NASA-CR-197762

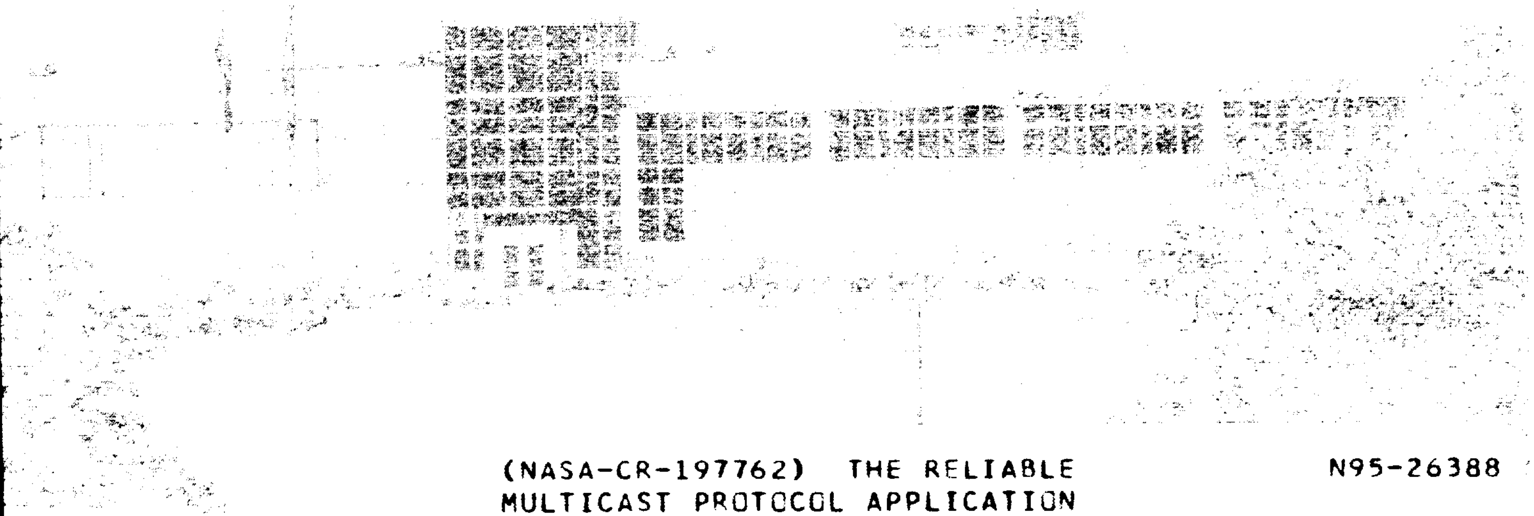
NASA-IVV-94-007
WVU-SRL-94-007
WVU-SCS-TR-95-7
CERC-TR-RN-94-014

NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

IN-61-CR
48510
p. 20

The Reliable Multicast Protocol Application Programming Interface

by Todd Montgomery and Brian Whetten



National Aeronautics and Space Administration

West Virginia University

According to the terms of Cooperative Agreement #NCCW-0040, the following approval is granted for distribution of this technical report outside the NASA/WVU Software Research Laboratory

[illegible] 4-10-95
George J. Sabolish Date
Manager, Software Engineering

[illegible] 4-19-95
John R. Callahan Date
WVU Principal Investigator

The Reliable Multicast Protocol 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 Berkeley/WVU implementation of the Reliable Multicast Protocol. This transport layer protocol is implemented as a user library that applications and software buses link against. This document assumes a basic familiarity of the RMP usage model and guarantees. 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 C++ interface for RMP primarily involves objects of four classes: RMP, RMPGroup, RMPEvent, and RMPMessage. The RMP object is the control structure for all of the RMP communication. Only one RMP object can exist per UNIX process. RMP objects contain zero or more RMPGroup objects, each of which correspond to a single group that the application using RMP is a member of. Please note that in some other documents, groups are called Token Rings. The RMP object returns a set of RMPEvents to the application, which corresponds to a data packet (sent to a RMPGroup) or to an application notification. Application notifications notify the application of group membership changes and errors. In order to save the overhead of having to copy the application's message from the application's buffer to a RMP buffer, RMP allows applications to request RMPMessage objects, which correspond to a single RMP data packet.

This API is structured in terms of data packets (also referred to as messages). In the current version, a data packet can usually be up to a little less than 8K long. This is necessary in order to support unreliable and unordered QoS levels, as well as handlers for messages. For most applications, which at least want messages to be ordered with respect to their source, it is a simple matter to provide a stream oriented interface on top of this, if desired.

The only restriction that RMP places on an application using it is that the application must use RMP to set alarms on its behalf instead of calling alarm() and ualarm() itself. An application can register as many alarm types with RMP as it wishes to, each with its own constant identifier, a callback function, and a time when the alarm will expire.

RMP must periodically get control from the application that is using it. This control can be granted either explicitly or implicitly. Explicit control is granted to RMP by applications that use an event driven paradigm. In these applications, the program is driven by an event loop. When input arrives on a file descriptor, a file descriptor becomes ready for output, or a RMP message arrives, the program is notified of these events. It then processes these events and returns control to the event loop. When an event driven application uses RMP, it must allow RMP to control, or "own" the actual event loop. Each time the application is finished processing an event, it passes control to RMP along with a list of the input and output file descriptors it is interested in. When an event occurs on one of these file descriptors or when a message comes in from a group the application is a member of, RMP will return this event to the application. Alarms that have been registered with RMP will automatically call the associated callback function in the application when they go off.

Event driven applications are the most efficient way to use RMP. However, traditional monolithic programs can also be used with RMP by granting implicit rather than explicit control to RMP. This is done by having RMP set periodic alarms that wake it up and allow it to process the events it is concerned with. This type of program can either periodically poll RMP to see if any RMPEvents have arrived, or it can set a callback function up that asynchronously calls a handler function in the application when an event arrives. At most one callback function can be called at a time by RMP, so a correctly programmed application does not have to worry about reentrancy.

RMP Class

Constructor

```
    RMP(
        int operatingFlag)              // Event driven or monolithic programming model
```

Description: This creates an RMP object, allocates space for communication, opens up a UDP/IP port to send and receive unicast packets on, and starts a periodic callback alarm if IMPLICIT_CONTROL is specified.

Arguments: The operatingFlag must be set to either EVENT_DRIVEN or IMPLICIT_CONTROL. The EVENT_DRIVEN flag notifies RMP that the application instantiating it intends to use an event driven paradigm and regularly grant it explicit control. The IMPLICIT_CONTROL flag notifies RMP that it will not get explicit control, and so must use alarms to regularly grab control from the application.

Events Generated: None.

Returns: Nothing.

Example: Below is the framework for a sample event driven application.

```
    RMP communicator(EVENT_DRIVEN);
    RMPEvent *event;

    for (;;) {
        event = communicator.RMPLoop()
        Handle event
        }
```

Destructor

```
    ~RMP();
```

Description: This destroys the RMP object, along with any RMPGroup and RMPEvent objects that it contains. This should not be called until after a leaveAll() has successfully completed, or else the other members of the groups this application was a member of may see it as a failed site.

Arguments: None.

Events Generated: Nothing.

Returns: Nothing.

Joining a Group

```
RMPGroup *join(
    char *groupName)              // Name of group to join

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
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

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
```

Description: The Join() operation creates a RMPGroup object and starts the join operation to add this process to a group. If the application is already a member of a group with this name, the pointer to this group is returned. 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 six fields each containing an arbitrary number of alphanumeric characters. No white space or control characters are allowed. The first four fields are delimited by three periods with the fourth field ending in a colon , the next field is separated by a colon and the last field must be null terminated. The first four fields resemble an Internet IP Address, but it must have all four fields. These fields represent the IP Multicast Address of the group. The first colon separated field is the IP Multicast Port representation. And the last field is the TTL of the group from this site.

An example group name: <oodb.cs.wvu.edu:port0:region>. The < and > mark the string beginning and end. In the Alpha version of the implementation, this transformation is done by the use of a hash function to hash each field. This will soon be replaced by something much more dynamic and cleaner to use. For a list of the supported fields see the header files.

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.

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. 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

void leaveAll()
```

Description: The leave() method removes this application from a group and destroys the RMPGroup object corresponding to it. 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.

Arguments: The leave() operation must pass in a pointer to the RMPGroup object corresponding to the group to be left.

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.

Looking Up a Group Object

```
RMPGroup *getTokenRing(
    char *groupName)                    // Name of group to get
```

Description: This looks up a group object for a group name.

Arguments: The null-terminated group name to look up.

Events Generated: None.

Returns: A group object, or NULL if the group object for this name could not be found.

Granting RMP Explicit Control

```
RMPEvent *RMPLoop()

RMPEvent *RMPLoop(
    int width,
    fd_set *inreadfds,
    fd_set *inwritefds,
    fd_set *inexceptfds,
    struct timeval *timeout)
```

Description: If RMP is started with the EVENT_DRIVEN flag, control must be returned to RMP regularly by calling the RMPLoop() method.

Arguments: The first version of RMPLoop requires no arguments, and blocks until an RMPEvent occurs. The second version contains the arguments normally used in a select() call. When used in this way, the RMPLoop functions in the same way as a select() function, except that it may also return a RMPEvent. On return from the select version of RMPLoop, the fd_set structures are set to the list of file descriptors that the application requested access to, and that are now ready for I/O.

Events Generated: None.

Returns: A RMPEvent, if one occurred within the timeout period.

Example: The following example allows the application to monitor a set of input file descriptors with the blocking RMPLoop call.

```
fd_set userreadfds;
int userFD = STANDARD_IN;
RMPEvent *event;

for (;;) {
    FD_ZERO(&userreadfds);
    FD_SET(userFD, &userreadfds);
    event=RMPLoop(FD_SETSIZE, &userreadfds, NULL, NULL, NULL);
    if (event != NULL) {
        handle RMPEvent
        }
    if (FD_ISSET(userFD) {
        handle application interface messages
        }
    }
```

Polling for RMP Events

```
        RMPEvent *poll()
```

Description: Applications that do not use an event driven paradigm will usually poll for RMPEvents. This is done with the poll() method, which returns a RMPEvent if one has occurred. RMPEvents are queued until retrieval by the poll() method.

Arguments: None.

Events Generated: None.

Returns: An RMPEvent, if one occurred, or NULL if not.

Registering Callback Functions

```
        setEventHandler(
            RMPEventType type,              // Type of Event to handle
            void (*func)(RMPEvent *))       // callback function to call when event occurs
```

Description: If the application would like to trap RMP events (such as membership changes, handler requests, and message deliveries) and process those events asynchronously with the rest of the application control flow, then it may register callback functions to be executed when events occur. When a registered event occurs, RMP calls the function and passes it a pointer to the RMPEvent. It is the callback function's responsibility to release the RMPEvent eventually, preferably as soon as possible. Only one callback function can be in progress at a time, so the event handler routines do not have to be reentrant.

Arguments: The event type to trap, and the function to call when the event occurs. EventType may also be assigned to CALLBACK_ALL. In this case, this function is designated as the handler Events. The func argument may be assigned the value CALLBACK_IGNORE, which can be used to tell RMP to ignore all ocurrences of this event type. When this is done, RMP discards these events as soon as they are received. Any events that do not have an associated callback funtion and which are not set to be ignored are enqued for retrieval by the poll() method.

Events Generated: None.

Returns: Nothing.

Setting an Alarm

```
    void setTimer(
        unsigned long int time,
        unsigned short int number,
        void *obj,
        void (*func)(void *))

    void setPeriodicAlarm(
        unsigned long int time,
        unsigned short int number,
        void *obj,
        void (*func)(void *))
```

Description: If the application wants to set an alarm it must do so through RMP with one of these functions. The setTimer() method schedules a one shot timer to go off in time milliseconds. When the alarm goes off, it will execute the func parameter, passing obj to it. The setPeriodicAlarm() method is exactly the same except that it schedules a periodic alarm to occur every time milliseconds instead of just once.

Arguments: The time period, a constant identifier for it, the callback function, and a single 32-bit argument to be passed to the function.

Events Generated: None.

Returns: Nothing.

Clearing an Alarm

```
    int cancelAlarm(
        unsigned short int number,
        void *obj,
        void (*func)(void *))
```

Description: This cancels all of the pending alarms with the given constant callback function and argument.

Arguments: The callback function and the argument to the callback function.

Events Generated: None

Returns: The number of alarms that were cancelled.

Querying an Alarm

```
    int queryAlarm(
        void *obj,
        void (*func)(void *))
```

Description: This checks to see how many pending alarms exist with the given callback function and argument.

Arguments: The callback function and the argument to the callback function.

Events Generated: None

Returns: The number of pending alarms that exist that match the passed in arguments.

Getting a Buffer

```
        RMPMessage *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.

Arguments: Whether the buffer should be zeroed before being returned to 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: An RMPMessage which can be filled in and sent.

Releasing a Buffer

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

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

Arguments: RMPMessage object to return to pool.

Events Generated: None

Returns: None

Sending a Packet 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

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

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
```

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

Description: This sends a packet from a non-member of a group to 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.

Arguments: Each version of the call requires the message, the length of the message, the QoS level to deliver it with, and the handler number for the message. In addition, 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. The version of the call which takes a RMPMessage also takes these parameters, but in that case the parameters must be filled in to the RMPMessage before being sent.

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

Requesting a Handler Lock

```
void requestHandler(
    Handler handler)                    // The lock number to attempt to get
```

Description: This requests a handler lock. Each group contains six handler locks, numbered 1 through 6. When a handler lock is requested, RMP will attempt to obtain this lock for the application. Handler locks are mutually exclusive, so one will only be granted if no other application is holding it. Requesting a handler lock is an asynchronous operation. Once the success or failure of this operation has been determined, an appropriate event will be returned to the application.

Arguments: The number of the requested handler lock.

Events Generated: A HANDLER_RECEIVED notification will be generated if the application received the handler lock. A HANDLER_DENIED notification will be generated if the application did not receive the handler lock.

Returns: Nothing

Querying a Handler Lock

```
BOOL holdingHandler(
    Handler handler)                    // The handler number to check
```

Description: This method checks to see whether or not this application is holding a given handler for this group.

Arguments: The handler number for the handler to check.

Events Generated: None.

Returns: TRUE if this application holds the handler, FALSE otherwise.

Releasing a Handler Lock

```
void releaseHandler(
    Handler handler)                    // The lock number to release
```

Description: This attempts to release a handler lock for a given group. If the specified lock is held by this application, it is released. Once released, an event notifying the application of this is generated. The application is typically expected to handle requests until after this notification is received.

Arguments: The number of the handler lock being released.

Events Generated: A HANDLER_RELEASED notification is generated if the handler lock is released. A MEMBERSHIP_CHANGE notification is generated if the handler lock is not released.

Returns: Nothing.

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
    BOOL loopBack)              // Should this be delivered to ourself

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

Description: This method sends a given data packet 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.

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	Description
-2	DATA_BUFFERED	The transmission window is now full, so the message was buffered for later retransmission.
-1	FLOW_CONTROL_WARNING	The transmission window is full and the buffering space is almost exhausted.

0	FLOW_CONTROL_OK	The data was accepted without overflowing the transmission window
1	FLOW_CONTROL_FULL	The data could not be sent because the buffering area is full
2	BAD_TOKEN_RING_ERROR	The data could not be sent because there was an error with the group object passed in. The sender may no longer be a member of the given group.
3	ARGUMENT_ERROR	Some other argument was in error.
4	SYSTEM_ERROR	An error occurred with RMP. Should not occur.

Checking the Flow Control Status of a Group

```
FlowControlStatus queryFlowControl(
    unsigned short int packetToSend)        // The size of the packet to test for
```

Description: This acts as a way of querying the Flow Control of a RMPGroup to see if a given packet size would be acceptable to send. In addition, by setting the packetToSend size to zero, the current flow control status of the group can be determined.

Arguments: The size of the data to test for. If this argument is non equal to zero, it will be increased to account for the header size.

Events Generated: None.

Returns: A subset of the values usually returned by the send() method. The values are DATA_BUFFERED, FLOW_CONTROL_WARNING, FLOW_CONTROL_FULL, and FLOW_CONTROL_OK.

RMPEvent Class

Replying to Just the Sender of a Message

```
SendStatus replyToSender(
    char *reply,                    // The data to reply with
    unsigned long int size)         // The size of the data to send
```

Description: A reply can be generated in response to any message. It is most typically used by a group member named as the handler for a particular message. If the reply is to a member of a group, the reply is sent to the entire group. Otherwise it is sent to just the non-member that sent the message.

Arguments: The data for the reply and the size of that data.

Events Generated: None.

Returns: The same values as the RMPGroup send() method.

Note: Not yet supported.

Releasing a RMPEvent Object

```
void release()
```

Description: When the application is done with an RMPEvent, it should use the release() method to return it to RMP. It can not free it up directly.

Arguments: None.

Events Generated: None.

Returns: Nothing.

RMPEvent Access Methods

```
RMPGroup *returnRMPGroupForEvent()
```

Description: Returns a pointer to the RMPGroup of the event.

```
RMPEventType returnRMPEventType()
```

Description: Returns the type of RMPEvent object. An RMPEvent is one of the below types.

Event Type Name	Description
MESSAGE	A message from another RMP process.

MEMBERSHIP_CHANGE (Group List)	Notification of a membership change in a group. This can be either that a different process has joined the ring, that a different process has left the ring, that a handler lock has been granted, or that a handler lock has been released.
HANDLER_RECEIVED (Group List)	This application received a Handler Lock.
HANDLER_REJECTED (Group List)	A request for a Handler Lock was rejected.
HANDLER_RELEASED (Group List)	This application released a Handler Lock
FAILURE_RECOVERY (Group List)	A Failure has occurred and been recovered from.
FORMED_OWN_GROUP (Group List)	In attempting to join a group, no other members of the group existed, so we created our own.
JOINED_GROUP (Group List)	This application has just joined the group
LEFT_GROUP (Group List)	This application has finished leaving the group
LEFT_ALL_GROUPS	This application is not longer a member of any group.
SEND_FAILURE	A packet that was being sent to a group may not have gotten through with the desired QoS. This occurs for multi-RPC packets that do not get an acknowledgment after a set number of retries.
ATOMICITY_VIOLATED	Due to one or more failures, one or more packets sent to the group may have had their atomicity and ordering guarantees violated.

Note: The (Group List) denoted on some of the Events above signify that a group list is returned in the RMPEvent class. See below for details on accessing a group list.

RMPEvent Access Methods - MESSAGE type

These access methods apply to RMPEvents of type MESSAGE.

unsigned short int returnMessageLength()

Description: Returns the size of the data of the message in bytes.

Handler returnMessageHandler()

Description: Returns the Handler of the message.

BOOL returnWeShouldHandle()

Description: Returns TRUE if this application currently holds the handler lock that corresponds to the handler requested by this packet. If TRUE, this application is typically expected to respond to the packet.
(NOTE: Added)

```
QOS returnMessageQOS()
```

Description: Returns the QoS of the message.

```
Source returnMessageSourceIP()
```

Description: Returns the IP Address of the message source.

```
unsigned short int returnMessageSourcePort()
```

Description: Returns the Port of the message source.

```
char *returnData()
```

Description: Returns a pointer to the data of the message.

RMPEvent Access Methods - Application Notifications

Application notifications that return a group list can be accessed with the following methods.

```
unsigned short int returnNumMembersInList()
```

Description: Returns the number of members in the group.

```
BOOL multicastCapable(int entry)
```

Description: Returns a flag denoting if the specified member of the group list is multicast capable.
Arguments: The index of the group member to be queried. This can be between zero and the number of members minus one.

```
HandlerBits returnHandlers(int entry)
HANDLER_CHECK(HandlerBits handlers, int bitNumber)
```

Description: Returns a value containing a series of set or cleared bits denoting the held Handlers of a group member. A specific handler in the HandlerBits entry can be checked for by using the HANDLER_CHECK macro. BitNumber must be a value between 1 and 6.
Arguments: The index of the group member to be queried. This can be between zero and the number of members minus one.

```
Source returnIPAddress(int entry)
```

Description: Returns the IP Address of a group member.
Arguments: The index of the group member to be queried. This can be between zero and the number of members minus one.

```
unsigned short int returnIPPort(int entry)
```

Description: Returns the UDP port of a group member
Arguments: The index of the group member to be queried. This can be between zero and the number of members minus one.

```
unsigned short int returnMinSizeReq(int entry)
```

Description: Returns the minimum size allowed for the group after a failure or partition occurs.
Arguments: The index of the group member to be queried. This can be between zero and the number of members minus one.

RMPMessage Class

The RMPMessage 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. A RMPMessage object is returned from the RMP returnBuffer() method. The object can then be filled in with the access methods below. Once filled in, the object can be passed as a parameter to the RMPGroup send() method or to the RMP sendMultiRPC() method. If the members below are not filled out, the default values will be used. For more details, see the RMPGroup send() method.

RMPMessage Access Methods

The following access methods are applicable to all RMPMessage objects.

```
unsigned short int returnMessageLength()
void setMessageLength(unsigned short int length)
```

Description: Sets and returns the value of the message's length. This may not exceed the current maximum data size, as returned by the returnMaximumDataSize() method below.

```
unsigned short int returnMaximumDataSize()
```

Description: Returns the maximum amount of data that can be put in a RMPMessage buffer. (NOTE: Added this)

```
QOS returnMessageQOS()
void setMessageQOS(QOS qos)
```

Description: Sets and returns the value of the message's QoS.

```
Handler returnMessageHandler()
void setMessageHandler(Handler hndlr)
```

Description: Sets and returns the value of the message's Handler.

```
char *returnDataBuffer()
```

Description: Returns a pointer to the message's data buffer, which can be filled in by the application.

RMPMessage Access Methods - From a Non Member

The below access methods are to be used only for packets that are sent to a group from a process that is not a member of that group. In addition to the fields used by normal data packet, a packet sent from a non member must also specify the destination of the packet. This involves specifying the name of the group, and optionally specifying an address for the group. The address can be either a UDP/IP Multicast {address, port, TTL} tuple, or a UDP/IP {address, port} tuple. If neither is specified, RMP will use a default policy to map the group name into a UDP/IP Multicast tuple. In the current version, this is a provided through a hash function.

```
Source returnIPMAddress()
void setIPMAddress(Source ipm)
```

Description: Sets and returns the IP Multicast address for the packet.

```
unsigned short int returnIPMPort()
void setIPMPort(unsigned short int port)
```

Description: Sets and returns the IP Multicast port for the packet.

```
unsigned char returnTTL()
void setTTL(unsigned char ttl)
```

Description: Sets and returns the IP Multicast TTL (time-to-live) for the packet.

```
Source returnIPAddress()
void setIPAddress(Source ip)
```

Description: Sets and returns the IP Address of the unicast destination for the packet. This is used as an option to using the above IP Multicast access methods.

```
unsigned short int returnIPPort()
void setIPPort(unsigned short int port)
```

Description: Sets and returns the port of the unicast destination for the packet.

```
char *returnGroupName()
void setGroupName(char *groupName)
```

Description: Sets and returns the group name for the message.