RIC Message Router -- RMR
User’s Manual

Original: 30 July 2019
Revised: 13 August 2019

Table Of Contents

RIC Message Router -- RMR
User’s Manual

OVERVIEW

The RIC Message Router (a.k.a. RMR) is a thin library which provides a latency sensitive application with the ability to send and receive messages with other RMR based applications. The library provides the following major features:

- Routing and endpoint selection is based on *message type.*
- Application is insulated from the underlying transport mechanism and/or protocols.
- Message distribution (round robin or fanout) is selectable by message type.
- Route management updates are received and processed asynchronously and without overt application involvement.

Purpose

RMR’s main purpose is to provide an application with the ability to send and receive messages to/from other peer applications with minimal effort on the application’s part. To achieve this, RMR manages manages all endpoint information, connections, and routing information necessary to establish and maintain communication. From the application’s point of view, all that is required to send a message is to allocate (via RMR) a message buffer, add the payload data, and set the message type. To receive a message, the application needs only to invoke the receive function; when a message arrives a message buffer will be returned as the function result.

Message Routing

Applications are required to place a message type into a message before sending, and may optionally add a subscription ID when appropriate. The combination of message type, and subscription ID are refered to as the *message key,* and is used to match an entry in a routing table which provides the possible endpoints expecting to receive messages with the matching key.

Round Robin Delivery

An endpoint from RMR’s perspective is an application to which RMR may establish a connection, and expect to send messages with one or more defined message keys. Each entry in the route table consists of one or more endpoint groups, called round robin groups. When a message matches a specific entry, the entry’s groups are used to select the destination of the message. A message is sent once to each group, with messages being *balanced* across the endpoints of a group via round robin selection. Care should be taken when defining multiple groups for a message type as there is extra overhead required and thus the overall message latency is somewhat reduced.

Routing Table Updates

Route table information is made available to RMR a static file (loaded once), or by updates sent from a separate route manager application. If a static table is provided, it is loaded during RMR initialisation and will remain in use until an external process connects and delivers a route table update (often referred to as a dynamic update). Dynamic updates are listened for in a separate process thread and applied automatically; the application does not need to allow for, or trigger, updates.

Latency And Throughput

While providing insulation from the underlying message transport mechanics, RMR must also do so in such a manner that message latency and throughput are not impacted. In general, the RMR induced overhead, incurred due to the process of selecting an endpoint for each message, is minimal and should not impact the overall latency or throughput of the application. This impact has been measured with test applications running on the same physical host and the average latency through RMR for a message was on the order of 0.02 milliseconds.

As an application's throughput increases, it becomes easy for the application to overrun the underlying transport mechanism (e.g. NNG), consume all available TCP transmit buffers, or otherwise find itself in a situation where a send might not immediately complete. RMR offers different *modes* which allow the application to manage these states based on the overall needs of the application. These modes are discussed in the *Configuration* section of this document.

GENERAL USE

To use, the RMR based application simply needs to initialise the RMR environment, wait for RMR to have received a routing table (become ready), and then invoke either the send or receive functions. These steps, and some behind the scenes details, are described in the following paragraphs.

Initialisation

The RMR function `rmr_init()` is used to set up the RMR environment and must be called before messages can be sent or received. One of the few parameters that the application must communicate to RMR is the port number that will be used as the listen port for new connections. The port number is passed on the initialisation function call and a TCP listen socket will be opened with this port. If the port is already in use RMR will report a failure; the application will need to reinitialise with a different port number, abort, or take some other action appropriate for the application.

In addition to creating a TCP listen port, RMR will start a process thread which will be responsible for receiving dynamic updates to the route table. This thread also causes a TCP listen port to be opened as it is expected that the process which generates route table updates will connect and send new information when needed. The route table update port is **not** supplied by the application, but is supplied via an environment variable as this value is likely determined by the mechanism which is starting and configuring the application.

The RMR Context

On successful initialisation, a void pointer, often called a *handle* by some programming languages, is returned to the application. This is a reference to the RMR control information and must be passed as the first parameter on most RMR functions calls. RMR refers to this as the context, or ctx.

Wait For Ready

An application which is only receiving messages does not need to wait for RMR to *become ready* after the call to the initialisation function. However, before the application can successfully send a message, RMR must have loaded a route table, and the application must wait for RMR to report that it has done so. The RMR function `rmr_ready()` will return the value *true* (1) when a complete route table has been loaded and can be used to determine the endpoint for a send request.

Receiving Messages

The process of receiving is fairly straight forward. The application invokes the RMR `rmr_rcv_msg()` function which will block until a message is received. The function returns a pointer to a message block which provides all of the details about the message. Specifically, the application has access to the following information either directly or indirectly:

- The payload (actual data)
- The total payload length in bytes
- The number of bytes of the payload which contain valid data
- The message type and subscription ID values
- The hostname and IP address of the source of the message (the sender)
- The transaction ID
- Tracing data (if provided)

The Message Payload

The message payload contains the *raw* data that was sent by the peer application. The format will likely depend on the message type, and is expected to be known by the application. A direct pointer to the payload is available from the message buffer (see appendix B for specific message buffer details).

Two payload related length values are also directly available: the total payload length, and the number of bytes actually filled with data. The used length is set by the caller, and may or not be an accurate value. The total payload length is determined when the buffer is created for sending, and is the maximum number of bytes that the application may modify should the buffer be used to return a response.

Message Type and Subscription ID

The message type and subscription ID are both directly available from the message buffer, and are the values which were used to by RMR in the sending application to select the endpoint. If the application resends the message, as opposed to returning the message buffer as a response, the message number and/or the subscription ID might need to be changed to avoid potential issues[1].

Sender Information

The source, or sender information, is indirectly available to the application via the `rmr_get_src()` and `rmr_get_ip()` functions. The former returns a string containing `hostname:port,` while the string `ip:port` is returned by the latter.

Transaction ID

The message buffer contains a fixed length set of bytes which applications can set to track related messages across the application concept of a transaction. RMR will use the transaction ID for matching a response message when the `rmr_call()` function is used to send a message.

[1] It is entirely possible to design a routing table, and application group, such that the same message type is is left unchanged and the message is forwarded by an application after updating the payload. This type of behaviour is often referred to as service chaining, and can be done without any "knowledge" by an application with respect to where the message goes next. Service chaining is supported by RMR in as much as it allows the message to be resent, but the actual complexities of designing and implementing service chaining lie with the route table generator process.

Trace Information

RMR supports the addition of an optional trace information to any message. The presence and size is controlled by the application, and can vary from message to message if desired. The actual contents of the trace information is determined by the application; RMR provides only the means to set, extract, and obtain a direct reference to the trace bytes. The trace data field in a message buffer is discussed in greater detail in the *Trace Data* section.

Sending Messages

Sending requires only slightly more work on the part of the application than receiving a message. The application must allocate an RMR message buffer, populate the message payload with data, set the message type and length, and optionally set the subscription ID. Information such as the source IP address, hostname, and port are automatically added to the message buffer by RMR, so there is no need for the application to worry about these.

Message Buffer Allocation

The function `rmr_msg_alloc()` allocates a *zero copy* buffer and returns a pointer to the RMR `rmr_mbuf_t` structure. The message buffer provides direct access to the payload, length, message type and subscription ID fields. The buffer must be preallocated in order to allow the underlying transport mechanism to allocate the payload space from it's internal memory pool; this eliminates multiple copies as the message is sent, and thus is more efficient.

If a message buffer has been received, and the application wishes to use the buffer to send a response, or to forward the buffer to another application, a new buffer does **not** need to be allocated. The application may set the necessary information (message type, etc.), and adjust the payload, as is necessary and then pass the message buffer to `rmr_send_msg()` or `rmr_rts_msg()` to be sent or returned to the sender.

Populating the Message Buffer

The application has direct access to several of the message buffer fields, and should set them appropriately.

`len`	This is the number of bytes that the application placed into the payload. Setting length to 0 is allowed, and length may be less than the allocated payload size.
`mtype`	The message type that RMR will use to determine the endpoint used as the target of the send.
`sub_id`	The subscription ID if the message is to be routed based on the combination of message type and subscription ID. If no subscription ID is valid for the message, the application should set the field with the RMR constant `RMR_VOID_SUBID.`
`payload`	The application should obtain the reference (pointer) to the payload from the message buffer and place any data into the payload. The application is responsible for ensuring that the maximum payload size is not exceeded. The application may obtain the maximum size via the `rmr_payload_size()` function.
`trace data`	Optionally, the application may add trace information to the message buffer.

Sending a Message Buffer

Once the application has populated the necessary bits of a message, it may be sent by passing the buffer to the `rmr_send_msg()` function. This function will select an endpoint to receive the message, based on message type and subscription ID, and will pass the message to the underlying transport mechanism for actual transmission on the connection. (Depending on the underlying transport mechanism, the actual connection to the endpoint may happen at the time of the first message sent to the endpoint, and thus the latency of the first send might be longer than expected.)

On success, the send function will return a reference to a message buffer; the status within that message buffer will indicate what the message buffer contains. When the status is `RMR_OK` the reference is to a **new** message buffer for the application to use for the next send; the payload size is the same as the payload size allocated for the message that was just sent. This is a convenience as it eliminates the need for the application to call the message allocation function at some point in the future, and assumes the application will send many messages which will require the same payload dimensions.

If the message contains any status other than `RMR_OK,` then the message could **not** be sent, and the reference is to the unsent message buffer. The value of the status will indicate whether the nature of the failure was transient (`RMR_ERR_RETRY`) or not. Transient failures are likely to be successful if the application attempts to send the message at a later time. Unfortunately, it is impossible for RMR to know the exact transient failure (e.g. connection being established, or TCP buffer shortage), and thus it is not possible to communicate how long the application should wait before attempting to resend, if the application wishes to resend the message. (More discussion with respect to message retries can be found in the *Handling Failures* section.)

ADVANCED USAGE

Several forms of usage fall into a more advanced category and are described in the following sections. These include blocking call, return to sender and wormhole functions.

The Call Function

The RMR function `rmr_call()` will send a message in the exact same manner as the `rmr_send_msg()` function, with the endpoint selection based on the message key. Unlike the send function, `rmr_call()` will block and wait for a response from the application that is selected to receive the message. The matching message is determined by the transaction ID which the application must place into the message buffer prior to invoking `rmr_call()`. Similarly, the responding application must ensure that the same transaction ID is placed into the message buffer before returning its response.

The return from the call is a message buffer with the response message; there is no difference between a message buffer returned by the receive function and one returned by the &fucnt function. If a response is not received in a reasonable amount of time, a nil message buffer is returned to the calling application.

Returning a Response

Because of the nature of RMR's routing policies, it is generally not possible for an application to control exactly which endpoint is sent a message. There are cases, such as responding to a message delivered via `rmr_call()` that the application must send a message and guarantee that RMR routes it to an exact destination. To enable this, RMR provides the `rmr_rts_msg(),` return to sender, function. Upon receipt of any message, an application may alter the payload, and if necessary the message type and subscription ID, an pass the altered message buffer to the `rmr_rts_msg()` function to return the altered message to the application which sent it. When this function is used, RMR will examine the message buffer for the source information and use that to select the connection on which to write the response.

Multi-threaded Calls

The basic call mechanism described above is **not** thread safe, as it is not possible to guarantee that a response message is delivered to the correct thread. The RMR function `rmr_mt_call()` accepts an additional parameter which identifies the calling thread in order to ensure that the response is delivered properly. In addition, the application must specifically initialise the multi-threaded call environment by passing the `RMRFL_MTCALL` flag as an option to the `rmr_init()` function[2].

One advantage of the multi-threaded call capability in RMR is the fact that only the calling thread is blocked. Messages received which are not responses to the call are continued to be delivered via normal `rmr_rcv_msg()` calls.

While the process is blocked waiting for the response, it is entirely possible that asynchronous, nonmatching, messages will arrive. When this happens, RMR will queues the messages and return them to the application over the next calls to `rmr_rcv_msg()`.

Wormholes

As was mentioned earlier, the design of RMR is to eliminate the need for an application to know a specific endpoint, even when a response message is being sent. In some rare cases it may be necessary for an application to establish a direct connection to an RMR based application rather than relying on message type and subscription ID based routing. The *wormhole* functions provide an application with the ability to create a direct connection and then to send and receive messages across the connection. The following are the RMR functions which provide wormhole communications:

rmr_wh_open	Open a connection to an endpoint. Name or IP address and port of the endpoint is supplied. Returns a wormhole ID that the application must use when sending a direct message.
rmr_wh_send_msg	Sends an RMR message buffer to the connected application. The message type and subscription ID may be set in the message, but RMR will ignore both.
rmr_wh_close	Closes the direct connection.

HANDLING FAILURES

The vast majority of states reported by RMR are fatal; if encountered during setup or initialisation, then it is unlikely that any message oriented processing should continue, and when encountered on a message operation continued operation on that message should be abandoned. Specifically with regard to message sending, it is very likely that the underlying transport mechanism will report a *soft,* or transient, failure which might be successful if the operation is retried at a later point in time. The paragraphs below discuss the methods that an application might deal with these soft failures.

Failure Notification

When a soft failure is reported, the returned message buffer returned by the RMR function will be `RMR_ERR_RETRY`. These types of failures can occur for various reasons; one of two reasons is typically the underlying cause:

- The session to the targeted recipient (endpoint) is not connected.
- The transport mechanism buffer pool is full and cannot accept another buffer.

[2] There is additional overhead to support multi-threaded call as a special listener thread must be used in order to deliver responses to the proper application thread.

Unfortunately, it is not possible for RMR to determine which of these two cases is occurring, and equally as unfortunate the time to resolve each is different. The first, no connection, may require up to a second before a message can be accepted, while a rejection because of buffer shortage is likely to resolve in less than a millisecond.

Application Response

The action which an application takes when a soft failure is reported ultimately depends on the nature of the application with respect to factors such as tolerance to extended message latency, dropped messages, and over all message rate.

RMR Retry Modes

In an effort to reduce the workload of an application developer, RMR has a default retry policy such that RMR will attempt to retransmit a message up to 1000 times when a soft failure is reported. These retries generally take less than 1 millisecond (if all 1000 are attempted) and in most cases eliminates nearly all reported soft failures to the application. When using this mode, it might allow the application to simply treat all bad return values from a send attempt as permanent failures.

If an application is so sensitive to any delay in RMR, or the underlying transport mechanism, it is possible to set RMR to return a failure immediately on any kind of error (permanent failures are always reported without retry). In this mode, RMR will still set the state in the message buffer to `RMR_ERR_RETRY,` but will **not** make any attempts to resend the message. This zero-retry policy is enabled by invoking the `rmr_set_stimeout()` with a value of 0; this can be done once immediately after `rmr_init()` is invoked.

Regardless of the retry mode which the application sets, it will ultimately be up to the application to handle failures by queuing the message internally for resend, retrying immediately, or dropping the send attempt all together. As stated before, only the application can determine how to best handle send failures.

Other Failures

RMR will return the state of processing for message based operations (send/receive) as the status in the message buffer. For non-message operations, state is returned to the caller as the integer return value for all functions which are not expected to return a pointer (e.g. `rmr_init()`.) The following are the RMR state constants and a brief description of their meaning.

`RMR_OK`	state is good; operation finished successfully
`RMR_ERR_BADARG`	argument passed to function was unusable
`RMR_ERR_NOENDPT`	send/call could not find an endpoint based on msg type
`RMR_ERR_EMPTY`	msg received had no payload; attempt to send an empty message
`RMR_ERR_NOHDR`	message didn’t contain a valid header
`RMR_ERR_SENDFAILED`	send failed; errno may contain the transport provider reason
`RMR_ERR_CALLFAILED`	unable to send the message for a call function; errno may contain the transport provider reason
`RMR_ERR_NOWHOPEN`	no wormholes are open
`RMR_ERR_WHID`	the wormhole id provided was invalid
`RMR_ERR_OVERFLOW`	operation would have busted through a buffer/field size
`RMR_ERR_RETRY`	request (send/call/rts) failed, but caller should retry (EAGAIN for wrappers)
`RMR_ERR_RCVFAILED`	receive failed (hard error)
`RMR_ERR_TIMEOUT`	response message not received in a reasonable amount of time

`RMR_ERR_UNSET`	the message hasn't been populated with a transport buffer
`RMR_ERR_TRUNC`	length in the received buffer is longer than the size of the allocated payload, received message likely truncated (length set by sender could be wrong, but we can't know that)
`RMR_ERR_INITFAILED`	initialisation of something (probably message) failed
`RMR_ERR_NOTSUPP`	the request is not supported, or RMr was not initialised for the request

Depending on the underlying transport mechanism, and the nature of the call that RMR attempted, the system `errno` value might reflect additional detail about the failure. Applications should **not** rely on errno as some transport mechanisms do not set it with any consistency.

CONFIGURATION AND CONTROL

With the assumption that most RMR based applications will be executed in a containerised environment, there are some underlying mechanics which the developer may need to know in order to properly provide a configuration specification to the container management system. The following paragraphs briefly discuss these.

TCP Ports

RMR requires two (2) TCP listen ports: one for general application to application communications and one for route table updates. The general communication port is specified by the application at the time RMR is initialised. The port used to listen for route table updates is likely to be a constant port shared by all applications provided they are running in separate containers. To that end, the port number defaults to 4561, but can be configured with an environment variable (see later paragraph in this section).

Host Names

RMR is typically host name agnostic. Route table entries may contain endpoints defined either by host name or IP address. In the container world the concept of a *service name* might exist, and likely is different than a host name. RMR's only requirement with respect to host names is that a name used on a route table entry must be resolvable via the `gethostbyname` system call.

Environment Variables

Several environment variables are recognised by RMR which, in general, are used to define interfaces and listen ports (e.g. the route table update listen port), or debugging information. Generally this information is system controlled and thus RMR expects this information to be defined in the environment rather than provided by the application. The following is a list of the environment variables which RMR recognises:

`RMR_BIND_IF`	The interface to bind to listen ports to. If not defined 0.0.0.0 (all interfaces) is assumed.
`RMR_RTG_SVC`	The port RMR will listen on for route manager connections. If not defined 4561 is used.
`RMR_SEED_RT`	Where RMR expects to find the name of the seed (static) route table. If not defined no static table is read.
`RMR_RTG_ISRAW`	If the value set to 0, RMR expects the route table manager messages to be messages with and RMR header. If this is not defined messages are assumed to be "raw" (without an RMR header.
`RMR_VCTL_FILE`	Provides a file which is used to set the verbose level of the route table collection thread. The first line of the file is read and expected to contain an integer value to set the verbose level. The value may be changed at any time and the route table thread will adjust accordingly.

`RMR_SRC_NAMEONLY`

If the value of this variable is greater than 0, RMR will not permit the IP address to be sent as the message source. Only the host name will be sent as the source in the message header.

Logging

RMR does **not** use any logging libraries; any error or warning messages are written to standard error. RMR messages are written with one of three prefix strings:

`[CRI]`	The event is of a critical nature and it is unlikely that RMR will continue to operate correctly if at all. It is almost certain that immediate action will be needed to resolve the issue.
`[ERR]`	The event is not expected and RMR is not able to handle it. There is a small chance that continued operation will be negatively impacted. Eventual action to diagnose and correct the issue will be necessary.
`[WRN]`	The event was not expected by RMR, but can be worked round. Normal operation will continue, but it is recommended that the cause of the problem be investigated.

APPENDIX A -- QUICK REFERENCE

The prototype for each of the externally available functions which comprise the RMR API is listed in alphabetical order below. For each prototype a brief description of the function is given. The developer is encouraged to install the RMR development package which contains the manual pages. The manual pages completely describe each function in a level of detail consistent with UNIX man pages.

Context Specific Functions

These functions require that the RMR context (provided as the result of an `rmr_init()` call, be passed as the first argument (vctx).

`rmr_mbuf_t* rmr_alloc_msg( void* vctx, int size );`

This function allocates a *zero copy* message buffer. The payload portion is allocated from the underlying transport space such that on send the buffer does not require a second copy. The size parameter is the size of the payload portion of the buffer; if the recipient of the message is expected to send a response, this should be large enough to accomodate the response which will enable the remote application to reuse the message for the response and avoid a costly reallocation.

`rmr_mbuf_t* rmr_call( void* vctx, rmr_mbuf_t* msg );`

The call function accepts a message, selects an endpoint to receive the message and sends the message. RMR will block the return to the application until a matching response is received, or a timeout period is reached. The transaction ID in the outbound message is used to match a response, so it is imperative that the application making the response reuse the received message, or copy the transaction ID to the new message before sending. Messages arriving which do not match the expected response are queued and will be delivered to the application on subsequent calls to `rmr_rcv_msg().`

`void rmr_close( void* vctx );`

This function terminates all sessions with the underlying transport mechanism. Buffers pending may or may not be flushed (depending on the underlying mechanism), thus it is recommended that before using this function the application pause for a second or two to ensure that the pending transmissions have completed.

`int rmr_get_rcvfd( void* vctx );`

When the underlying transport mechanism supports this, a file descriptor suitable for use with the `select, poll` and `epoll` system calls is returned. The file descriptor may not be used for read or write operations; doing so will have unpredictable results.

`void* rmr_init( char* proto_port, int max_msg_size, int flags );`

This function must be used before all other RMR functions to initialise the environment. The `max_msg_size` parameter defines the maximum receive buffer size which will be used if required by the underlying transport mechanism. The value is also used as the default payload size if a zero length is given to `rmr_alloc_msg().`

`rmr_mbuf_t* rmr_mt_call( void* vctx, rmr_mbuf_t* mbuf, int call_id, int max_wait );`

Similar to the `rmr_call()` function, the message is sent to an endpoint and a response message is waited for. This call is thread safe, and while the thread that invokes this function blocks on the response, it is possible for other application threads to continue to receive messages via the `rmr_rcv_msg()` function.

In order to use the multi-threaded call functions, the option to enable threaded receive support must be set on the call to `rmr_init().`

`rmr_mbuf_t* rmr_mtosend_msg( void* vctx, rmr_mbuf_t* msg, int max_to );`

This function sends the provided message allowing RMR to retry soft failures for approximately &ccw milliseconds. When a value of zero (0) is given for the maximum timeout, RMR will not attempt any retires and will return the state after the first attempt. It is unlikely that a user application will use this function as it is possible (and recommended) to set the max timeout via the specific, one time, function call, and then to allow that value to be used as the default when `rmr_send_msg()` is invoked.

`rmr_mbuf_t* rmr_mt_rcv( void* vctx, rmr_mbuf_t* mbuf, int max_wait );`

This function waits for a message to arrive and returns a message buffer with the received message. The function will timeout after `max_wait` milliseconds (approximately) if no message is received.

`rmr_mbuf_t* rmr_send_msg( void* vctx, rmr_mbuf_t* msg );`

This function accepts a message, selects a destination endpoint using the message type and subscription ID, and then attempts to send the message to the destination. The function returns a message buffer to the caller with the state set to indicate the state of the send operation. On success, the message buffer is a new buffer that the application can "fill in," and send without the need to overtly allocate a new buffer. On failure, the message buffer is the buffer that the application attempted to send.

`int rmr_init_trace( void* vctx, int size );`

This function is used to set the default trace data size. The size defaults to zero until this function is called; after the application sets a non-zero value messages created with the `rmr_alloc_msg()` function will be allocated with trace data set to the size provided.

`rmr_mbuf_t* rmr_rcv_msg( void* vctx, rmr_mbuf_t* old_msg );`

The `rmr_rcv_msg()` function is used to block and wait for a message to arrive. When a message is received a pointer to an RMR message buffer structure is returned to the caller. The `old_msg` parameter allows the application to to pass a message buffer for reuse. If the application does not have an old buffer, a nil pointer is given and the function will allocate a new buffer.

```
rmr_mbuf_t* rmr_rcv_specific( void* uctx, rmr_mbuf_t* msg, char* expect,
     int allow2queue );
```

This function blocks until a message with a specific transaction ID is received. If the `allowd2queue` parameter is set to 1, messagess which do not match the ID are queued and returned to the application on subsequent calls to `rmr_rcv_msg()`.

`int rmr_ready( void* vctx );`

This function is used to test whether RMR is capable of sending messages. In other words once this function returns true (!0) RMR has received a route table (either from a static file or from a route manager process, and can map message types to endpoints. If the application attempts to send a message before this function returns true, the sends will fail. Applications which are only message receivers do not need to use this function.

`rmr_mbuf_t*  rmr_rts_msg( void* vctx, rmr_mbuf_t* msg );`

The `rmr_rts_msg()` function allows the application to send a response message to the endpoint from which the message originated. This requires that the application use the same message buffer that was received for the response as it contains the sender information that is needed for this function to be successful. If the message cannot be sent, a pointer to the message buffer is returned and the status in the message buffer is set to indicate the reason. On success, a nil pointer will be returned.

`int rmr_set_rtimeout( void* vctx, int time );`

This function provides the ability to return from a receive operation after a timeout threshold is reached before a message is received, and is intended only to support the underlying Nanomsg transport mechanism (support for Nanomsg is deprecated). The `rmr_torcv_msg()` function should be used if timed receives are required.

For transport mechanisms which support a receive timeout, this function allows the application to set a default timeout for receive operations. If a call to `rmr_rcv_msg()` does not complete before *time* milliseconds has elapsed, the receive function will return a nil message buffer. This may not be supported by the underlying transport mechanism, and if it is not the return from this function will be -1.

`int rmr_set_stimeout( void* vctx, int rounds );`

This function allows the application to set a maximum number of retry *rounds* that RMR will attempt when send operations report a transient (retry) failure. Each *round* of retries requires approximately 1 millisecond, and setting the number of rounds to zero (0) causes RMR to report the transient failure to the application without any retry attempts. If the user application does not invoke this function, the default is one (1) round of retries.

`rmr_mbuf_t* rmr_torcv_msg( void* vctx, rmr_mbuf_t* old_msg, int ms_to );`

This function because identically to the `rmr_rcv_msg()` function, and allows the application to set a specific timeout value for the receive operation. If a message is not received before the timeout period expires (ms_to milliseconds), a message buffer is returned with the state set to `RMR_ERR_TIMEOUT.`

```
rmr_mbuf_t*  rmr_tralloc_msg( void* context, int msize, int trsize,
     unsigned const char* data );
```

Similar to the `rmr_alloc_msg()` this function allocates a message buffer, and adds the referenced trace data to the buffer. The new message buffer is returned.

`void rmr_wh_close( void* vctx, int whid );`

This function closes an existing wormhole connection.

`rmr_whid_t rmr_wh_open( void* vctx, char const* target );`

This function allows the application to create a *wormhole,* direct, connection to another application. The peer application must also be using RMR (messages sent on a wormhole connection are RMR messages). The target may be a hostname:port or IP-address:port combination. Upon successful completion, the *wormhole ID* is returned; this ID must be passed on all subsequent calls to wormhole functions for this connection.

`rmr_mbuf_t* rmr_wh_send_msg( void* vctx, rmr_whid_t whid, rmr_mbuf_t* msg );`

Once a wormhole has been established to a peer application, this function allows the application to send a message to the peer. All semantics of normal RMR sends (retries, etc.) are observed. The application may opt not to supply the message type or subscription ID in the message as neither are used by RMR; they may be required by the peer application depending on the application level protocol(s) in use.

Message Buffer Functions

The message buffer functions operate directly on a message buffer, and as such do not require that RMR context as a parameter.

`int rmr_bytes2meid( rmr_mbuf_t* mbuf, unsigned char const* src, int len );`

Copy the bytes referenced by *src* to the *meid* field in the RMR message header. Up to *len* bytes are copied, though the maximum length of the field as governed by `RMR_MAX_MEID` is enforced.

`void rmr_bytes2payload( rmr_mbuf_t* mbuf, unsigned char const* src, int len );`

This function copies *len* bytes from *src* into the message buffer payload. This function is primarily provided to support wrappers which don't directly support C pointers.

`int rmr_bytes2xact( rmr_mbuf_t* mbuf, unsigned char const* src, int len );`

This function copies *len* bytes of data from *src* to the transaction ID field in the message buffer. The number of bytes provided will be limited to a maximum of `RMR_MAX_XID.`

`void rmr_free_msg( rmr_mbuf_t* mbuf );`

This function should be used by the application to release the storage used by a message buffer.

`unsigned char*  rmr_get_meid( rmr_mbuf_t* mbuf, unsigned char* dest );`

The bytes from the `meid` field of the message buffer are copied to the *dest* buffer provided by the application. The full field of `RMR_MAX_MEID` bytes are copied; the caller must ensure that *dest* is large enough.

`unsigned char*  rmr_get_src( rmr_mbuf_t* mbuf, unsigned char* dest );`

The source of a message is copied to the *dest* buffer provided by the caller. This is generally the hostname and port, separated by a colon, of the application which sent the message, and is a zero terminated string. Up to `RMR_MAX_SRC` bytes will be copied, so the caller must ensure that *dest* is at least this large.

`unsigned char* rmr_get_srcip( rmr_mbuf_t* msg, unsigned char* dest );`

This function copies the source IP address and port, separated by a colon, to the *dest* buffer provided by the caller. This is the address of the application which sent the message. Up to `RMR_MAX_SRC` bytes will be copied, so the caller must ensure that *dest* is at least this large.

`int rmr_get_trlen( rmr_mbuf_t* msg );`

This function can be used to determine the size of the trace information in the message buffer. If no trace data is present, then 0 is returned.

`int rmr_get_trace( rmr_mbuf_t* msg, unsigned char* dest, int size );`

The bytes from the trace data, up to &tial bytes, is copied from the message buffer to the *dest* buffer provided by the caller. The return value is the number of bytes actually copied.

`int rmr_payload_size( rmr_mbuf_t* msg );`

This function returns the number of bytes in the message buffer's payload that are available for the application to use.

`rmr_mbuf_t* rmr_realloc_msg( rmr_mbuf_t* mbuf, int new_tr_size );`

This function allows the application to reallocate a message buffer with a different trace data size. The contents of the message buffer supplied are copied to the new buffer, and a reference to the new buffer is returned.

`int rmr_set_trace( rmr_mbuf_t* msg, unsigned const char* data, int size );`

The *size* bytes, up to the size of the trace data in the message buffer, at *data* are copied to the trace portion of the message buffer. The return value is the actual number of bytes copied which could be less than the number requested.

`int rmr_str2meid( rmr_mbuf_t* mbuf, unsigned char const* str );`

Accepts a pointer to a zero terminated string an copies it to the `meid` field in the message header. Up to `RMR_MAX_MEID` bytes are copied (including the final 0), and the number copied is returned.

`void rmr_str2payload( rmr_mbuf_t* mbuf, unsigned char const* str );`

Accepts a pointer to a zero terminated string, and copies the string to the payload portion of the message buffer. If the string is longer than the allocated payload, the string will be truncated and will **not** be terminated with a zero byte.

`int rmr_str2xact( rmr_mbuf_t* mbuf, unsigned char const* str );`

Accepts a pointer to a zero terminated string and copies the string to the transaction ID portion of the message buffer. If the string is longer than `RMR_MAX_XID`, the string will be truncated and will **not** be zero terminated.

`void* rmr_trace_ref( rmr_mbuf_t* msg, int* sizeptr );`

This function returns a pointer to the trace information in the message buffer. The intent is that the application will treat this as a read/only field and will not write trace data into the message buffer. The length of data available should be determined by calling `rmr_get_trlen()`.

APPENDIX B -- MESSAGE BUFFER DETAILS

The RMR message buffer is a C structure which is exposed in the `rmr.h` header file. It is used to manage a message received from a peer endpoint, or a message that is being sent to a peer. Fields include payload length, amount of payload actually used, status, and a reference to the payload. There are also fields which the application should ignore, and could be hidden in the header file, but we chose not to. These fields include a reference to the RMR header information, and to the underlying transport mechanism message struct which may or may not be the same as the RMR header reference.

The Structure

The following is the C structure. Readers are cautioned to examine the header file directly; the information here may be out of date (old document in some cache), and thus it may be incorrect.

```
typedef struct {
    int    state;            // state of processing
    int    mtype;            // message type
    int    len;              // length of data in the payload (send or received)
    unsigned char* payload;  // transported data
    unsigned char* xaction;  // pointer to fixed length transaction id bytes
    int    sub_id;           // subscription id
    int    tp_state;         // transport state (errno)

                             // these things are off limits to the user application
    void*    tp_buf;         // underlying transport allocated pointer (e.g. nng message)
    void*    header;         // internal message header (whole buffer: header+payload)
    unsigned char* id;       // if we need an ID in the message separate from the xaction id
    int      flags;          // various MFL_ (private) flags as needed
    int      alloc_len;      // the length of the allocated space (hdr+payload)
} rmr_mbuf_t;
```

State vs Transport State

The state field reflects the state at the time the message buffer is returned to the calling applicaiton. For a send operation, if the state is not `RMR_OK` then the message buffer references the payload that could not be sent, and when the state is `RMR_OK` the buffer references a *fresh* payload that the application may fill in.

When the state is not `RMR_OK,` C programmes may examine the global `errno` value which RMR will have left set, if it was set, by the underlying transport mechanism. In some cases, wrapper modules are not able to directly access the C-library `errno` value, and to assist with possible transport error details, the send and receive operations populate `tp_state` with the value of `errno.`

Regardless of whether the application makes use of the `tp_state,` or the `errno` value, it should be noted that the underlying transport mechanism may not actually update the errno value; in other words: it might not be accurate. In addition, RMR populates the `tp_state` value in the message buffer **only** when the state is not `RMR_OK.`

Field References

The transaction field was exposed in the first version of RMR, and in hindsight this shouldn't have been done. Rather than break any existing code the reference was left, but additional fields such as trace data, were not directly exposed to the application. The application developer is strongly encouraged to use the functions which get and set the transaction ID rather than using the pointer directly; any data overruns will not be detected if the reference is used directly.

In contrast, the payload reference should be used directly by the application in the interest of speed and ease of programming. The same care to prevent writing more bytes to the payload buffer than it can hold must be taken by the application. By the nature of the allocation of the payload in transport space, RMR is unable to add guard bytes and/or test for data overrun.

Actual Transmission

When RMR sends the application's message, the message buffer is **not** transmitted. The transport buffer (tp_buf) which contains the RMR header and application payload is the only set of bytes which are transmitted. While it may seem to the caller like the function `rmr_send_msg()` is returning a new message buffer, the same struct is reused and only a new transport buffer is allocated. The intent is to keep the alloc/free cycles to a minimum.

APPENDIX C -- GLOSSARY

Many terms in networking can be interpreted with multiple meanings, and several terms used in this document are RMR specific. The following definitions are the meanings of terms used within this document and should help the reader to understand the intent of meaning.

application	A programme which uses RMR to send and/or receive messages to/from another RMR based application.
Critical error	An error that RMR has encountered which will prevent further successful processing by RMR. Critical errors usually indicate that the application should abort.
Endpoint	An RMR based application that is defined as being capable of receiving one or more types of messages (as defined by a *message key.*)
Environment variable	A key/value pair which is set externally to the application, but which is available to the application (and referenced libraries) through the `getenv` system call. Environment variables are the main method of communicating information such as port numbers to RMR.
Error	An abnormal condition that RMR has encountered, but will not affect the overall processing by RMR, but may impact certain aspects such as the ability to communicate with a specific endpoint. Errors generally indicate that something, usually external to RMR, must be addressed.
Host name	The name of the host as returned by the `gethostbyname` system call. In a containerised environment this might be the container or service name depending on how the container is started. From RMR's point of view, a host name can be used to resolve an *endpoint* definition in a *route table.*
IP	Internet protocol. A low level transmission protocol which governs the transmission of datagrams across network boundaries.
Listen socket	A *TCP* socket used to await incoming connection requests. Listen sockets are defined by an interface and port number combination where the port number is unique for the interface.
Message	A series of bytes transmitted from the application to another RMR based application. A message is comprised of RMR specific data (a header), and application data (a payload).
Message buffer	A data structure used to describe a message which is to be sent or has been received. The message buffer includes the payload length, message type, message source, and other information.
Messgae type	A signed integer (0-32000) which identifies the type of message being transmitted, and is one of the two components of a *routing key.* See *Subscription ID.*
Payload	The portion of a message which holds the user data to be transmitted to the remote *endpoint.* The payload contents are completely application defined.
RMR context	A set of information which defines the current state of the underlying transport connections that RMR is managing. The application will be give a context reference (pointer) that is supplied to most RMR functions as the first parameter.
Round robin	The method of selecting an *endpoint* from a list such that all *endpoints* are selected before starting at the head of the list.
Route table	A series of "rules" which define the possible *endpoints* for each *message key.*

Route table manager	An application responsible for building a *route table* and then distributing it to all applicable RMR based applications.
Routing	The process of selecting an *endpoint* which will be the recipient of a message.
Routing key	A combination of *message type* and *subscription ID* which RMR uses to select the destination *endpoint* when sending a message.
Source	The sender of a message.
Subscription ID	A signed integer value (0-32000) which identifies the subscription characteristic of a message. It is used in conjunction with the *message type* to determine the *routing key.*
Target	The *endpoint* selected to receive a message.
TCP	Transmission Control Protocol. A connection based internet protocol which provides for lossless packet transportation, usually over IP.
Thread	Also called a *process thread, or pthread.* This is a lightweight process which executes in concurrently with the application and shares the same address space. RMR uses threads to manage asynchronous functions such as route table updates. &Term An optional portion of the message buffer that the application may populate with data that allows for tracing the progress of the transaction or application activity across components. RMR makes no use of this data.
Transaction ID	A fixed number of bytes in the *message buffer* which the application may populate with information related to the transaction. RMR makes use of the transaction ID for matching response messages with the &c function is used to send a message.
Transient failure	An error state that is believed to be short lived and that the operation, if retried by the application, might be successful. C programmers will recognise this as `EAGAIN.`
Warning	A warning occurs when RMR has encountered something that it believes isn’t correct, but has a defined work round.
Wormhole	A direct connection managed by RMR between the user application and a remote, RMR based, application.

APPENDIX D -- CODE EXAMPLES

The following snippet of code illustrate some of the basic operation of the RMR library. Please refer to the examples and test directories in the RMR repository for complete RMR based programmes.

Sender Sample

The following code segment shows how a message buffer can be allocated, populated, and sent. The snippet also illustrates how the result from the `rmr_send_msg()` function is used to send the next message. It does not illustrate error and/or retry handling.

```
    mrc = rmr_init( listen_port, MAX_BUF_SZ, RMRFL_NOFLAGS );
    rmr_set_stimeout( mrc, rmr_retries );

    while( ! rmr_ready( mrc ) ) {
        sleep( 1 );
    }

    sbuf = rmr_alloc_msg( mrc, 256 );   // 1st send buffer

    while( TRUE ) {
        sbuf->len = gen_status( (status_msg *) sbuf->payload );
        sbuf->mtype = STATUS_MSG;
        sbuf->sub_id = RMR_VOID_SUBID;     // subscription not used
        sbuf = rmr_send_msg( mrc, sbuf );

        sleep( delay_sec );
    }

    rmr_close( mrc );
```

Receiver Sample

The receiver code is even more simple than the sender code as it does not need to wait for a route table to arrive (only senders need to do that), nor does it need to allocate an initial buffer. The example assumes that the sender is transmitting a zero terminated string as the payload.

```
    rmr_mbuf_t* rbuf = NULL;
    void* mrc = rmr_init( listen_port, MAX_BUF_SZ, RMRFL_NOFLAGS );

    while( TRUE ) {
        rbuf = rmr_rcv_msg( mrc, rbuf );    // reuse buffer on all but first loop
        if( rbuf == NULL || rbuf->state != RMR_OK ) {
            break;
        }

        fprintf( stdout, "mtype=%d sid=%d pay=%s\n",
            rbuf->mtype, rbuf->sub_id, rbuf->payload );
        sleep( delay_sec );
    }

    fprintf( stderr, "receive error\n" );
    rmr_close( mrc );
```

Receive and Send Sample

The following code snippet receives messages and responds to the sender if the message type is odd. The code illustrates how the received message may be used to return a message to the source. Variable type definitions are omitted for clarity and should be obvious.

It should also be noted that things like the message type which id returned to the sender (99) is a random value that these applications would have agreed on in advance and is **not** an RMR definition.

```
mrc = rmr_init( listen_port, MAX_BUF_SZ, RMRFL_NOFLAGS );
rmr_set_stimeout( mrc, 1 );        // allow RMR to retry failed sends for ~1ms

while( ! rmr_ready( mrc ) ) {        // we send, therefore we need a route table
    sleep( 1 );
}

mbuf = NULL;                        // ensure our buffer pointer is nil for 1st call

while( TRUE ) {
    mbuf = rmr_rcv_msg( mrc, mbuf );        // wait for message

    if( mbuf == NULL || mbuf->state != RMR_OK ) {
        break;
    }

    if( mbuf->mtype % 2 ) {                // respond to odd message types
        plen = rmr_payload_size( mbuf );        // max size

                                                // reset necessary fields in msg
        mbuf->mtype = 99;                       // response type
        mbuf->sub_id = RMR_VOID_SUBID;          // we turn subid off
        mbuf->len = snprintf( mbuf->payload, plen, "pong: %s", get_info() );

        mbuf = rmr_rts_msg( mrc, mbuf );        // return to sender
        if( mbuf == NULL || mbuf->state != RMR_OK ) {
            fprintf( stderr, "return to sender failed\n" );
        }
    }
}

fprintf( stderr, "abort: receive failure\n" );
rmr_close( mrc );
```

INDEX