The Reliable Multicast Protocol Messaging Layer Application Programming Interface

Todd Montgomery and Brian Whetten
tmont@cerc.wvu.edu, whetten@cs.berkeley.edu

Introduction

This document describes the Application Programming Interface for the Messaging Layer of the Reliable Multicast Protocol. This document assumes a basic familiarity of the RMP usage model and guarantees. It describes a set of extensions to the basic API to provide message based delivery instead of packet based delivery. This requires the segmentation and reassembly of messages into packets, and the multiplexing of small messages into larger packets. It also provides message filtering based on clients and domains. Multiple clients are allowed to use an instance of RMP through this layer, and incoming messages are filtered to only the clients that have accepted them. These clients can accept to a domain, which is a subdivision of a group. If this is done, incoming packets are only delivered to the clients that have accepted that domain of the group. This layer can also associate replies with the messages that caused them, which makes blocking RPC calls very easy to implement. Finally, this layer provides a generic typing system which identifies the layer that a packet is destined for. This allows higher layers to all coexist on top of the messaging layer without conflict. All of these features, except message typing, are selectable on a per-message basis, to maintain efficiency.

There are two ways to send arbitrarily large messages with RMP. The messaging layer will do the segmentation and reassembly for the application, or if the application is using a stream oriented interface, it can break up the incoming data into packets itself. The advantages of using the messaging layer to do segmentation and reassembly is that it is very easy for the application, and that messages are delivered atomically. The disadvantage is that the whole message must be reassembled at the destinations before they will be delivered to the application. Because of this, stream oriented interfaces will usually choose to use a QoS of at least Source Ordered on its packets, and do the segmentation itself.

Packet filtering is done at two levels. First, multiple client objects can be instantiated, and each client can join its own set of groups. Incoming packets are then delivered to just the clients which have joined each packet's group. Second, each client can join a subset of a group, called a domain, instead of the whole group. Then only packets which come in addressed to that domain of the group will be delivered to that client. Each client using a RMP instance has its own instance of the Client class. Domains are represented with 16 bit unsigned integers. The applications are responsible for maintaining their own mappings of what the meanings of each domain identifier means. RMP only mandates that the value of 0 be reserved to stand for all domains. So any packet which is addressed to a domain of 0 will be delivered to all clients which have joined any domain of that group. The RMP group is still the unit of physical delivery and total ordering for messages, but some messages will get filtered out and not be delivered to all clients that are using a RMP instance.

For further details about RMP please see the associated documents [REFERENCES]. Programs using the Reliable Multicast Protocol (RMP) for communication interface with it either through the use of a set of C++ classes or through a C wrapper for the C++ interface. This document only explains the C++ interface. For further information on the exact C++ syntax and the C version of this syntax, please see the API header files.

The messaging layer has an interface that is very similar to the basic API. The functionality of this layer is primarily provided by subclassing some of the classes in the basic RMP API. These subclasses overload some of the methods of the basic API and add in some additional methods. These subclasses are denoted as mParent, with the m standing for the messaging layer.

mClient Class

Each instance of the mClient class corresponds to a client which is using this instance of RMP. Messages which are delivered by this layer will

contain a list of client objects which the message should be delivered to. The client object contains a void pointer which can be used to store a callback routine or data structure on behalf of the client. This can be used to quickly dispatch incoming messages for the client.

Creating and Destroying a Client

```
Client()
~Client()
```

Accessing the Client Pointer

```
Client(void *ClientPointer)
void *GetClientData()
```

mRMP Class

Joining a Group

```
RMPGroup *join(
    char *groupName,
            // Name of group to join
    Client *client,
            // The client to join the group
    unsigned short int domain)
            // The domain(s) of the group
        the client accepts

RMPGroup *join(
    char *groupName,
            // Name of group to join
    unsigned short int MinSizeReq,
            // Minimum number of sites
        that can be in a

            //      partition after a failure
    Client *client,
            // The client to join the group
    unsigned short int domain)
            // The domain(s) of the group
        which the client accepts

RMPGroup *join(
    char *groupName,
            // Name of group to join
    unsigned short int MinSizeReq,
            // Minimum number of sites
        that can be in a

            //      partition after a failure
    Source IPMAddress,
            // IP Multicast address of
        group
    unsigned short int IPMPort,
            // IP Multicast port of group
    unsigned char TTL,
            // IP Multicast TTL of group
    Client *client,
            // The client to join the group
    unsigned short int domain)
            // The domain(s) of the group
        which the client accepts

RMPGroup *join(
    char *groupName,
            // Name of group to join
    unsigned short int MinSizeReq,
            // Minimum number of sites
        that can be in a

            //      partition after a failure
    Source IPAddress,
            // IP address of a member of
        the group
    unsigned short int IPPort,
            // UDP/IP port of a member
        of the group
    Client *client,
            // The client to join the group
    unsigned short int domain)
            // The domain(s) of the group
        which the client accepts
```

Description: The join() operation joins either one domain or all domains for a group, for a given client. If no client using this instance of RMP is yet in the group, a RMPGroup object is created. The join() operation is asynchronous. When a join() operation completes, an event will be generated that notifies the application of this.

Arguments: The application must specify the name of the group, which resembles the textual representation of an Internet IP address. It contains up to four fields each containing an arbitrary number of alphanumeric characters. No white space or control characters are allowed. The four fields are

delimited by periods, and the string must be null terminated.

Optionally, the application can specify the minimum number of sites that must be in a partition after a failure in order for that partition to continue functioning. This is only needed by applications that are making use of the fault tolerance features of RMP. The legal values for the MinSizeReq field are:

Value	Minimum Partition Size
1-253	Equal to value
254	The majority of sites. Exactly half is not sufficient.
255	All sites. Disables fault tolerance.

The default value for the MinSizeReq field is 1.

Finally, the application may specify either a UDP/IP Multicast {address, port, TTL} tuple that should be used for the group's communication, or it can specify a UDP/IP {address, port} tuple naming a RMP process that is already a member of this group. If no address tuple is specified in a join operation, RMP uses a default mechanism to map the group name into a IP Multicast {address, port, TTL} tuple and multicasts its join request on the part of the MBone specified by this tuple.

The client identifier specifies the object which corresponds to one of the clients using this instance of RMP. The domain argument can be either 0 (meaning all domains) or some integer value.

Events Generated: Either a FORMED_OWN_GROUP or a JOINED_GROUP notification. The JOINED_GROUP notification will occur if the application was able to join an existing group. Otherwise, a FORMED_OWN_GROUP notification will occur, indicating that the application formed a group with just itself in it.

Returns: A pointer to the RMPGroup object that was created or modified. This pointer will be used when sending messages to this group, unless the message to be sent is a MultiRPC message.

Leaving a Group

```
void leave(
    RMPGroup *group,
            // The RMPGroup to leave
    Client *client,
            // The client leaving one or
            more domains in the group
    unsigned short int domain)
            // The domain(s) of the group
            which the client rejects

void leaveAll(
    Client *client,
            // The client leaving one or
            more domains in all groups
    unsigned short int domain)
            // The domain(s) of the
            groups which the client rejects
```

Description: The leave() method removes one client from one or more domains in a group. If, after the leave(), there are no more clients that are members of any domain in this group, the RMPGroup object corresponding to the group is destroyed and this RMP process is removed from the group. The leaveAll() method performs a leave() operation for each RMPGroup object that is in existence for this RMP object. Like join(), leave() and leaveAll() are asynchronous operations that do not happen immediately. A leave() operation may have 0 for its domain, in which case the client is removed from all domains for that group. If a client has joined all domains in a group, by using a wildcard domain argument of 0, leaving a single sub-domain will have no affect.

Arguments: The leave() operation must pass in a pointer to the RMPGroup object corresponding to the group to be left. It must also pass in a Client pointer, and a domain identifier.

Events Generated: A LEFT_GROUP notification will be generated when each leave() operation completes. If the group left was the last group the application was a member of, a LEFT_ALL_GROUPS event will also be generated. After the LEFT_ALL_GROUPS notification, it is safe to destroy the RMP object.

Returns: Nothing.

Getting a Buffer

```
mRMPMessage *returnBuffer(
        RMPMessageType
messageType,        // Type of message
        BOOL zero)
    // Whether the buffer should be
```

zeroed

Description: This call returns a preallocated Data Packet from a pool of packets. The application can fill in and then send the buffer. With the messaging layer, it can also string together a number of buffers in a single message, and send them all together.

Arguments: Whether the buffer should be zeroed before being returned to the application, and the type of message, either DATA_PACKET (for normal communication between Group Members) or NON_MEMBER_DATA_PACKET (for MultiRPC communication).

Events Generated: None

Returns: A mRMPMessage which can be filled in and sent.

Releasing a Buffer

```
void releaseBuffer(
        mRMPMessage *msg)
    // mRMPMessage to return to
preallocated pool
```

Description: This call returns a preallocated mRMPMessage to the pool of packets.

Arguments: mRMPMessage object to return to pool.

Events Generated: None.

Returns: None.

Sending a Message From a Non Member of a Group

```
SendStatus sendMultiRPC(
    Source IPMAddress,
            // IP Multicast address of
        group
    unsigned short int IPMPort,
            // IP Multicast port of group
    unsigned char TTL,
            // IP Multicast TTL of group
    char *groupName,
            // The group name
    char *msg,
            // The data to send
    unsigned short int size,
            // The size of the data to
        send
    QOS qos,
            // The QoS with which to
        deliver the data
    Handler handler,
            // The handler number for
        this message
    unsigned short int domain)
            // The domain to send the
        message to

SendStatus sendMultiRPC(
    Source IPAddress,
            // IP address of a member of
        the group
    unsigned short int IPPort,
            // UDP/IP port of a member
        of the group
    char *groupName,
            // The group name
    char *msg,
            // The data to send
    unsigned short int size,
            // The size of the data to
        send
    QOS qos,
            // The QoS with which to
        deliver the data
    Handler handler,
            // The handler number for
        this message
    unsigned short int domain)
            // The domain to send the
        message to

SendStatus sendMultiRPC(
    char *groupName,
            // Name of group to send to
    char *msg,
            // The data to send
    unsigned short int size,
            // The size of the data to
        send
    QOS qos,
            // The QoS with which to
        deliver the data
    Handler handler,
```

```
        // The handler number for
    this message
unsigned short int domain)
        // The domain to send the
    message to

SendStatus sendMultiRPC(
    mRMPMessage *msg)
        // mRMPMessage to send
```

Description: This sends a message from a non-member of a group to one or more domains in the group. It will be delivered to the members of the group with the given QoS, and may receive a reply from a handler, if so requested. All packets that are sent with a QoS of at least UNORDERED receive a positive acknowledgment from the group they are being sent to. In addition, packets which request a handler may be responded to by a member of the group.

Besides the addition of the domain argument, the key difference between these calls and those in the basic API version is that the length of the buffer to be sent can exceed that of a single packet.

Arguments: Each version of the call requires the message, the length of the message, the QoS level to deliver it with, the handler number for the message, and the domain that the message is to be sent to. The domain can be set to 0, in which case it will be delivered to all domains. In addition to these arguments, the group itself must be specified in one of three ways. If the group name is given, RMP will map this into a {IP Multicast address, UDP port, TTL} tuple to send the packet to. Alternatively, the tuple can be specified explicitly. Finally, the {IP address, UDP port} tuple for a known member of the group can be specified. See send() for more details on these parameters.

Events Generated: An optional reply may be sent in response to a multi-RPC packet. Also, a packet may occasionally fail to get an acknowledgment after a number of tries. In this case, a SEND_FAILURE notification is generated. By default, loop back is always performed on MultiRPC messages.

Returns: The status of the send operation. See the RMPGroup send() method for more details.

RMPGroup Class

Sending a Message From a Member of a Group

```
SendStatus send(
    char *msg,
        // The data to send
    unsigned short int size,
        // The size of the data to
    send
    QOS qos,
        // The QoS with which to
    deliver the data
    Handler handler,
        // The handler number for
    this message
    unsigned short int domain,
        // The domain to send the
    message to
    BOOL loopBack)
        // Should this be delivered to
    ourself?

SendStatus send(
    mRMPMessage *msg,
        // mRMPMessage to send
    BOOL loopBack)
        // Should this be delivered to
    ourself?
```

Description: This method sends a given message to a group that the sender is a member of. The data can be of arbitrary length. If a send() operation is attempted to a group that the sender is not a member of, an error will occur. If a send() operation is attempted to a group that the sender is in the process of joining, the packet will be enqued for transmission once the join operation is complete. If too much data is sent at once, RMP may block the transmission of further data until the current data has been sent. Each call to send() should check the return value to see if the operation was successful.

Besides the addition of the domain argument, the key difference between this calls and those in the basic API version is that the length of the buffer to be sent can exceed that of a single packet.

Arguments: Each send() operation must include the

data to be sent and the size of the data. In addition to these parameters, the following parameters can also be specified.

The qos field specifies which guarantees the packet will be delivered with. The valid types are given below. Please see the RMP description documents for a further description of what guarantees each QoS level provides. Except for the QOS levels of MAJORITY RESILIENT and TOTALLY RESILIENT, the defined QOS levels are strictly hierarchical, with each level including all guarantees of smaller QOS levels. The maximum K RESILIENT level for any given ring is equal to the number of members of the ring. If a packet is specified with a higher QOS than this, it will be delivered as if its QOS was equal to TOTALLY RESILIENT. The valid values for the quality of service are:

Val	QoS name	Description
1	UNRELIABLE	Unreliable, completely unordered delivery
2	UNORDERED	Reliable, completely unordered delivery
3	SOURCE ORDERED	Reliable, ordered with respect to each source
4	CAUSALLY ORDERED	Reliable, causally ordered (Not implemented)
5	TOTALLY ORDERED	Reliable, totally ordered
6-29	K RESILIENT	Reliable, totally ordered, and K-resilient fault tolerant
30	MAJORITY RESILIENT	Reliable, totally ordered, majority-resilient fault tolerant
31	TOTALLY RESILIENT	Reliable, totally ordered, completely fault tolerant

The handler field selects the handler, if any, for a data packet. There are six mutually exclusive locks included with each group. Based on the value of the handler field, the process that holds one of these locks may be responsible for providing a response to it. Requesting that a packet be handled guarantees that at most one process will handle it. The valid values for handler are:

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

The loopBack field specifies whether or not the packet should also be delivered to the application that sent it. Consistent shared applications can be developed by sending packets with a QoS of at least TOTALLY_ORDERED and with loopBack set to TRUE. If application events that occur are sent out as totally ordered messages and only acted upon when the associated message is returned to the application, a consistent series of events will be observed and operated upon at all members of the group.

These last three arguments have default values. The default for qos is TOTALLY ORDERED. The default for handler is 0, for no handler. The default for loopBack is FALSE (0), for no loop back.

Events Generated: None.

Returns: The send() method returns the status of the operation. RMP provides a transmission window for each sender which limits the amount of data that each sender can have in transit to each group at a given time. This is necessary in order to avoid overrunning the destinations and to avoid network congestion. RMP will provide limited buffering of data on behalf of the application. However, if the sender overruns both the transmission window and the buffering area, the send() operation will be rejected. To help applications make intelligent decisions about their transmission patterns, RMP provides flow control feedback from a send() operation. The possible return values are:

Val	Status name	
-2	DATA_BUFFERED	The transm
-1	FLOW_CONTROL_WARNING	The transm
0	FLOW_CONTROL_OK	The data wa
1	FLOW_CONTROL_FULL	The data co
2	BAD_TOKEN_RING_ERROR	The data cou group obj
3	ARGUMENT_ERROR	
4	SYSTEM_ERROR	An er

mRMPMessage Class

The mRMPMessage class provides an interface to a set of preallocated buffers that can be used to send data packets from either a member or a non member. Members of this class are very similar to the Message class defined in the basic API, except that they can be strung together in a chain. This allows messages composed of multiple buffers to be sent

and returned. The other main difference is that these messages can have a domain associated with them when they are sent.

mRMPMessage Access Methods

```
RMPMessage
*returnNextMessage()
void setNextMessage(RMPMessage
*msg)
```

Description: This is used to get and set the next buffer in the chain. This is used to string multiple buffers together in a row for messages that are longer than the maximum packet length. All applications that use the messaging layer should check to see if a received mRMPMessage instance has its next pointer set, with the use of the returnNextMessage. If so, the chain of mRMPMessage buffers together make up a single message.

```
unsigned short int returnDomain()
void setDomain(unsigned short int
domain)
```

Description: This is used to get and set the domain for a message.