variable length (maximum of 255 bytes) message can be appended to the end of the reply packet. The length of the message is stored in the second byte of the reply packet.

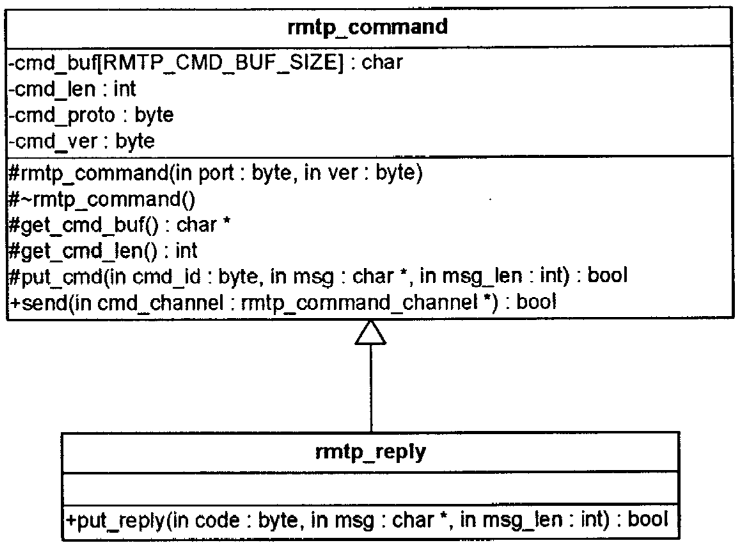
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Return Code  |    Length     |          Message ...          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6: Daemon-User Command Reply Format

There is a building block that describes both the command and the reply. The protocol and version fields are not handled by the command building block. These two fields will be prepended to the command message when the user sends the command. Without the protocol and version fields, the command message has the same form as the reply message. As shown in Figure 7, the *rmtp_reply* class is a subclass of the *rmtp_command* class.

4.1.2 Non-Blocking Command Processing Mechanism

From the example presented in the preceding section, a blocking command processing mechanism is very inefficient for commands that require a long processing time. A non-blocking command processing mechanism is required.



rmtp_command
-cmd_buf[RMTP_CMD_BUF_SIZE] : char
-cmd_len : int
-cmd_proto : byte
-cmd_ver : byte
#rmtp_command(in port : byte, in ver : byte)
#~rmtp_command()
#get_cmd_buf() : char *
#get_cmd_len() : int
#put_cmd(in cmd_id : byte, in msg : char *, in msg_len : int) : bool
+send(in cmd_channel : rmtp_command_channel *) : bool
rmtp_reply
+put_reply(in code : byte, in msg : char *, in msg_len : int) : bool

Figure 7: Command Building Block

In a non-blocking command processing mechanism, no matter how much time the RMTP process needs to handle a command, a reply is sent back to the user upon reception of the command. The format of this reply is as same as other replies. In this reply, a special "would block" return code is used to inform the user that the command would block the user. When the user receives the reply, the return code of "would block" will be returned to the application. The application is able to do other jobs during the time that passes while waiting for the command to be finished. Although the command channel is reliable, an immediately reply is still required in this mechanism so that a user can combine the blocking command processing mechanism with the non-blocking command processing mechanism.

Later, when finished processing the command, the RMTP process has to inform the user of the result. A special reply message is used for this purpose. It uses the same reply format described above along with a special return code. The actual result is found in the message part of the reply packet and also in the reply format [Figure 8]. When the user receives the result-in-reply message from the RMTP process, it will retrieve the real return code for the command from the message part and then inform the application of the result [Figure 9].

In the non-block command processing mechanism, the application is not blocked by any commands of a user during the time spent processing the command. Therefore, the application can handle multiple users more efficiently.

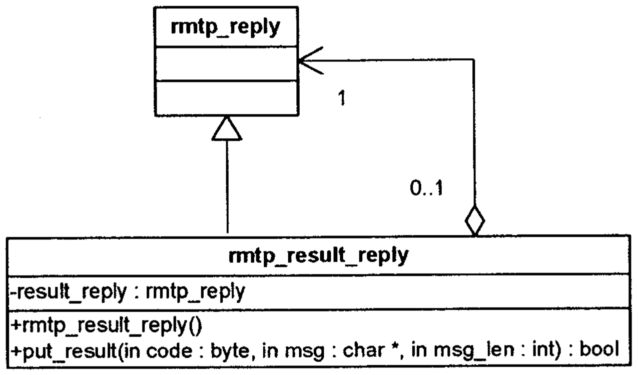

rmtp_reply
1
0..1
rmtp_result_reply
-result_reply : rmtp_reply
+rmtp_result_reply()
+put_result(in code : byte, in msg : char *, in msg_len : int) : bool

Figure 8: Result-in-Reply Class

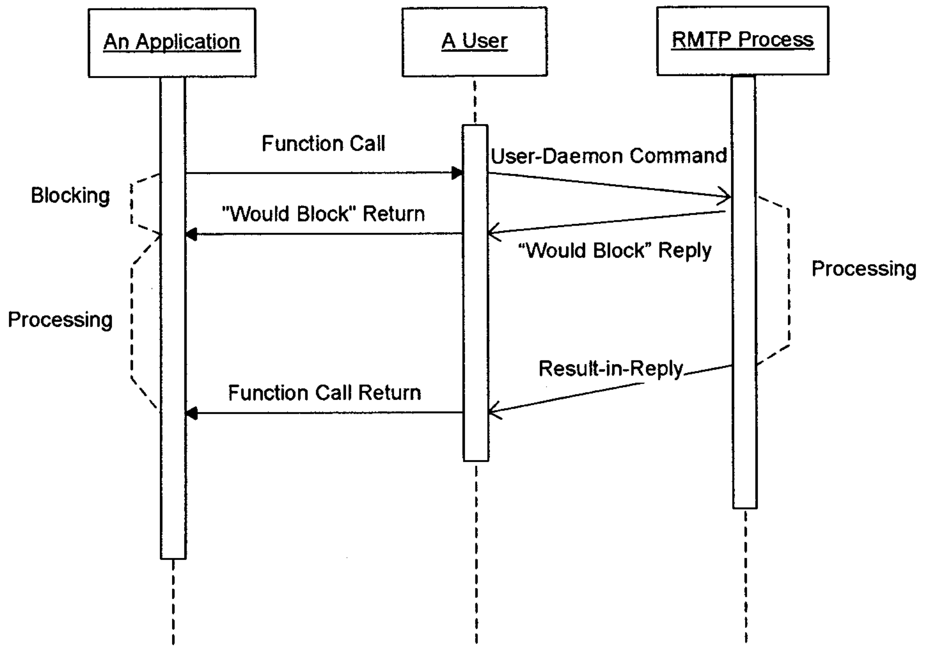


Figure 9: Non-blocking Command Process Mechanism

However, there remains a limitation for this mechanism. When a user issues a command to the RMTP process and receives the "would block" return code, the user cannot issue any other commands until it later receives the actual return code from the RMTP process.

4.1.3 Send/Receive Command Processing

Send and receive are two special commands sent by a user to the RMTP process. Either the command channel or the data channel is used. The details for using the data channel in order to reduce memory copying for the send and receive commands are discussed in the data channel building block design [section 4.4]. This section focuses on the command processing mechanisms that are used for these two commands.

- Send command

  When a user needs to send data, it first sets the data into the data channel. Since shared memory is used for the data channel, the RMTP process will not be notified about the new data. An explicit send command has to be issued by the user to notify the RMTP process.

  The send command reply mechanism is determined by whether or not the user needs to know that everyone received the data.

  If the user does not need to know the result of the delivery, once a user issues a send command, the user will receive an immediate reply from the RMTP process no matter whether the data are really sent out by the RMTP process or not. The RMTP process must guarantee that the data will eventually be sent out and acknowledged by the receivers. The user will not be blocked by the send command.

  If the user does need to know the result of the delivery, the mechanisms used to process the send command depends on whether the command is blocking or non-blocking. In the case of a blocking send command, the sending user will be blocked until the RMTP process receives the result of the delivery from the receivers and sends the result to the user. In the case of a non-blocking send command, the user who issues the send command will receive a reply from the RMTP process to identify that this command would be blocked so that its application can continue with other activities. Later on, when the RMTP process knows the result of the delivery, a second reply will be sent to the user to inform it of the actual result. These two mechanisms, used by the send command, are consistent with the general blocking and non-blocking command processing mechanisms.

- Receive command

  When a user needs to receive some data, it first sets aside enough space in the data channel. A receive command is then issued to the RMTP process.

  If there are data ready when the RMTP process receives the command from the user, the daemon puts the data into the data channel and sends a reply back to the user.

  If there are no data ready when the command arrives at the RMTP process, in the case of the blocking command processing mechanism, no reply will be sent back to the user until some data later arrive. In the case of the non-blocking command processing mechanism, a reply will be sent back to the user immediately to inform the user that the command would be blocked. A second reply will be sent to the user later when some data become ready.

Whether the blocking or non-blocking command processing mechanism is used for a send/receive command is protocol specific. Moreover, in some protocols, both mechanisms are feasible. The user building block [section 4.5] must support both mechanisms.

4.2 Daemon-User Notification Processing

The user-daemon command processing discussed in section 4.1 is driven by a user. However, in some cases, it is necessary for the RMTP process to drive the communication. For example, in case of an error, the RMTP process has to inform the user about its occurrence. The RMTP process driven communication is also useful when a user wants to monitor the behavior of the protocol. Instead of sending a command to retrieve the information all the time, a user can wait for the RMTP process to push the information automatically when it is ready.

The RMTP process pushes information to a user also through the command channel between the user and the RMTP process. In order to simplify decoding on the user side, the same format as the reply from the RMTP process to the user is reused for the information pushed from the RMTP process. The return code in the reply message is a specific predefined value so that a user can distinguish it from a reply to a command.

The functions addressed in this section are not provided by the user building block. However, since notification processing is between the RMTP process and a user, the user building block should be able to handle it.

4.3 Command Channel Building Block Design

A command channel is used between the RMTP process and a user in a separate application process. There are many mechanisms in UNIX can be used to achieve Inter-Process Communication (IPC) [Ste92, Ste98a]. Some examples are named pipe and TCP socket. Considering the requirements specified in the analysis of the command channel building block, a TCP socket based on the loopback address is used to implement the command channel.

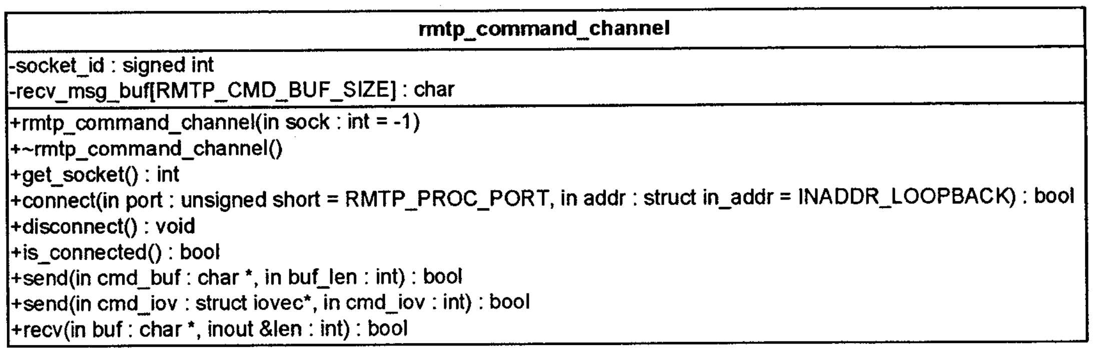
When the RMTP process starts, it will listen to a predefined TCP port for indication of any new command channel. The details for how the RMTP process handles the new command channel are found in the node manager building block design [section 4.14]. When a user wants to create a command channel with the RMTP process, it simply needs to connect to the predefined TCP port. Then, a TCP session, which is used as the command channel, is established. Based on a TCP session, a command channel can provide reliable and duplex communication between its user and the RMTP process. Moreover, the message notification mechanism, which is required by the command channel, can also be provided through the TCP socket [Ste98b].

A command channel belongs to a user. It is created and connected between its user and the RMTP process when the user is created. It is disconnected and deleted when its user quits.

The command channel building block provides transport service to the user building block. It is not public to the application of RMTP. All commands should be encapsulated by the user and sent through the command channel. All results or notifications are received through the command channel and may then be parsed and handled by the user.

The command channel building block is also used in the RMTP process by the node building block [section 4.13].

Figure 10 shows the command channel building block class diagram.



```
rmtp_command_channel
-socket_id : signed int
-recv_msg_buf[RMTP_CMD_BUF_SIZE] : char
+rmtp_command_channel(in sock : int = -1)
+~rmtp_command_channel()
+get_socket() : int
+connect(in port : unsigned short = RMTP_PROC_PORT, in addr : struct in_addr = INADDR_LOOPBACK) : bool
+disconnect() : void
+is_connected() : bool
+send(in cmd_buf : char *, in buf_len : int) : bool
+send(in cmd_iov : struct iovec*, in cmd_iov : int) : bool
+recv(in buf : char *, inout &len : int) : bool
```

Figure 10: Command Channel Building Block

4.4 Data Channel Building Block Design

When there is a large amount of data to be exchanged, the command channel is not suitable to achieve high performance in transferring the data between a user and the RMTP process. This is due to the use of a TCP socket, wherein at least two memory copies are necessary. One memory copy is between the sender and the socket, while the other memory copy is between the socket and the receiver. By using shared memory, one memory copy can be eliminated. When a user, or the RMTP process, needs to transfer data, the sender copies the data into the shared memory. Then, a command is sent by the sender via the command channel to inform the receiver of the presence of data in the shared memory. After receiving the command, the receiver can directly use the data in the shared memory instead of copying the data to its own buffer. In this thesis, System V shared memory is used to implement the data channel.

The data channel building block provides the methods to create, attach, and release a shared memory. Either the user or the RMTP process may create the shared memory and the other end of the data channel attaches to the shared memory. If the user creates the shared memory, the shared memory ID is sent to the RMTP process when the user executes registration (details are in the user building block design). In this case, the RMTP process will attach to the data channel when it processes the register command. If the RMTP process creates the shared memory, the shared memory ID is sent back to the user in the reply to registration. And then the user attaches to the data channel. In the design proposed in this thesis, a user will create its data channels and the RMTP process will attach to them. When a user quits, both the user and the RMTP process will detach from the data channels. However, only the last to detach from the shared memory will delete it.

Because both the user and the RMTP process use the shared memory in a data channel, a mutex locking mechanism should be implemented to protect the shared memory. A System V semaphore is used here for this purpose. The semaphore is created by the same entity that created the shared memory (in this thesis, the user). Then the new semaphore key is sent to the other side of the data channel along with the shared memory ID. The semaphore will be deleted in the same manner as the shared memory.

The data channel building block [Figure 11] only provides methods for maintaining the data channel. The manner in which the data are stored within the data channel is not in the scope of this building block. The details for managing the data are in the data buffer building block design [section 4.7].

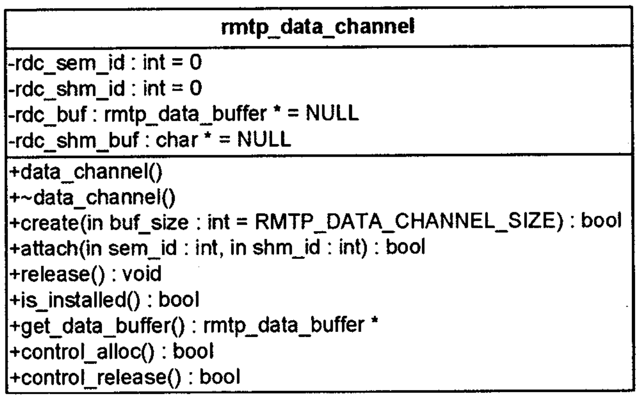

rmtp_data_channel
-rdc_sem_id : int = 0
-rdc_shm_id : int = 0
-rdc_buf : rmtp_data_buffer * = NULL
-rdc_shm_buf : char * = NULL
+data_channel()
+~data_channel()
+create(in buf_size : int = RMTP_DATA_CHANNEL_SIZE) : bool
+attach(in sem_id : int, in shm_id : int) : bool
+release() : void
+is_installed() : bool
+get_data_buffer() : rmtp_data_buffer *
+control_alloc() : bool
+control_release() : bool

Figure 11: Data Channel Building Block

4.5 User Building Block Design

A user is an entity within a multicast group session. When a user is created, it registers itself with the RMTP process. When a user leaves a multicast group session, it de-registers itself from the RMTP process.

A user can be a sender, a receiver, or a any kind of node defined in an RMTP. The type of the node is the role of the user. The different types of roles are defined by the RMTP protocol. In order to simplify the implementation, a user can only belong to one role. The role is decided when the user is created and cannot be changed later. A user sends its role to the RMTP process when it registers itself with the RMTP process.

In order to communicate with the RMTP process, a command channel will be created between the user and the RMTP process. Moreover, both receive and send data channels are created when the command channel is created. The register command delivers, to the RMTP process, the role of the user as well as information relating to the send and receive data channels [Figure 12]. After the command channel is established, the user sends the register command to the RMTP process, allowing the RMTP process to complete its part in the initialization of the new user (details in the node manager building block design). When a user leaves the multicast group session, a de-register command [Figure 13] is sent to the RMTP process. Both the register command and de-register command use the blocking command processing mechanism. It is worth pointing out that the protocol fields in both command messages are filled with 0xFF. The protocol number, 0xFF, is reserved for commands that are protocol independent.

After the user has successfully registered with the RMTP process, the user can implement protocol functions by sending commands to the RMTP process. The format of commands depends on the protocol. Use of the blocking or non-blocking mechanism for each command is implementation dependent. When the RMTP process needs to send notifications to the user, the daemon-user notification processing mechanism is used. How to dispatch the notifications received from the RMTP process to the right user is addressed in the user manager building block. However, each user must provide a method to handle such notifications. This method is also the handler of the reply messages, which carry the results of asynchronous commands. When a result-in-reply message is received, the actual result code and message will

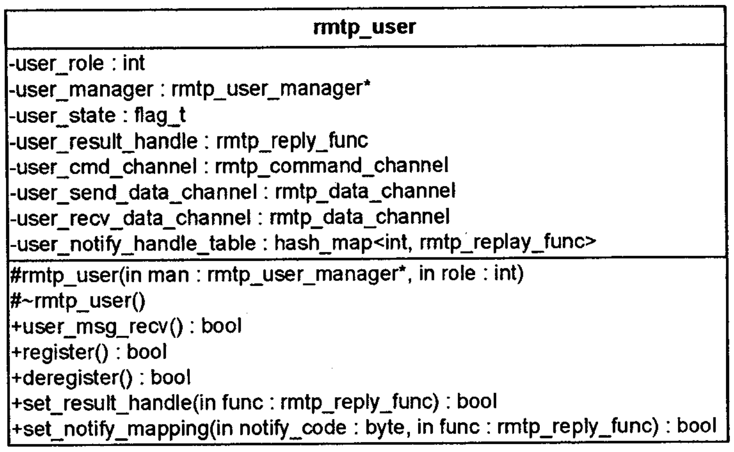
```
                    1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     0xFF      |     0x01      | RMTP_CMD_REG  |     0x14      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|User Role Type |                   Reserved                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Receive Buffer Shared Memory ID                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Receive Buffer Semaphore ID                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Send Buffer Shared Memory ID                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Send Buffer Semaphore ID                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
RMTP_CMD_REG = 0x01
```

Figure 12: Register Command Format

```
                    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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     0xFF      |     0x01      |RMTP_CMD_DEREG |     0x00      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
RMTP_CMD_DEREG = 0x02
```

Figure 13: Deregister Command Format

be extracted from the result-in-reply message and passed to another method, which then handles the result of an asynchronous command. The user can dynamically register the result handle method after a blocking command is sent out. However, at any time, a user can only have one result handle method registered. It is the responsibility of the user building block and node building block to detect the error of multiple blocking commands being sent simultaneously.



```
rmtp_user
-user_role : int
-user_manager : rmtp_user_manager*
-user_state : flag_t
-user_result_handle : rmtp_reply_func
-user_cmd_channel : rmtp_command_channel
-user_send_data_channel : rmtp_data_channel
-user_recv_data_channel : rmtp_data_channel
-user_notify_handle_table : hash_map<int, rmtp_replay_func>
#rmtp_user(in man : rmtp_user_manager*, in role : int)
#~rmtp_user()
+user_msg_recv() : bool
+register() : bool
+deregister() : bool
+set_result_handle(in func : rmtp_reply_func) : bool
+set_notify_mapping(in notify_code : byte, in func : rmtp_reply_func) : bool
```

Figure 14: User Building Block

The user building block provides a base class [Figure 14] for the protocol specific user classes. In a specific RMTP implementation, a concrete class is created for each node type defined in the protocol. The methods provided by the concrete class are used by the application to execute protocol functions.

4.6 User Manager Building Block Design

In a non-blocking command processing mechanism, because a reply is sent by the RMTP process immediately, a user will not be blocked. The user has to wait for the second reply for the actual result of the command. Constantly polling the command channel is only acceptable if the application has a single user. In order to support more than one user, a new module, which listens to the RMTP process for all users, is required. This module is the user manager building block [Figure 15].

There is only one user manager within an application. All users of the application must be registered with the user manager. This registration is achieved by passing a user manager object into the construct method of the *rmtp_user* class, that is, when

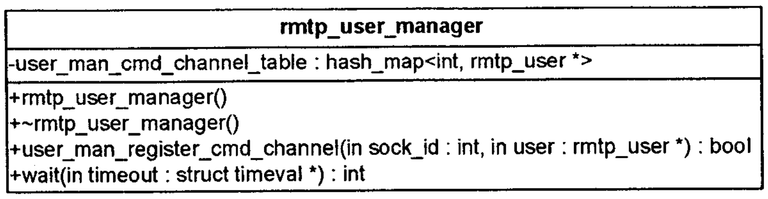

rmtp_user_manager
-user_man_cmd_channel_table : hash_map<int, rmtp_user *>
+rmtp_user_manager()
+~rmtp_user_manager()
+user_man_register_cmd_channel(in sock_id : int, in user : rmtp_user *) : bool
+wait(in timeout : struct timeval *) : int

Figure 15: User Manager Building Block

a new user is created. Within the constructor of the *rmtp_user* class, a command channel will be created. Then, the newly created command channel will be passed to the user manager so that the user manager will have all command channels from all users.

A method to wait for replies from the RMTP process for all users is provided, by the user manager building block, to the application when needed. This method can also be used to wait for information pushed by the RMTP process. The method monitors all command channels for all users. When any message sent by the RMTP process arrives through one command channel, the method, *user_msg_recv*, of the user that owns the command channel will be called. After handling all messages in all command channels, the method returns to the application with the number of messages handled. Using this method, no user will cause the application to be blocked. However, in some cases, the application does not want to wait for messages from its users for ever, even if no message arrives. The waiting time for the messages can be specified when the method is called by the application. The method waits for messages within this time and then returns to the application even if no message arrives.

4.7 Data Buffer Building Block Design

According to the analysis of the data buffer, every data buffer has a provider and a consumer. Every time the provider submits some data to the buffer, the new data should be appended after the data already existing in the buffer. There should be a pointer, *tail*, pointing to the first un-used position, from where the new received data provided by the provider will be written. Following each writing this pointer will be moved to the next available position. Another pointer, *head*, is also required to point to the position where the consumer reads data. This pointer will be moved to the next un-read data in the buffer after each reading. Both *head* and *tail* pointers are required by both the sender and receiver buffers.

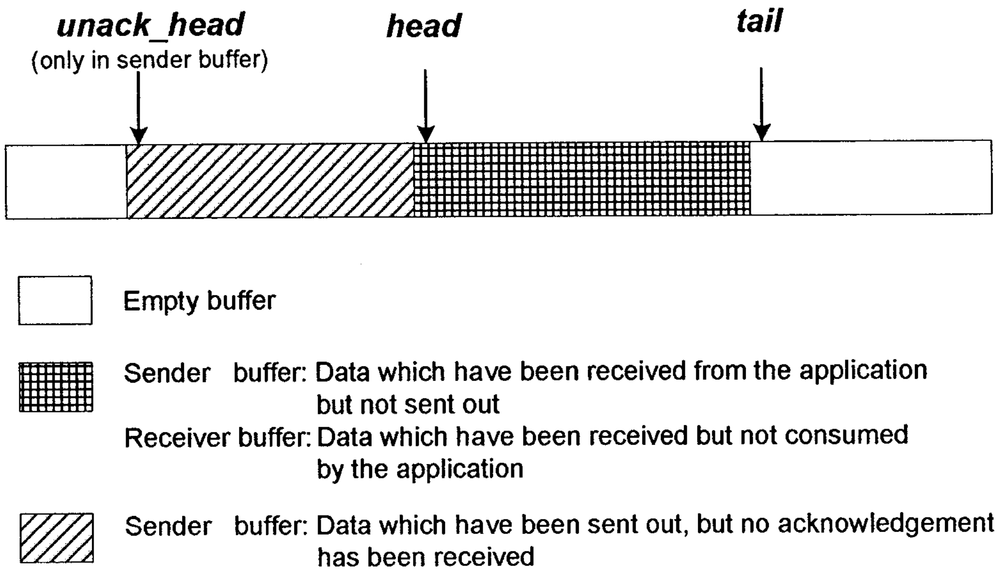

unack_head
(only in sender buffer)
head
tail
Empty buffer
Sender buffer: Data which have been received from the application but not sent out
Receiver buffer: Data which have been received but not consumed by the application
Sender buffer: Data which have been sent out, but no acknowledgement has been received

Figure 16: Data Buffer Layout

Any reliable protocol requires that the sender keeps the data until it receives acknowledgement of data reception. However, it is impossible for a sender to receive the acknowledgement immediately after sending the data. In a buffer used by the sender, *head* always points to the data to be sent next. If the timer for receiving the acknowledgement expires, *head* should be reset to the first un-acknowledged datum, which is pointed to by *unack_head*, so all un-acknowledged data will be re-sent in the future. In the sender buffer, the data from *unack_head* to *head* are those that have been sent out but no acknowledgement has been received. The data from *head* to *tail* are those that have not been sent out. The portion from *tail* to *unack_head* is the un-used space [Figure 16].

An acknowledgement of reception of higher sequence number data implies that all lower sequence data have been safely received. The receiver will use *tail - 1* as the acknowledgement sequence number. Usually, an acknowledgement will not be sent immediately after the receiver receives the data. Rather, it will be delayed until the acknowledgement timer expires, or the receiver needs to send data packets to the sender, allowing the receiver to aggregate acknowledgements. The data stored in the receiver buffer will be consumed at a time depending on the specific receiver application. Receiving data, acknowledging data, and consuming data are different processes. The receiving data process will use, then move, the *tail* pointer; the acknowledging data process will use *tail* pointer; and, the consuming data process will use, then move, the *head* pointer.

A round-robin mechanism should be applied to these points, *head*, *tail*, and *unack_head*, so that the buffer can be used cyclically.

If the protocol uses the byte number as the sequence number, as occurs in TCP or XTP, the buffer manager building block should be able to convert the pointers in the buffer to and from the sequence number. When there occurs a gap between the highest sequence number previously received and lowest sequence number recently arrived, the gap contains missing data. Then, whether to keep or discard the newly received data is protocol specific. The buffer manager building block provides the selective filling method to meet this requirement in the case where the newly received data should be stored.

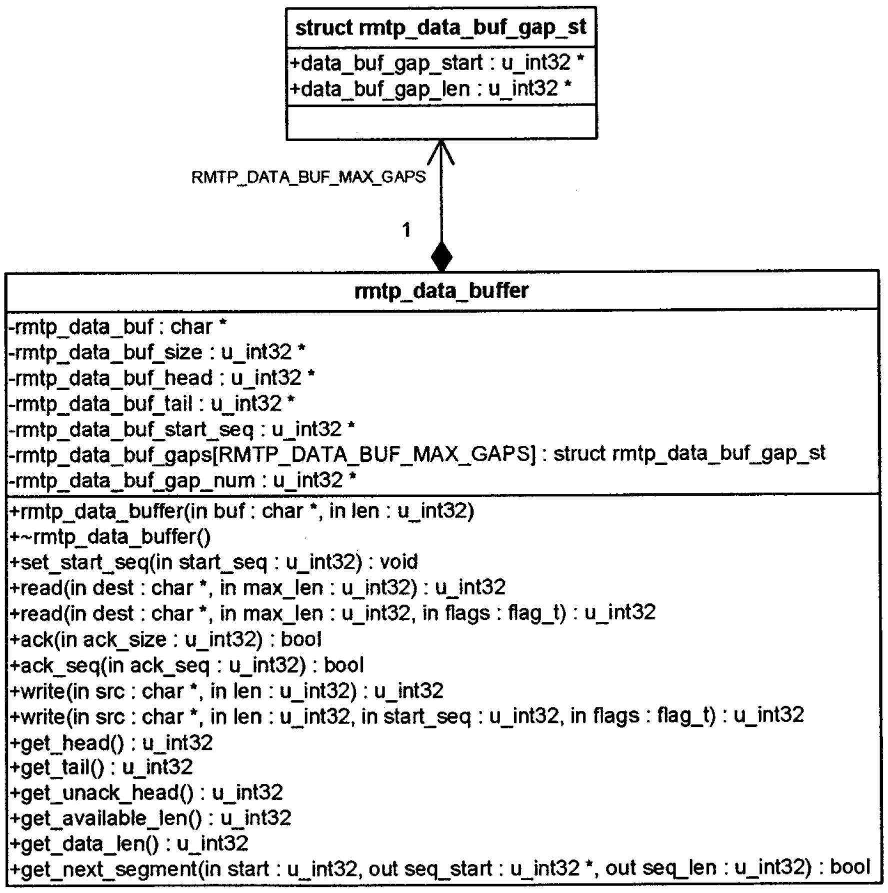

struct rmtp_data_buf_gap_st
+data_buf_gap_start : u_int32 *
+data_buf_gap_len : u_int32 *
RMTP_DATA_BUF_MAX_GAPS
1
rmtp_data_buffer
-rmtp_data_buf : char *
-rmtp_data_buf_size : u_int32 *
-rmtp_data_buf_head : u_int32 *
-rmtp_data_buf_tail : u_int32 *
-rmtp_data_buf_start_seq : u_int32 *
-rmtp_data_buf_gaps[RMTP_DATA_BUF_MAX_GAPS] : struct rmtp_data_buf_gap_st
-rmtp_data_buf_gap_num : u_int32 *
+rmtp_data_buffer(in buf : char *, in len : u_int32)
+~rmtp_data_buffer()
+set_start_seq(in start_seq : u_int32) : void
+read(in dest : char *, in max_len : u_int32) : u_int32
+read(in dest : char *, in max_len : u_int32, in flags : flag_t) : u_int32
+ack(in ack_size : u_int32) : bool
+ack_seq(in ack_seq : u_int32) : bool
+write(in src : char *, in len : u_int32) : u_int32
+write(in src : char *, in len : u_int32, in start_seq : u_int32, in flags : flag_t) : u_int32
+get_head() : u_int32
+get_tail() : u_int32
+get_unack_head() : u_int32
+get_available_len() : u_int32
+get_data_len() : u_int32
+get_next_segment(in start : u_int32, out seq_start : u_int32 *, out seq_len : u_int32) : bool

Figure 17: Data Buffer Building Block

When selective filling is used, a list of gaps is maintained. Each read performed by the consumer may extract all data in the buffer (from *head* to *tail*), including all gaps, or just the first contiguous data segment. A method, *get_next_segment()*, can be used to retrieve the next contiguous data segment based on a given starting sequence number.

```
                     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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          buffer size                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           start_seq                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           unack_head                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              head                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              tail                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            gap_num                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           gap_start                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            gap_len                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             . . .                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             . . .                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           gap_start                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            gap_len                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
|                                                               |
|                             DATA                              |
|                                                               |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 18: Data Buffer Structure

The data buffer is used as an intermediate data container between an application and the RMTP process. It is used along with the data channel building block to provide fast transmission. As shown in Figure 17, no real buffer is allocated inside the data buffer building block. The pointer to a real buffer is received from outside the block when a new data buffer object (*rmtp_data_buffer::rmtp_data_buffer()*) is initialized. The first section of the buffer will be used to store various fields used by the buffer, allowing those fields to be shared between its application and the RMTP process [Figure 18].

4.8 Address Building Block Design

As described in the analysis of this building block, only raw IP or UDP can be used as the low-level delivery protocol for RMTPs. A raw IP address includes an address and a protocol number. A UDP address includes an address and a port number. As shown in Figure 19, a base class is designed to set and get the address. The protocol number in a raw IP address or the port number in a UDP address is handled in different sub-classes.

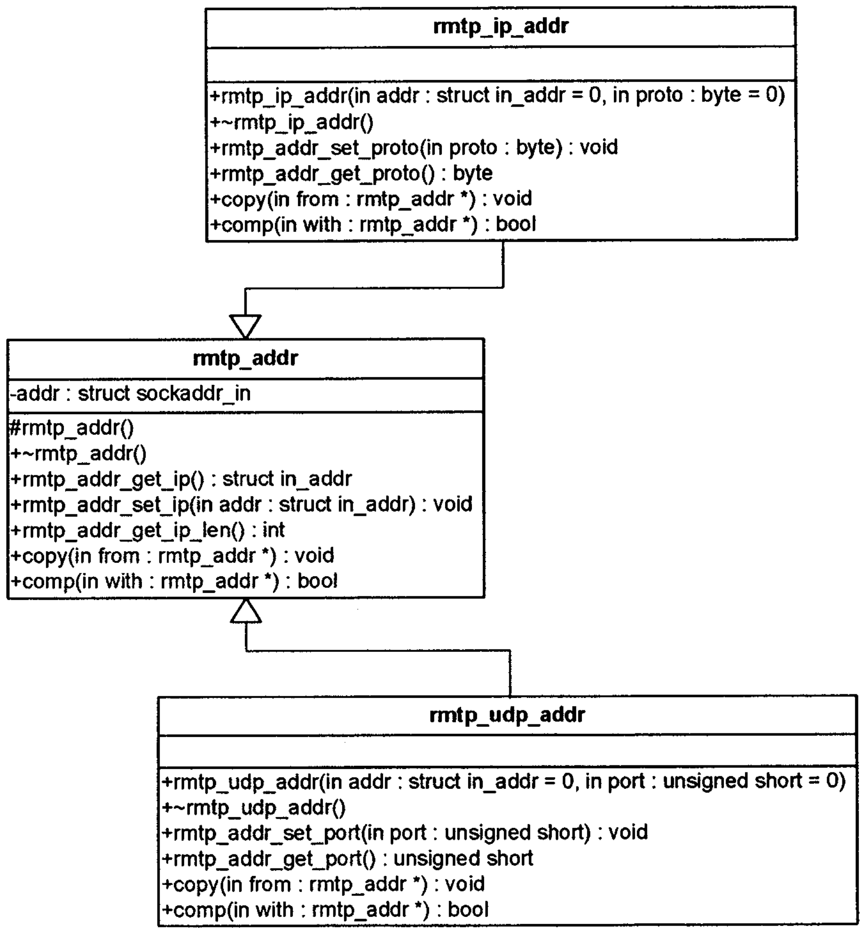

rmtp_ip_addr
+rmtp_ip_addr(in addr : struct in_addr = 0, in proto : byte = 0)
+~rmtp_ip_addr()
+rmtp_addr_set_proto(in proto : byte) : void
+rmtp_addr_get_proto() : byte
+copy(in from : rmtp_addr *) : void
+comp(in with : rmtp_addr *) : bool
rmtp_addr
-addr : struct sockaddr_in
#rmtp_addr()
+~rmtp_addr()
+rmtp_addr_get_ip() : struct in_addr
+rmtp_addr_set_ip(in addr : struct in_addr) : void
+rmtp_addr_get_ip_len() : int
+copy(in from : rmtp_addr *) : void
+comp(in with : rmtp_addr *) : bool
rmtp_udp_addr
+rmtp_udp_addr(in addr : struct in_addr = 0, in port : unsigned short = 0)
+~rmtp_udp_addr()
+rmtp_addr_set_port(in port : unsigned short) : void
+rmtp_addr_get_port() : unsigned short
+copy(in from : rmtp_addr *) : void
+comp(in with : rmtp_addr *) : bool

Figure 19: Address Building Block

4.9 Delivery Building Block Design

Since either raw IP or UDP can be used as the delivery protocol, the delivery building block will be designed similarly to the address building block. One base class is created to support all common attributes and methods. Two sub-classes are designed specifically for raw IP and UDP.

When initializing a new raw IP delivery object, the protocol number must be specified. For a new UDP delivery object, a port number may be specified. If the port number is specified with a value other than zero, the socket for this UDP delivery object will be bound to the specified port number. This allows an RMTP to listen to or send packets out from a specified UDP port.

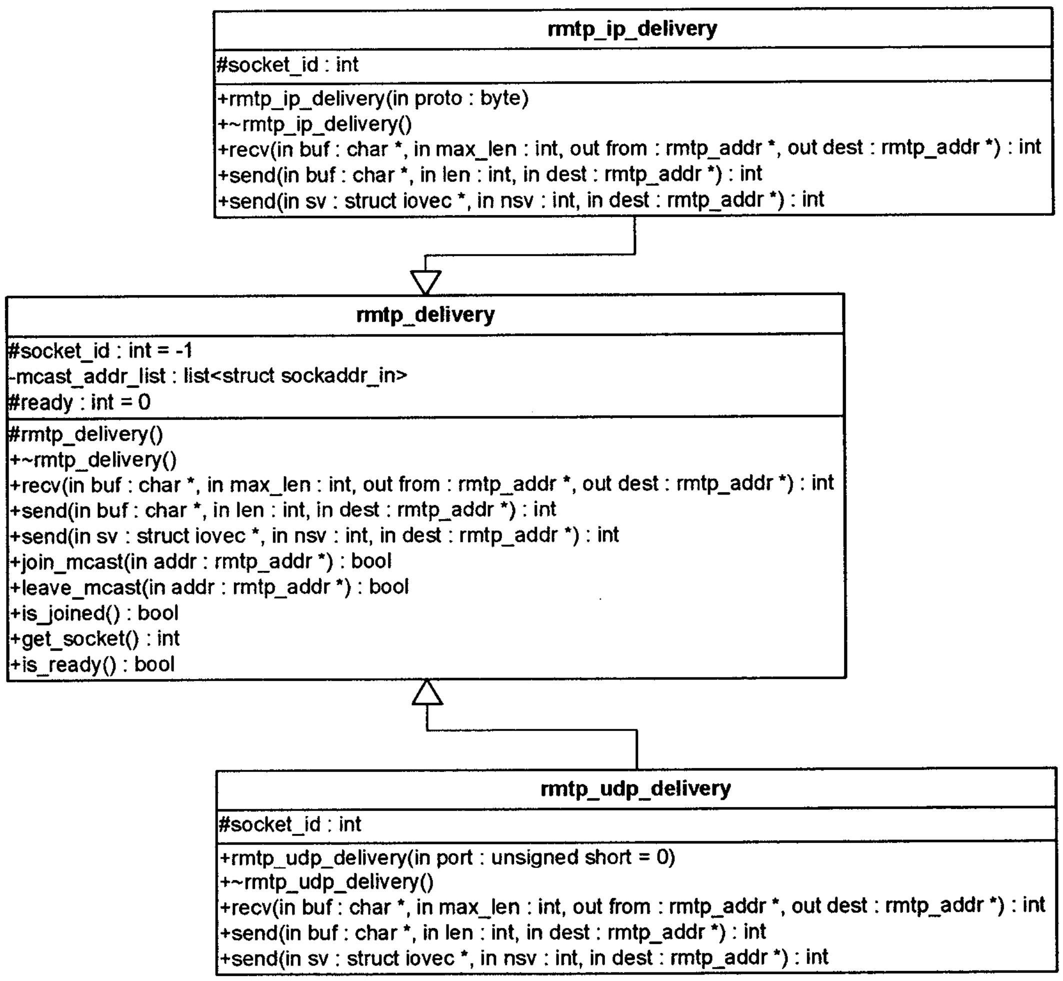

rmtp_ip_delivery
#socket_id : int
+rmtp_ip_delivery(in proto : byte)
+~rmtp_ip_delivery()
+recv(in buf : char *, in max_len : int, out from : rmtp_addr *, out dest : rmtp_addr *) : int
+send(in buf : char *, in len : int, in dest : rmtp_addr *) : int
+send(in sv : struct iovec *, in nsv : int, in dest : rmtp_addr *) : int
rmtp_delivery
#socket_id : int = -1
-mcast_addr_list : list<struct sockaddr_in>
#ready : int = 0
#rmtp_delivery()
+~rmtp_delivery()
+recv(in buf : char *, in max_len : int, out from : rmtp_addr *, out dest : rmtp_addr *) : int
+send(in buf : char *, in len : int, in dest : rmtp_addr *) : int
+send(in sv : struct iovec *, in nsv : int, in dest : rmtp_addr *) : int
+join_mcast(in addr : rmtp_addr *) : bool
+leave_mcast(in addr : rmtp_addr *) : bool
+is_joined() : bool
+get_socket() : int
+is_ready() : bool
rmtp_udp_delivery
#socket_id : int
+rmtp_udp_delivery(in port : unsigned short = 0)
+~rmtp_udp_delivery()
+recv(in buf : char *, in max_len : int, out from : rmtp_addr *, out dest : rmtp_addr *) : int
+send(in buf : char *, in len : int, in dest : rmtp_addr *) : int
+send(in sv : struct iovec *, in nsv : int, in dest : rmtp_addr *) : int

Figure 20: Delivery Building Block

With IP, a multicast group session is defined by a multicast address. The parameter used in the join and leave methods is an *rmtp_addr* object, which should return a multicast address by calling *rmtp_addr_get_ip()* [Figure 20].

When a delivery object is created, even if it does not join any multicast session, it may receive a packet, whose destination is a multicast session joined by another delivery object in the same system. Having both source and destination addresses returned by the *recv()* method allows the task manager building block [section 4.12] to find the right task(s) to process the packet.

For the method *send()*, two different APIs are provided based on the analysis in section 3.3.2.

4.10 Packet Building Block Design

A packet is used as an intermediate container for sending or receiving data. There are two ways to allocate the space to hold the data. If the space is created internally, a contiguous buffer of sufficient size should be allocated when a packet object is initialized. This type of packet is referred to as being of the linear type. In some cases, it is simple for a user to treat a packet as multiple logical segments; for example, one segment for the header, one segment for options, and another segment for data. No segment is contiguous with others. In order to avoid copying all segments into one contiguous buffer before sending, another type of packet, the vector type packet [Figure 21], is used. For this type of packet, the buffer for the data is not allocated by the packet building block. It is allocated outside and passed into the packet building block. The packet building block only contains an array of pointers to multiple segments. In comparison to the linear type packet, the vector type packet is very convenient when constructing a packet to be sent. For example, the data segment can be filled before the size of the option segment is known, even if the option segment should be placed ahead of the data segment. With linear type, such flexibility is not possible because the offset of the data segment cannot be decided without knowing the exact size of the option segment. A vector type packet can also be used when receiving data. In this case, the received data fills the segments one by one. However, in most cases, the size of the header or the option is not fixed. Before the data received are analyzed, it is not possible to set the correct size of each segment in order to preserve the logic of each segment. So, when receiving a packet, a linear type packet is used.

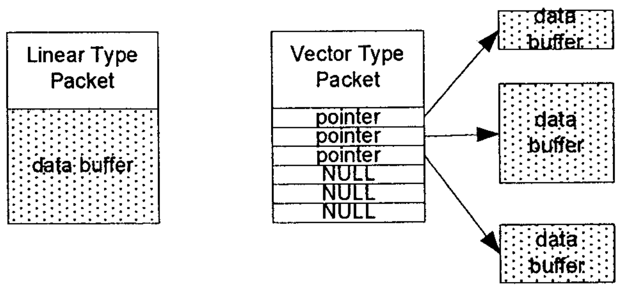


Figure 21: Linear & Vector Type of Packet

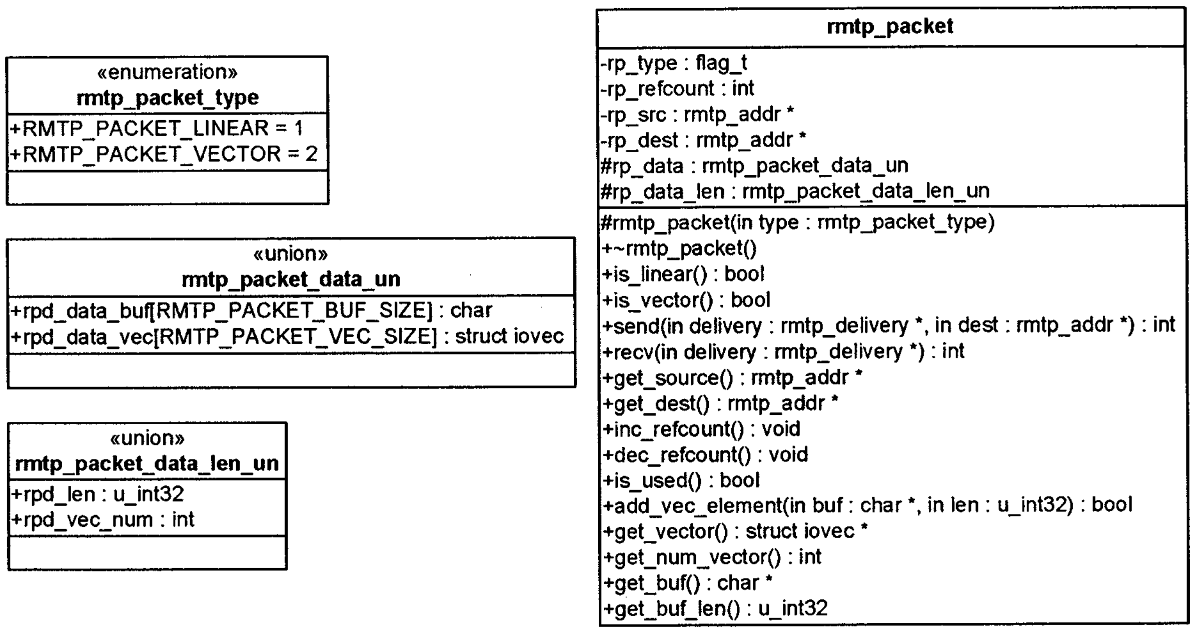

«enumeration»
rmtp_packet_type
+RMTP_PACKET_LINEAR = 1
+RMTP_PACKET_VECTOR = 2
«union»
rmtp_packet_data_un
+rpd_data_buf[RMTP_PACKET_BUF_SIZE] : char
+rpd_data_vec[RMTP_PACKET_VEC_SIZE] : struct iovec
«union»
rmtp_packet_data_len_un
+rpd_len : u_int32
+rpd_vec_num : int
rmtp_packet
-rp_type : flag_t
-rp_refcount : int
-rp_src : rmtp_addr *
-rp_dest : rmtp_addr *
#rp_data : rmtp_packet_data_un
#rp_data_len : rmtp_packet_data_len_un
#rmtp_packet(in type : rmtp_packet_type)
+~rmtp_packet()
+is_linear() : bool
+is_vector() : bool
+send(in delivery : rmtp_delivery *, in dest : rmtp_addr *) : int
+recv(in delivery : rmtp_delivery *) : int
+get_source() : rmtp_addr *
+get_dest() : rmtp_addr *
+inc_refcount() : void
+dec_refcount() : void
+is_used() : bool
+add_vec_element(in buf : char *, in len : u_int32) : bool
+get_vector() : struct iovec *
+get_num_vector() : int
+get_buf() : char *
+get_buf_len() : u_int32

Figure 22: Packet Building Block

As illustrated in Figure 22, not only does the packet building block provide the place to store the data received or to be sent, but it also provides the methods used to send or receive data. When sending data, the user of the building block provides the destination address. When receiving data, the source and destination addresses are recorded within the packet object for further use. As well, an object of the *rmtp_delivery* class is passed to both methods as the access point with the OS. The packet building block send and receive methods can choose the right format for the parameters passed to the send and receive methods of the delivery building block according to the type of the packet used.

In RMTPs, one received packet may be useful to more than one user. A packet object should not be deleted until all users have finished processing it. The reference count mechanism, which keeps track of the number of users using a packet, is implemented in the packet building block. Each user of a packet will increase the packet's reference count by one. When the user of the packet finishes processing the packet, the reference count will be decremented by one. When the reference count reaches zero, the packet can be freed.

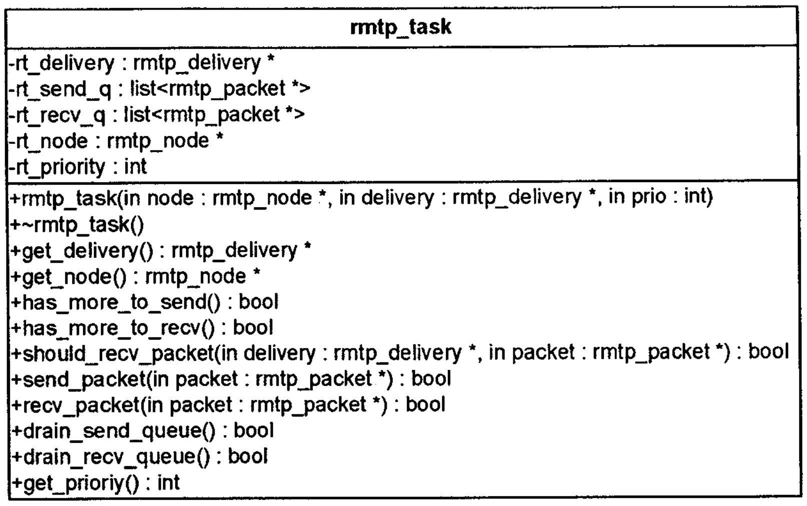
In an RMTP implementation, multiple subclasses of the *rmtp_packet* class are constructed for different packet types defined in the protocol. Every subclass provides methods to set or extract different packet fields for its corresponding protocol packet.

4.11 Task Building Block Design

The task building block is used as an access point for a user in an application. It contains a reference to an *rmtp_delivery* object. Having a reference to an object instead of an object of the *rmtp_delivery* class in a task allows all tasks to share the same *rmtp_delivery* object. The protocol specification determines whether different tasks share one *rmtp_delivery* object or use different objects.

As shown in Figure 23, each *rmtp_task* object includes two FIFO queues of packet objects. One is the receive queue and the other is the send queue. When the rate of processing packets is slower than that of receiving packets, the packets that are to be processed are stored into one of those queues. Various methods are available to handle these queues. The methods, *has_more_to_send()* and *has_more_to_recv()*, can be used to check whether or not a queue is empty. The methods, *send_packet()* and *recv_packet()*, append a new packet to a queue. The methods, *drain_send_queue()* and *drain_recv_queue()*, are called when packets can be processed. Using these two queues, the lack of synchronization between processing and receiving is resolved.

A task is not only used to send a packet, but also behaves as a handler of a packet. If different tasks use different delivery objects, the delivery objects can be used to differentiate tasks from each other. When a packet arrives through a delivery object, the packet will be handled by the task that owns that delivery object. If a delivery object is shared by multiple tasks, there should be some other means of locating the



```
rmtp_task
-rt_delivery : rmtp_delivery *
-rt_send_q : list<rmtp_packet *>
-rt_recv_q : list<rmtp_packet *>
-rt_node : rmtp_node *
-rt_priority : int
+rmtp_task(in node : rmtp_node *, in delivery : rmtp_delivery *, in prio : int)
+~rmtp_task()
+get_delivery() : rmtp_delivery *
+get_node() : rmtp_node *
+has_more_to_send() : bool
+has_more_to_recv() : bool
+should_recv_packet(in delivery : rmtp_delivery *, in packet : rmtp_packet *) : bool
+send_packet(in packet : rmtp_packet *) : bool
+recv_packet(in packet : rmtp_packet *) : bool
+drain_send_queue() : bool
+drain_recv_queue() : bool
+get_prioriy() : int
```

Figure 23: Task Building Block

right task to handle the packet.

One technique is to use the source address, source port, destination address, and destination port of the packet transmission. In some RMTPs, these four-tuples can uniquely identify what task should be used to handle the packet. Another technique is to use the contents of the packet. Since the mapping between the packet and the task is protocol dependent, a pure virtual method, *should_recv_packet()*, is provided by the task building block. The two parameters for this method, an *rmtp_delivery* object and an *rmtp_packet* object, are used to check whether a task should handle the packet.

4.12 Task Manager Building Block Design

Different data structures can be used to organize all tasks. The simplest way is to use a linked list. When a packet arrives, the task manager will descend the list seeking to match the packet with its task(s). Another method, which provides faster searching, is to use a hash table. The hash function is based on the socket ID in the *rmtp_delivery* object referred to by the task. If all tasks are referring to the same *rmtp_delivery* object, all tasks will be linked together into the same bucket of the hash table. In this case, the task manager will be the owner of this delivery object and all tasks will refer to it.

After collecting all sockets from all *rmtp_delivery* objects, the task manager building block can provide an *fd_set* object to the RMTP process. The *fd_set* object can be used, together with another *fd_set* object provided by the node manager building block, to wait for incoming packets or user requests. Listening to packets or user requests and dispatching them are detailed in the RMTP process building block design.

Figure 24 illustrates the class diagram of the task manager building block.

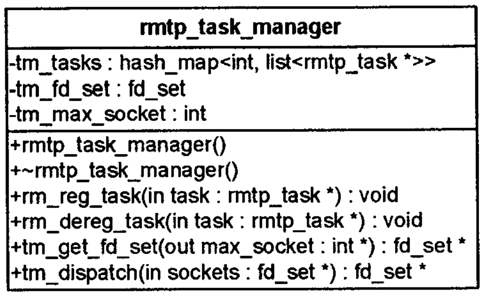

rmtp_task_manager
-tm_tasks : hash_map<int, list<rmtp_task *>>
-tm_fd_set : fd_set
-tm_max_socket : int
+rmtp_task_manager()
+~rmtp_task_manager()
+rm_reg_task(in task : rmtp_task *) : void
+rm_dereg_task(in task : rmtp_task *) : void
+tm_get_fd_set(out max_socket : int *) : fd_set *
+tm_dispatch(in sockets : fd_set *) : fd_set *

Figure 24: Task Manager Building Block

4.13 Node Building Block Design

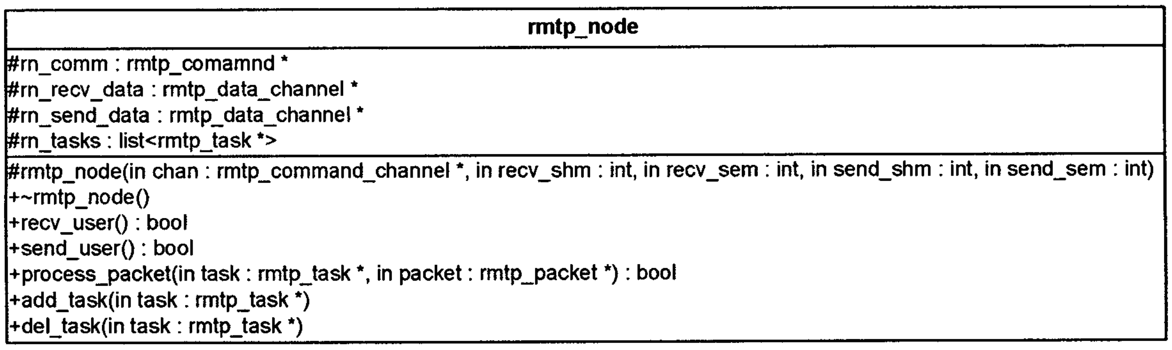
One node corresponds to one user. It is a one-to-one relationship. As in the design of the user building block, the node building block will not implement any protocol specified functions.

A node is created when a new user registers itself with the RMTP process (details are in the node manager building block design). At that time, the command channel has already been created and connected. This command channel, along with the information received in the register command, will be passed to the new node for initialization. Based on the information in the registration command, two data channels will be created and connected with the user. When the user de-registers itself from the RMTP process, its corresponding node may be deleted. Before a node is deleted, the two data channels and one command channel will first be disconnected and deleted.

Unlike the task manager building block, the node manager already knows the command channel socket ID when it creates a new node. Therefore, there is no need to register the node with the node manager. After the node is created, how to locate a corresponding node to dispatch a command from a user is described in the node manager building block. After the node manager locates the corresponding node, the pure virtual method, *recv_user()*, will be called to handle the command.

Typically, an RMTP user, which is a node within the RMTP process, has only one access point with other RMTP users in a multicast group session. Such a node includes only one task object. However, if the user has to communicate with multiple endpoints, which requires more than one delivery object, the node will create more tasks. A node maintains a list of tasks belonging to it.

When a task receives a packet, the node that owns the task will be called through the method *process_packet()* to handle the packet.



```
rmtp_node
#rn_comm : rmtp_comamnd *
#rn_recv_data : rmtp_data_channel *
#rn_send_data : rmtp_data_channel *
#rn_tasks : list<rmtp_task *>
#rmtp_node(in chan : rmtp_command_channel *, in recv_shm : int, in recv_sem : int, in send_shm : int, in send_sem : int)
+~rmtp_node()
+recv_user() : bool
+send_user() : bool
+process_packet(in task : rmtp_task *, in packet : rmtp_packet *) : bool
+add_task(in task : rmtp_task *)
+del_task(in task : rmtp_task *)
```

Figure 25: Node Building Block

The node building block provides a base class [Figure 25] for protocol specific node classes. A subclass is created for each protocol specific node. The subclass overrides the methods, *recv_user()* and *process_packet()*, to provide protocol specific user command and packet data handling. An instance of a subclass is created after a register command for the subclass' node type is received by the node manager.

4.14 Node Manager Building Block Design

In the RMTP process, only one instance of the node manager building block is needed. This instance is created when the RMTP process is created, and deleted when the RMTP process is deleted.

The main function of the node manager building block is to create nodes. When the node manager is created, it will create a TCP socket, which listens to a predefined port. The node manager will put this socket into an *fd_set* object, which will be given to the RMTP process to wait for events. Once a new user is created in an application, a command channel, in the user, will be created and connected to the RMTP process. The RMTP process notifies the node manager about this new connection. After this, a command channel is created in the RMTP process. The node manager cannot yet create a node for the new connection because the information needed to create a new node is in the register command (i.e., the node type). The node manager will maintain a list for all command channels, which is updated with the new connection. All sockets from all newly created command channels will be put into the *fd_set* object. So, when a register command arrives through a command channel, the node manager will be notified to process it using the pure virtual method *process_register()*. All parameters will be extracted from the register command and a new node will be created. Then, the command channel will be de-queued from the list and the new node will be linked into a list for all nodes. After that, the new node will handle any commands received on its command channel.

In total, there are three types of sockets handled by the node manager. There is only one instance of a socket of the first type, which is that socket listening for new TCP connections. All sockets from all command channels, from which register commands have not been received yet, are sockets of the second type. Sockets of the third type are from all command channels whose nodes have already been created. These three types of sockets are put into an *fd_set* object, which is used by the RMTP process to wait for user requests. When there is a user request in a socket, the node manager dispatches the request to its node to be handled.

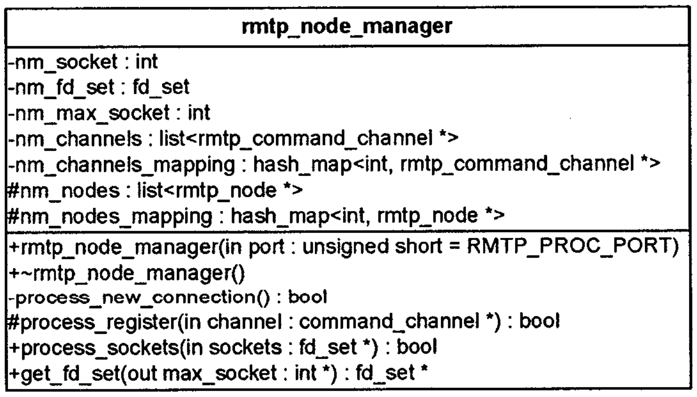

rmtp_node_manager
-nm_socket : int
-nm_fd_set : fd_set
-nm_max_socket : int
-nm_channels : list<rmtp_command_channel *>
-nm_channels_mapping : hash_map<int, rmtp_command_channel *>
#nm_nodes : list<rmtp_node *>
#nm_nodes_mapping : hash_map<int, rmtp_node *>
+rmtp_node_manager(in port : unsigned short = RMTP_PROC_PORT)
+~rmtp_node_manager()
-process_new_connection() : bool
#process_register(in channel : command_channel *) : bool
+process_sockets(in sockets : fd_set *) : bool
+get_fd_set(out max_socket : int *) : fd_set *

Figure 26: Node Manager Building Block

4.15 Timer Building Block Design

As described in the analysis of the timer building block, a timer includes: an expiry time, which sets when it will be fired; a jitter, which is applied when setting the expiry time; a handle, which is called when the timer expires; an interval, if necessary, for resetting the timer after it has expired; and its flags. All these attributes are passed to the construction method of the *rmtp_timer* class [Figure 27] through the parameters. The second parameter, *offset*, is the difference between the current time and the expiry time. Although the parameter is a relative time, the expiry time stored in the *rmtp_timer* class is an absolute time, which is calculated by adding *offset* to the current time.

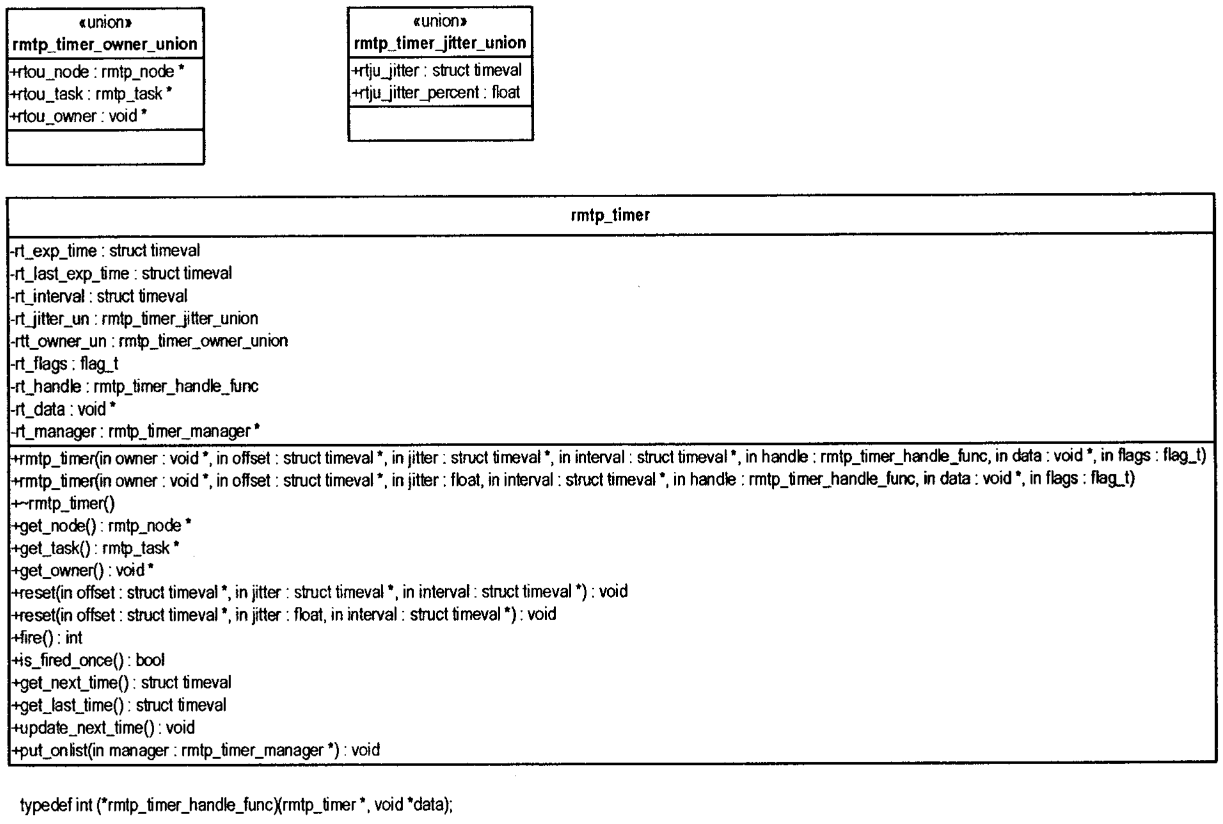

«union»
rmtp_timer_owner_union
+rtou_node : rmtp_node *
+rtou_task : rmtp_task *
+rtou_owner : void *
«union»
rmtp_timer_jitter_union
+rtju_jitter : struct timeval
+rtju_jitter_percent : float
rmtp_timer
-rt_exp_time : struct timeval
-rt_last_exp_time : struct timeval
-rt_interval : struct timeval
-rt_jitter_un : rmtp_timer_jitter_union
-rtt_owner_un : rmtp_timer_owner_union
-rt_flags : flag_t
-rt_handle : rmtp_timer_handle_func
-rt_data : void *
-rt_manager : rmtp_timer_manager *
+rmtp_timer(in owner : void *, in offset : struct timeval *, in jitter : struct timeval *, in interval : struct timeval *, in handle : rmtp_timer_handle_func, in data : void *, in flags : flag_t)
+rmtp_timer(in owner : void *, in offset : struct timeval *, in jitter : float, in interval : struct timeval *, in handle : rmtp_timer_handle_func, in data : void *, in flags : flag_t)
+~rmtp_timer()
+get_node() : rmtp_node *
+get_task() : rmtp_task *
+get_owner() : void *
+reset(in offset : struct timeval *, in jitter : struct timeval *, in interval : struct timeval *) : void
+reset(in offset : struct timeval *, in jitter : float, in interval : struct timeval *) : void
+fire() : int
+is_fired_once() : bool
+get_next_time() : struct timeval
+get_last_time() : struct timeval
+update_next_time() : void
+put_onlist(in manager : rmtp_timer_manager *) : void
typedef int (*rmtp_timer_handle_func)(rmtp_timer *, void *data);

Figure 27: Timer Building Block

Moreover, another parameter, *data*, which is of the *void** type, is user specified data, which will be passed as the second parameter to the handle of the timer, *rmtp_timer_handle_func*. This data is useful when multiple timers share the same handle. For instance, when using the same handle for re-sending multiple packets, different packet objects can be used as *data* when creating different timers. Therefore, the handle method knows which packet needs to be re-sent from the *data* parameter.

Every *rmtp_timer* object belongs to an owner. This owner can be an *rmtp_node*, *rmtp_task*, or any other object. The user of the *rmtp_timer* object should be aware of the type of its owner. The owner will be passed to the construct method of the *rmtp_timer* class as the first parameter.

After a timer is created, it should be passed to the timer manager to be included in a list ordered by expiry time. Determining which timer should be fired is also handled by the timer manager building block. When a timer expires, the method, *fire()*, will be called to trigger the timer handle.

After a timer expires, if it has been configured to include an *interval*, the method, *update_next_time()*, can be called to update the timer for its next expiry time.

A timer's next expiry time, jitter, and interval can be changed at any time by calling the *reset()* method. If the timer is already on the list of the timer manager, the timer will be re-positioned on the list after *reset()* is called.

As shown in the *rmtp_timer* class diagram, other methods are available for retrieving attributes of the timer; for example, the owner, the last expired time, etc.

4.16 Timer Manager Building Block Design

In order to be able to find all expired timers or the nearest timer, the timer manager links all timers in a list in the order of their expiry times. Since the expiry time stored in *rmtp_timer* is an absolute time, there is no need to manipulate any other timers when a timer is inserted or removed from the list.

The method, *process()*, goes through the list to find all timers whose expiry time has passed, then calls the timer handle methods for these timer. All expired timers are first removed from the list, then, those timers that have been configured with interval are updated for their next expiry times and are later re-inserted into the list.

The method, *get_next()*, can be used to retrieve the time when the nearest timer will expire.

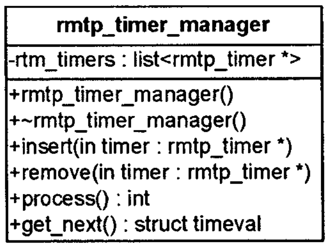

rmtp_timer_manager
-rtm_timers : list<rmtp_timer *>
+rmtp_timer_manager()
+~rmtp_timer_manager()
+insert(in timer : rmtp_timer *)
+remove(in timer : rmtp_timer *)
+process() : int
+get_next() : struct timeval

Figure 28: Timer Manager Building Block

4.17 Trace Building Block Design

Any instance of the trace building block includes multiple trace bits. Every trace bit controls one specific trace option, which is defined for a feature in a module. A trace bit is only meaningful to its instance of the trace building block and all inheriting instances. With this design, different modules are able to define different trace instances. A sub-module can create its own trace instance that inherits all trace options from a module's trace.

A trace instance is created by allocating an *rmtp_trace* object. As illustrated in the class diagram below [Figure 29], in the construct method of the *rmtp_trace* class, an existing *rmtp_trace* object can be passed as the first parameter. The new *rmtp_trace* object will inherit all trace options of the existing object. However, if the trace options of an existing instance are changed, the changes will not be propagated to any previously created instance even if that instance inherited from the instance just changed.

In the *rmtp_trace* class, the trace destination is specified by an opened file descriptor. There are two sources from which a new *rmtp_trace* object may get its trace destination. The first source is to get the trace destination from an existing object. This is achieved by passing the existing object as the first parameter, and not passing the second parameter, to the construct method of the *rmtp_trace* class. The second source to obtain the trace destination is from the second parameter of the construct method of the class. If the second parameter is specified, even when an existing instance (the first parameter) is also passed, the new trace instance will use the file descriptor, *fd*, as its destination.

After an object of *rmtp_trace* is created, the methods *enable()* or *disable()* can be called to enable or disable one or multiple trace options.

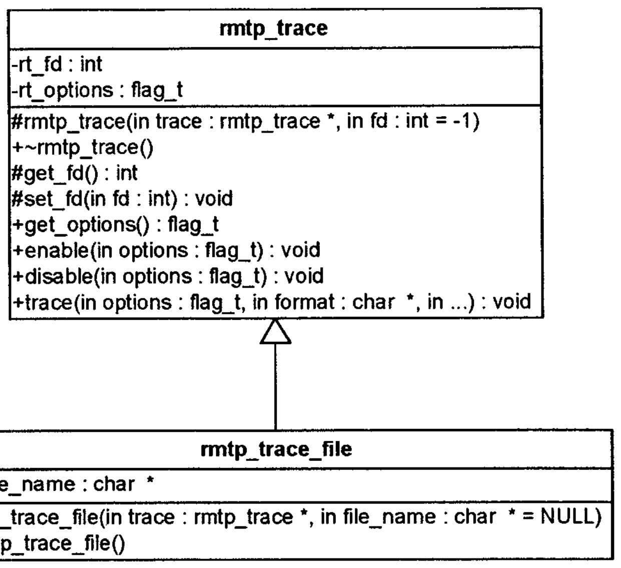

rmtp_trace
-rt_fd : int
-rt_options : flag_t
#rmtp_trace(in trace : rmtp_trace *, in fd : int = -1)
+~rmtp_trace()
#get_fd() : int
#set_fd(in fd : int) : void
+get_options() : flag_t
+enable(in options : flag_t) : void
+disable(in options : flag_t) : void
+trace(in options : flag_t, in format : char *, in ...) : void
rmtp_trace_file
-rtf_file_name : char *
+rmtp_trace_file(in trace : rmtp_trace *, in file_name : char * = NULL)
+~rmtp_trace_file()

Figure 29: Trace Building Block

The method, *trace()*, can be called to write a trace message to the trace destination. However, the trace message will only be written when at least one trace option in the first parameter is enabled in the trace instance. If none of the trace options in the first parameter is enabled, no message will be written.

A sub-class of *the rmtp_trace* class is designed for each trace destination. The sub-class is responsible for opening, and, if necessary, setting, the desired trace destination. The destination of an *rmtp_trace* instance can, at any time, be changed by calling *set_fd()*.

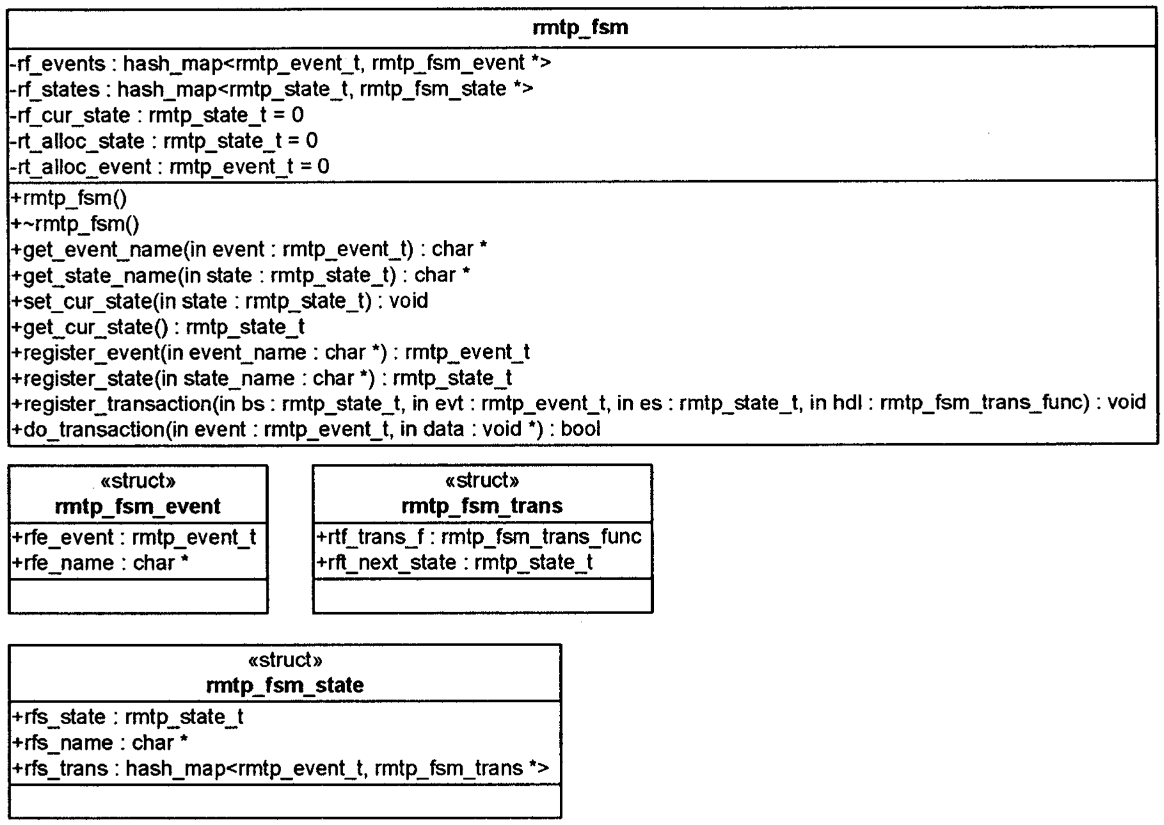
4.18 FSM Building Block Design

Three register methods are provided by the FSM building block: one for states, another for events, and lastly one for transactions. The register methods for states and events accept a name, which can be used to generate user-friendly traces, as their parameter.

The method, *do_transaction()*, is called to carry out transactions based on the current state and a given event. The second parameter to this method, *data*, which includes user specified data, will be passed on to the transaction handle method. The *data* parameter will not affect the behavior of the state machine.

The class diagram of the FSM building block is illustrated in Figure 30.

```
typedef unsigned int rmtp_state_t;
typedef unsigned int rmtp_event_t;
typedef void (*rmtp_fsm_trans_func)(rmtp_state_t cur_state, rmtp_event_t event, rmtp_state_t next_state, void *data)
```



```
rmtp_fsm
-rf_events : hash_map<rmtp_event_t, rmtp_fsm_event *>
-rf_states : hash_map<rmtp_state_t, rmtp_fsm_state *>
-rf_cur_state : rmtp_state_t = 0
-rt_alloc_state : rmtp_state_t = 0
-rt_alloc_event : rmtp_event_t = 0
+rmtp_fsm()
+~rmtp_fsm()
+get_event_name(in event : rmtp_event_t) : char *
+get_state_name(in state : rmtp_state_t) : char *
+set_cur_state(in state : rmtp_state_t) : void
+get_cur_state() : rmtp_state_t
+register_event(in event_name : char *) : rmtp_event_t
+register_state(in state_name : char *) : rmtp_state_t
+register_transaction(in bs : rmtp_state_t, in evt : rmtp_event_t, in es : rmtp_state_t, in hdl : rmtp_fsm_trans_func) : void
+do_transaction(in event : rmtp_event_t, in data : void *) : bool

«struct»
rmtp_fsm_event
+rfe_event : rmtp_event_t
+rfe_name : char *

«struct»
rmtp_fsm_trans
+rtf_trans_f : rmtp_fsm_trans_func
+rft_next_state : rmtp_state_t

«struct»
rmtp_fsm_state
+rfs_state : rmtp_state_t
+rfs_name : char *
+rfs_trans : hash_map<rmtp_event_t, rmtp_fsm_trans *>
```

Figure 30: FSM Building Block

4.19 Memory Manager Building Block Design

Using a memory pool is a standard technique that makes memory allocation and deallocation more efficient. This technique is described as follows:

A large amount of memory (usually a page) is allocated at the initialization time. When memory needs to be allocated for a structure, the real system function call is not called. Instead, one chunk of the memory within the pool is returned to the requester and actions are taken to unlink memory returned from the pool. When the memory is no longer needed, and is freed, it is returned to the memory pool instead of to the system to allow for future usage. When the memory pool is empty and memory allocation is still required, another large quantity of memory will be allocated and linked into the empty memory pool for future usage.

Even so, the problem of memory fragmentation is not solved. To solve this problem, we need to consider what will first cause memory fragmentation. Packet buffers frequently need to be allocated and freed. Whenever a packet buffer is required, for sending or receiving data, a buffer will be allocated. When the packet buffer is sent out or processed, it will be freed. The IP address is another example that calls for memory usage. In routing protocol implementations, there are thousands of routes that need to be stored in the memory. Every route corresponds to an IP address. When inserting or deleting a route to or from a routing table, memory for a route entry and its corresponding IP address will be allocated and freed. Memory usage for IP addresses in RMTPs is less than in routing protocols. However, since RMTPs provide continuous service to the application, as time passes, many more allocations and deallocations for IP addresses can be expected. This kind of memory usage, which needs to be allocated and deallocated very often, causes memory fragmentation. In order to solve the memory fragmentation problem, the following mechanism is used to handle the memory pool.

One memory pool is divided into multiple smaller, fixed-sized chunks [Figure 31]. All chunks in one memory pool are linked as a list. When a memory allocation is required, first we find a memory pool whose chunks are of the same size as is required. The first available chunk in the list from that memory pool is returned. Then, the chunk returned is removed from the list. When a memory is freed, the memory is returned to the memory pool whose chunk size is equivalent to that of the returned memory. Since the chunks are linked as a list, the time for removing or appending

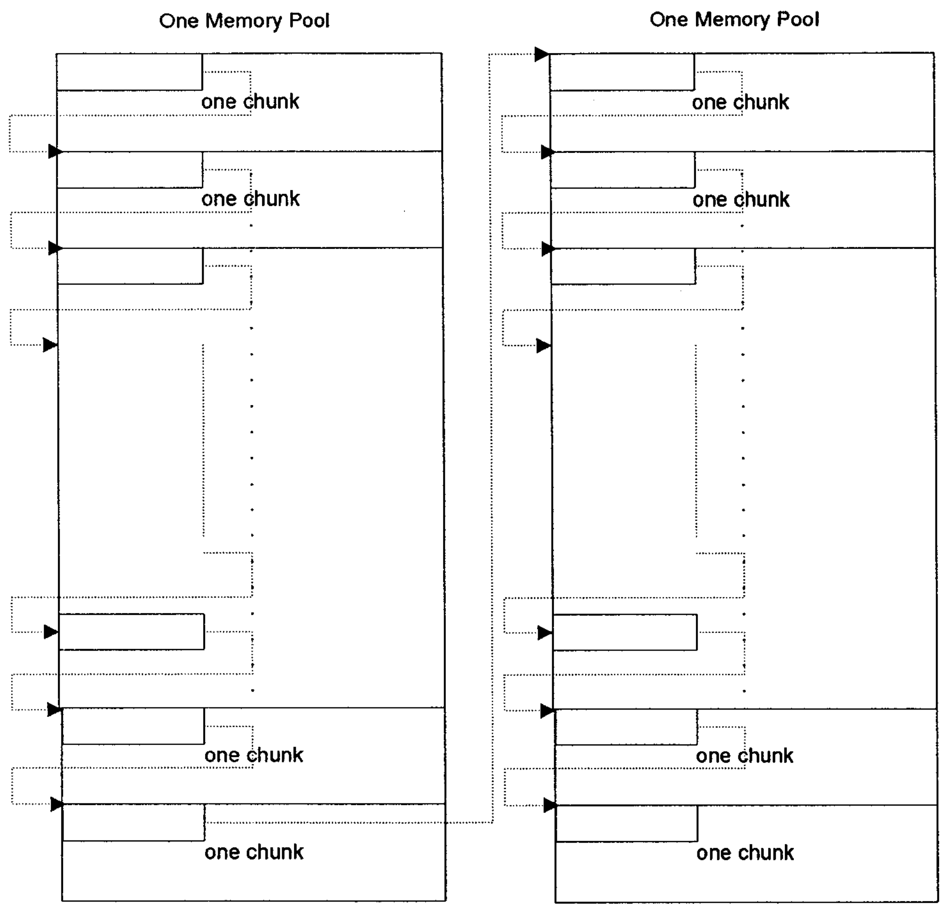


Figure 31: Memory Pool Mechanism

a chunk into the list is of O(1). Through the use of this mechanism, the memory fragmentation problem is solved.

In order to extract statistics about memory usage, we need to log every allocation and deallocation of memory. Above, we introduced several memory pools of different chunk size to solve the fragmentation problem. This mechanism can be re-used to trace the memory usage. Since all memory allocation and deallocation is done on the memory pool, a statistical counter can easily be added to the memory pool. However, if there are multiple memory usages, which require the same size chunks of memory, all memory allocation and deallocation for these usages will be performed on the same memory pool. Therefore, it becomes impossible to know the exact statistics for every memory usage. In order to solve this problem, a list of the request sources is added to every memory pool to log different memory usages [Figure 32].

Figure 33 shows the class diagram for the memory manager building block. In

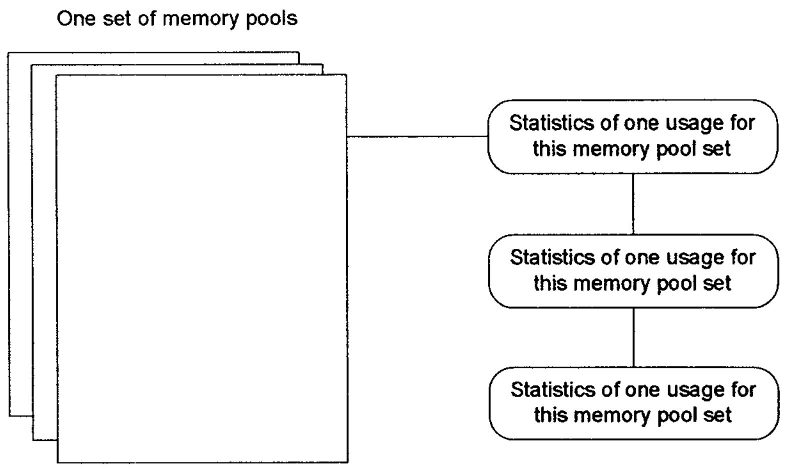

One set of memory pools
Statistics of one usage for this memory pool set
Statistics of one usage for this memory pool set
Statistics of one usage for this memory pool set

Figure 32: Memory Usage Statistics

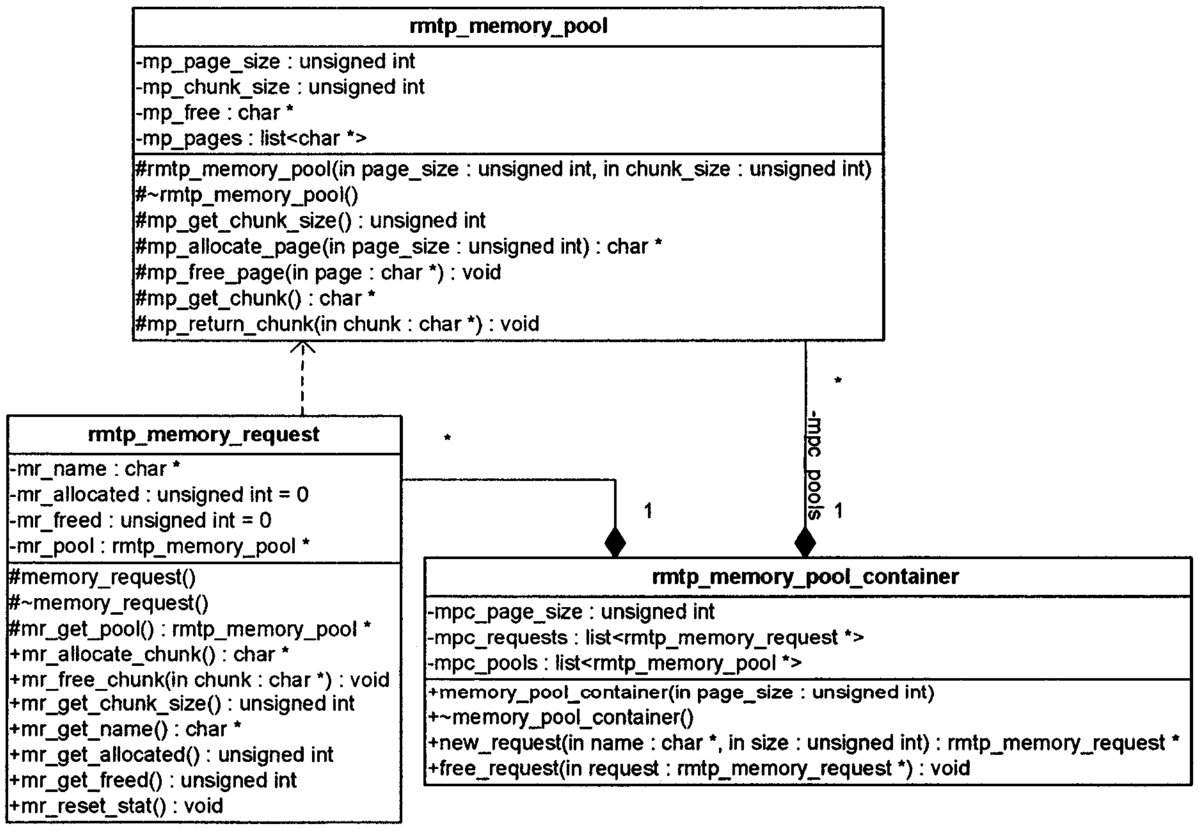

rmtp_memory_pool
-mp_page_size : unsigned int
-mp_chunk_size : unsigned int
-mp_free : char *
-mp_pages : list<char *>
#rmtp_memory_pool(in page_size : unsigned int, in chunk_size : unsigned int)
#~rmtp_memory_pool()
#mp_get_chunk_size() : unsigned int
#mp_allocate_page(in page_size : unsigned int) : char *
#mp_free_page(in page : char *) : void
#mp_get_chunk() : char *
#mp_return_chunk(in chunk : char *) : void
rmtp_memory_request
-mr_name : char *
-mr_allocated : unsigned int = 0
-mr_freed : unsigned int = 0
-mr_pool : rmtp_memory_pool *
#memory_request()
#~memory_request()
#mr_get_pool() : rmtp_memory_pool *
+mr_allocate_chunk() : char *
+mr_free_chunk(in chunk : char *) : void
+mr_get_chunk_size() : unsigned int
+mr_get_name() : char *
+mr_get_allocated() : unsigned int
+mr_get_freed() : unsigned int
+mr_reset_stat() : void
*
1
*
-mpc_pools
1
rmtp_memory_pool_container
-mpc_page_size : unsigned int
-mpc_requests : list<rmtp_memory_request *>
-mpc_pools : list<rmtp_memory_pool *>
+memory_pool_container(in page_size : unsigned int)
+~memory_pool_container()
+new_request(in name : char *, in size : unsigned int) : rmtp_memory_request *
+free_request(in request : rmtp_memory_request *) : void

Figure 33: Memory Manager Building Block

order to allocate memory, an object of the *rmtp_memory_request* class first has to be created. All the memory allocation, deallocation, and retrieving statistics are handled by this object. The *rmtp_memory_pool* class is not visible to the user of the memory manager building block. It manages all memory pools internally as described above. The *rmtp_memory_pool_container* class is used to create or free objects of the *rmtp_memory_request* class. Usually, there is only one object of the *rmtp_memory_pool_container* class existing in a system. However, there is no restriction on the number of *rmtp_memory_pool_container* objects. If multiple objects of *rmtp_memory_pool_container* exist in a system, the memory pools in different object will not be shared, even if the chunk sizes of the memory pools are the same.

The chunk returned by the memory manager building block can be used directly as any structure. If the memory manager building block is used to manage memory usage for a class, both the *new* and *delete* operators for that class should be overridden to allocate and free memory by using the memory manager building block.

4.20 RMTP Process Building Block Design

The RMTP process building block provides the central framework for the RMTP process. Figure 34 illustrates the flow chart of the method *main()*.

The main function first parses the options passed from the command line. In the method *parse()*, the following options are processed:

- -n: non-daemon mode; stays in foreground
- -t *filename*: specifies the default trace file name
- -d *filename*: specifies the dump file name
- -p *port*: specifies the port number to which the node manager will listen

The *parse()* method can be overridden, if necessary, to allow more user options.

If *-n* is not specified in the options, the RMTP process should default to daemon-mode. The RMTP process building block provides a method, *daemonize()*, to transform the process into a daemon. This method accomplishes the following:

- Create a child RMTP process, using *fork()*, and make the parent process exit.

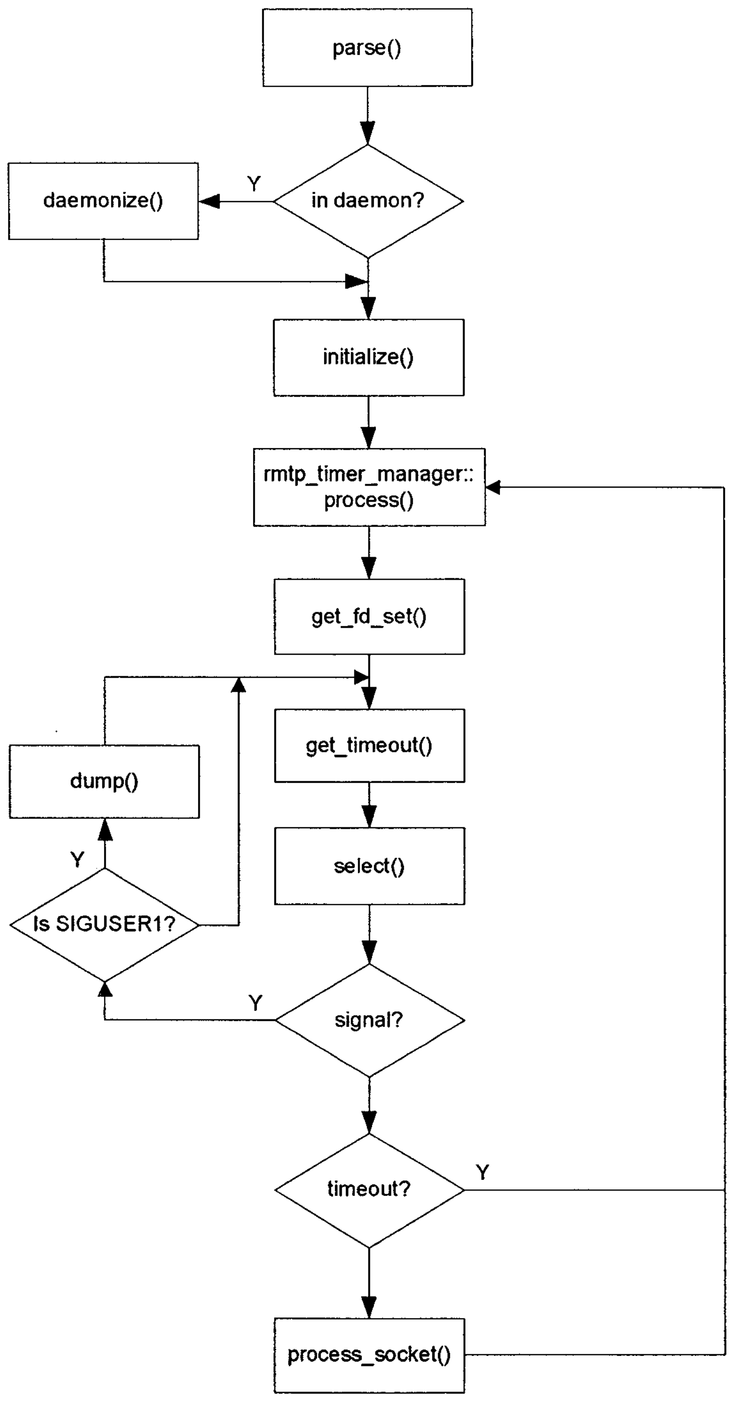


Figure 34: RMTP Process Main Function

- Call *setsid()* to create a new session, causing the process to detach from the controlling terminal.
- Change the current working directory to */var/tmp*.
- Set the file mode creation mask to 0, which will affect the permission of the dump file and trace files.

All setup procedures for the RMTP process will be included in the *initialize()* method, which can be overridden, if desired, for specific initializations.

A loop will follow the initialization. Within the loop, the *fd_set* of all sockets will be retrieved by calling the method *get_fd_set()*. This method will merge the *fd_set*s retrieved from the task manager and the node manager. A call to *select()* will be performed using the timeout value from the timer manager method *get_next()*. If there is a message in one of the sockets, *process_sockets()* will be called to determine whether the node manager or the task manager should handle the message, and then let the corresponding manager to handle the message. The timer manager will have a chance to process its already expired timers after *process_sockets()* or *select()* returns because of timeout. Finally, the RMTP process will return to the beginning of the loop to repeat the above processing.

During the main loop, in order to retrieve the status of the RMTP process, a new mechanism is developed. This mechanism could operate through a special control channel, which is separate from with the command channel and the data channel. The control channel can be implemented using a pipe or a socket. However, when bugs within the process make it loop infinitely, any command going through the pipe or socket will not be processed. Using a signal (SIGUSR1 in this case) can avoid this problem. However, by using a signal, the RMTP process can only write its status to a dump file. The RMTP process can register a handle in *initialize()* for signal SIGUSR1. Within the signal handler, a dump file will be opened and all information for the RMTP process will be written to that file. Because the RMTP process cannot directly dump every module it contains, the file descriptor of the dump file is passed to the timer manager, task manager, and node manager to allow other modules to be dumped. Writing all dump information into a file may take quite some time, so the signal handler will first fork another child and use the new process to implement the dump.

The class diagram for RMTP process building block is shown in Figure 35.

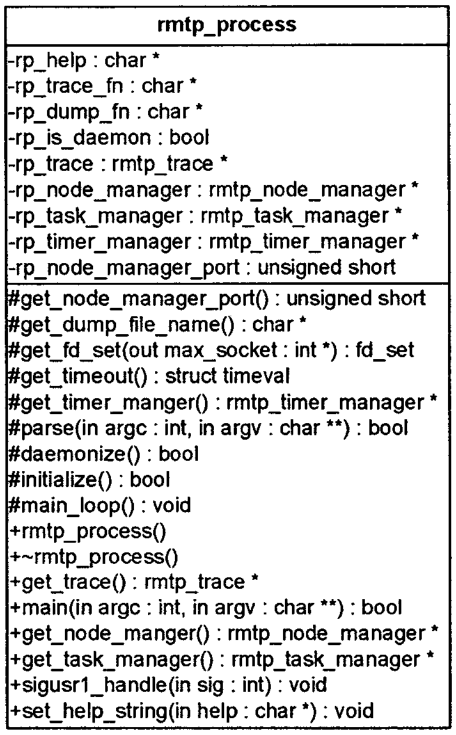

rmtp_process
-rp_help : char *
-rp_trace_fn : char *
-rp_dump_fn : char *
-rp_is_daemon : bool
-rp_trace : rmtp_trace *
-rp_node_manager : rmtp_node_manager *
-rp_task_manager : rmtp_task_manager *
-rp_timer_manager : rmtp_timer_manager *
-rp_node_manager_port : unsigned short
#get_node_manager_port() : unsigned short
#get_dump_file_name() : char *
#get_fd_set(out max_socket : int *) : fd_set
#get_timeout() : struct timeval
#get_timer_manger() : rmtp_timer_manager *
#parse(in argc : int, in argv : char **) : bool
#daemonize() : bool
#initialize() : bool
#main_loop() : void
+rmtp_process()
+~rmtp_process()
+get_trace() : rmtp_trace *
+main(in argc : int, in argv : char **) : bool
+get_node_manger() : rmtp_node_manager *
+get_task_manager() : rmtp_task_manager *
+sigusr1_handle(in sig : int) : void
+set_help_string(in help : char *) : void

Figure 35: RMTP Process Building Block

Chapter 5

The Comparison between the RMTP Implementation Building Blocks and the MTL

Although the RMTP implementation building blocks are designed based on the analysis of the requirements of RMTP implementations, those blocks are similar, in some aspects, to the MTL. In other aspects, the RMTP implementation building blocks are superior to the MTL.

5.1 Architecture

Both the MTL and the RMTP implementation building blocks share the same architecture [Figure 1, Figure 2]. Use of a central stand-alone process (daemon) and a thin library linked with the application is a compromise between implementation efficiency and simplicity. A user (*mtlif* or *rmtp_user*) is created in an application and it communicates with the stand-alone process. It is this process that implements the protocol details and exchanges information with other endpoints in the protocol. The process can simultaneously handle multiple users from different applications.

5.2 User Handling

5.2.1 In the Application

The MTL does not provide sufficient multi-user support to applications, while, in comparison, the RMTP implementation building blocks have many advantages.

- The MTL uses an unreliable protocol (UDP) between the daemon and the application. The RMTP implementation building blocks use a reliable protocol (TCP) to implement the command channel, which provides reliability for all user commands.
- The MTL transfers a predefined C structure (*user_request*) for any commands. The RMTP implementation building blocks use Type, Version, Length, and Value (TVLV) command format. This mechanism provides backward compatibility even if the command format is changed later.
- The RMTP implementation building blocks support both synchronous and asynchronous command processing. For a user, some commands can be synchronous and others can be asynchronous. Unlike the MTL, the implementation building blocks are not limited to only one at a time sending an asynchronous command. When multiple users are waiting for the results for their asynchronous commands, the user manager building block provides the method, *wait()*, to listen for all users. Moreover, the RMTP implementation building blocks support pushing information from the process to a user, which does not exist in the MTL.

5.2.2 In the Process

Inside the process, both the MTL and the RMTP implementation building blocks define an entity to represent a user. This entity is the *context* class in the MTL and the *rmtp_node* class in the RMTP implementation building blocks. However, the MTL requires all users to use only one access point with the OS, which is not the case in RMTP-II. With the RMTP implementation building blocks, any user, that is, an *rmtp_node* object, can have multiple access points (*rmtp_task*). In addition, users can share one access point just as in the MTL.

The user request dispatching mechanism used in the MTL is different from that in the RMTP implementation building blocks. In the MTL, no individual command channel between a user, within an application, and the process exists. When, in the MTL, the process receives a user request, it has to locate the correct *context* object based on the contents within the request. Therefore, user identification information has to be included within the request. For the RMTP implementation building blocks, every user has its own command channel. When the RMTP process receives a user request, the command channel can be used to find the corresponding *rmtp_node* object. Therefore, there is no need to include user identification information inside the user request.

5.3 Packet Handling

With the MTL, there exists only one access point between the OS and the daemon. When the daemon receives a packet sent by another endpoint in the protocol, a pure virtual method, *handle_new_packet()*, in the *context_manager* class will be called to process it. A concrete class, which inherits from the *context_manager* class and implements *handle_new_packet()*, will analyze the contents in the packet and distribute it to the correct *context* to process.

The RMTP implementation building blocks provide a more flexible way to handle incoming packets. When a packet arrives, it will be given to the *rmtp_task_manager*. The *rmtp_task_manager* will search through all *rmtp_task* objects to determine who is interested in the packet. An *rmtp_task* object decides, based on the access point, the payload in the packet, source, and destination address of the packet, whether it is interested in the packet. Since these four elements include all information related to the packet, this mechanism is flexible enough to meet the different requirements in various RMTPs.

5.4 Memory Management

As described in section 2.2.4, the MTL does not provide any memory management mechanism. In order to avoid memory fragmentation, the MTL pre-allocates a pool of objects at initialization. The *packet* and *context* classes are two examples of pre-allocated objects. The RMTP implementation building blocks, through the memory manager building block, resolve this difficulty.

5.5 Others

- Trace

  There is no support for the trace mechanism in the MTL. The trace building block provides a way in which to do offline analysis and debug of RMTPs and their implementations.

- Finite State Machine

  The MTL does not provide any methods to create an FSM. The FSM building block can be used to create an FSM dynamically. Different events can be defined and entered to run an FSM.

- Timer

  A more efficient and flexible timer solution is provided by the timer building block than that found in the MTL.

Chapter 6

Conclusion

Multicast is used to transmit data from one or many sources to many destinations. IP multicast services are based on the raw IP or UDP protocol. In order to provide "reliability" of IP multicast services, multiple reliable multicast transport protocols have been developed. However, because of different definitions of "reliability" in different applications, "one size does not fit all". The research focus of RMTPs has changed from individual protocols to the protocol building blocks. This change of focus provides impetus for this thesis describing reliable multicast transport protocol implementation building blocks.

The MTL was designed as a set of C++ base classes, which were used to implement RMTPs. However, due to the limits in supporting multiple users, using multiple access points, and memory management, the MTL is insufficient to be used as the RMTP implementation building blocks.

After analyzing the requirements of the RMTP implementation building blocks, we present the RMTP implementation framework. Based on the framework, all implementation building blocks are grouped into three categories: the application library, the interface with the OS, and the RMTP process.

With the utilization of synchronous and asynchronous command processing mechanisms and the user manager building block in the application library, an application can easily handle multiple users simultaneously.

In the RMTP process, all packets received from other endpoints in the protocol are handled by the task and task manager building blocks. These two building blocks provide a general locating mechanism, which is suitable for any RMTP, to map an incoming packet to its corresponding task(s). The node building block and node manager building block are designed to handle requests from all users in applications. With multiple task instances in a node instance, any user can have its own access point(s). The memory manager building avoids memory fragmentation and provides memory usage statistics.

The RMTP implementation building blocks presented in this thesis provide a framework and a full set of common components that can be used to implement a wide range of reliable multicast transport protocol.

Bibliography

[ABHM03] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-Oriented Reliable Multicast (NORM) building blocks. Internet draft (work in progress), IETF, November 2003.

[ABHM04] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-Oriented Reliable Multicast protocol (NORM). Internet draft (work in progress), IETF, January 2004.

[AM99] B. Adamson and J. Macker. Multicast Dissemination Protocol version 2 (MDPv2). Internet draft (expired), IETF, October 1999.

[Atw] J. William Atwood. A classification of multicast protocols. *IEEE Network*. (to appear in 2004).

[BLMR98] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. *Proc ACM SIGCOMM*, 1998.

[C+92] G. Chesson et al. *Xpress Transfer Protocol Definition Revision 3.6*. Protocol Engines Inc., Santa Barbara, CA, January 1992.

[Cal01] K. Calvert. Generic Router Assist - functional specification. Internet draft (expired), IETF, July 2001.

[CST01] B. Cain, T. Speakman, and D. Towsley. Generic Router Assist (GRA) building block motivation and architecture. Internet draft (expired), IETF, July 2001.

[Dee89] S. Deering. Host extensions for IP multicasting. Request For Comments 1112, IETF, August 1989.

[DEF$^+$03] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, A. Helmy, D. Meyer, and L. Wei. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol specification (revised). Internet draft (expired), IETF, September 2003.

[EFH$^+$98] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol specification. Request For Comments 2362, IETF, June 1998.

[Eri94] H. Eriksson. MBONE: the multicast backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[FJM95] S. Floyd, V. Jacobson, and S. McCanne. A reliable multicast framework for light-weight sessions and application level framing. *Proc ACM SIGCOMM 95*, pages 342–356, August 1995.

[HWK$^+$00] M. Handley, B. Whetten, R. Kermode, S. Floyd, L. Vicisano, and M. Luby. The reliable multicast design space for bulk data transfer. Request For Comments 2887, IETF, August 2000.

[Jia01] Y. Jiang. Timer management in Sandia XTP. Major report, Department of Computer Science, Concordia University, April 2001.

[KCWP00] M. Kadansky, D. Chiu, J. Wesley, and J. Provino. Tree-based reliable multicast (TRAM). Internet draft (expired), IETF, January 2000.

[KWC$^+$01] M. Kadansky, B. Whetten, B. Cain, D. M. Chiu, B. Levine, D. Thaler, S. Koh, and G. Taskale. Reliable multicast transport building block: Tree auto-configuration. Internet draft (expired), IETF, March 2001.

[KWCT01] M. Kadansky, B. Whetten, D. M. Chiu, and G. Taskale. Reliable multicast transport building block for TRACK. Internet draft (expired), IETF, March 2001.

[LG02] M. Luby and V. K Goyal. Wave and equation based rate control building block. Internet draft (expired), IETF, December 2002.

[LGV$^+$02a] M. Luby, J. Gemmell, L. Vicisano, L. Rizzo, and J. Crowcroft. Asynchronous Layered Coding (ALC) protocol instantiation. Request For Comments 3450, IETF, December 2002.

[LGV$^+$02b] M. Luby, J. Gemmell, L. Vicisano, L. Rizzo, M. Handley, and J. Crowcroft. Layered Coding Transport (LCT) building block. Request For Comments 3451, IETF, December 2002.

[LP96] K. Lin and S. Paul. RMTP: A reliable multicast transport protocol. *IEEE INFOCOMM 1996*, pages 1414–1424, March 1996.

[LV04] M. Luby and L. Vicisano. Compact Forward Error Correction (FEC) schemes. Request For Comments 3695, IETF, February 2004.

[LVG$^+$02a] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft. Forward Error Correction (FEC) building block. Request For Comments 3452, IETF, December 2002.

[LVG$^+$02b] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft. The use of Forward Error Correction (FEC) in reliable multicast. Request For Comments 3453, IETF, December 2002.

[Moy94] J. Moy. Multicast extensions to OSPF. Request For Comments 1584, IETF, March 1994.

[Moy98] J. Moy. OSPF Version 2. Request For Comments 2328, IETF, April 1998.

[PLL$^+$03] T. Paila, M. Luby, R. Lehtonen, V. Roca, and R. Walsh. FLUTE - file delivery over unidirectional transport. Internet draft (work in progress), IETF, December 2003.

[Pos80] J. Postel. User datagram protocol: Darpa internet program protocol specification. Request For Comments 768, IETF, August 1980. STD0006.

[Pos81a] J. Postel. Internet protocol: Darpa internet program protocol specification. Request For Comments 791, IETF, September 1981. STD0005.

[Pos81b] J. Postel. Transmission control protocol: Darpa internet program protocol specification. Request For Comments 793, IETF, September 1981. STD0007.

[RV97] L. Rizzo and L. Vicisano. A reliable multicast data distribution protocol based on software FEC techniques. *Proc. of The Fourth IEEE Workshop on the Architecture and Implementation of High Performance Communication Systems (HPCS'97), Sani Beach, Chalkidiki, Greece June 23-25*, June 1997.

[S$^+$95] W. Timothy Strayer et al. *Xpress Transfer Protocol Definition Revision 4.0.* XTP Forum, Santa Barbara, CA, March 1995.

[S$^+$98] W. Timothy Strayer et al. *Xpress Transport Protocol Definition Revision 4.0b.* XTP Forum, Santa Barbara, CA, June 1998.

[San96] Sandia National Laboratories, P.O. Box 969 Mailstop 9011, Livermore, California 94551-0969. *Meta-Transport Library Reference Manual*, 1.5 edition, December 1996.

[San97] Sandia National Laboratories, P.O. Box 969 Mailstop 9011, Livermore, California 94551-0969. *Meta-Transport Library User's Guide*, 1.5.1 edition, 1997.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, MA, USA, 1992.

[Ste98a] W. Richard Stevens. *UNIX Network Programming, Interprocess Communications*, volume 2. Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1998.

[Ste98b] W. Richard Stevens. *UNIX network programming: Networking APIs: sockets and XTI*, volume 1. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 1998.

[WBP$^+$98a] B. Whetten, M. Basavaiah, S. Paul, T. Montgomery, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol. Internet draft (expired), IETF, April 1998.

[WBP$^+$98b] B. Whetten, M. Basavaiah, S. Paul, T. Montgomery, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol appendices. Internet draft (expired), IETF, April 1998.

[WH01] B. Whetten and T. Hardjono. Security requirements for TRACK. Internet draft (expired), IETF, April 2001.

[WH03] J. Widmer and M. Handley. TCP-Friendly Multicast Congestion Control (TFMCC): Protocol specification. Internet draft (expired), IETF, July 2003.

[WVK$^+$01] B. Whetten, L. Vicisano, R. Kermode, M. Handley, S. Flod, and M. Luby. Reliable multicast transport building blocks for one-to-many bulk-data transfer. Request For Comments 3048, IETF, January 2001.