RMR Frequently Asked Questions

(We'll also provide the answers!)

FYI: most recently added questions and answers are at the top.

My xAPP cannot send message type N to another xAPP (RMR_ERR_NOENDPT)

There are several potential causes for this issue:

- The route table issue for the message type N is not defined
- The subscription ID used on the message isn't defined in the route table for the message type
- Network issues between the two end point hosts (including docker/kubentes port definition mistakes)
- The application on the "other side" is not running
- One application is using NNG as a transport while the other is using SI95

Several things which can be done to investigate:

- Use `netstat` or `ss` in the target container (container that should receive the message) and ensure that the port as an active listen (grep for LISTEN)
- Attempt to use telnet to connect to the listen port from the source container (container sending the message)
- Use `tcpdump` to examine the flow between the endpoints

If the problem is caused by a mismatch of libraries, the symptom when NNG is the sending side will be a connection followed by a nearly immediate disconnection. `Nestat` and/or `ss` will show the connection in a TIME_WAIT state (indicating that it connected, but has closed).

Can an xAPP using librmr_nng (NNG) communicate with an xAPP using librmr_si (SI95)?

No. The "handshake" protocol used by the NNG transport library is not the same as that used by SI95. There was some consideration regarding the support of the NNG protocol inside of SI95, but that would have resulted in both increased maintenance of SI95, and potential compatibility issues should NNG change in the future.

What environment variables are used and/or expected by RMR?

There are several RMR environment variables which cause the behaviour of RMR to vary from the default. The following table lists the variables, their effects, and where applicable possible values.

Variable	Use/Effect	Value Limits /Meanings
RMR_SEED_RT	Provides RMR with an initial Route Table. The table is read once during initalisation and is overlaid with the first table delivered by the Route Manager	Any valid file name readable by the application.
RMR_VCTL_FILE	Provides the filename of a verbosity level control file used by the route collection thread. The first line of the file is read and if non-zero debugging information related to the Route Manager updates will be written to the standard error device. The value in the file may change during the lifetime of the process, but in order to change it dynamically the variable and file must exist at the time rmr_init() is invoked by the application (thus it is possible for the application to set the name and value before initialising RMR. This is meant as a debugging tool only and no effort to make this any more "pretty" has been made.	Values in the file that are recognised: 0 == off 1== some 2 == more.
RMR_SRC_NAME ONLY	RMR will send only the hostname:port as the source of each message and will never send the IP-address:port. The source information is what a receiving process will use when the rmr_rts_msg( ) (return to sender) is used.	values may be 0 (off) or 1(on).
RMR_SRC_ID	This provides the string which RMR will place into all outbound messages as the source of the message. This should be hostname:port or similar. It might be needed when a container hostname cannot be used to route messages back to the container (e.g. when host networking is being used). If not supplied, the host:port for the container is used.	

RMR_LOG_VLEVEL	This controls the verbosity level of RMR. Normally, RMR does not write any messages to standard error except severe and fatal errors. Most of RMR debugging message generation code is disabled during compile so as not to bog the process down with "if(err_level)" checks which eventually can consume a significant number of CPU cycles, thus setting a level of 5 (debug) will likely not generate the amount of messages that might be expected. If not set a value of 2 (error and critical) is assumed.	Values are in a range of 0 through 5; 0 == off 1 == critical only 2 == +errors 3 == +warnings 4 == + info 5 == + debug
RMR_RTG_SVC	This variable controls how RMR expects to receive update messages from the Route Manager. This variable's value will contain either a host:port or just a port. When it is a host:port RMR will attempt to connect to the Route Manager using the address. When only a port is given, RMR will assume that it will listen on the port for Route Manager connections.	If not defined, then the servicename:port "routemgr:4561" is used.
RMR_CTL_PORT	This is the control port that RMR will use to receive control messages and is the port sent to Route Manager as the return to sender port when the RMR_RTG_SVC contains a host:port value. If not supplied the default is 4561. (This is a new variable with the 3.x.x versions of RMR.)	Any valid port for the application, likely assigned by the container manager.

When the xAPP receives an RMR response for a subscription request, what should the subscription ID in that RMR message be?

When Subscription Manager is used in the RIC, it will have ability to merge subscriptions and thus when an xApp receives a response to its subscription request, the sub_id will be the sub_id as assigned by Subscription Manager. However, when no Subscription Manager is used, the sub_id of the response should match the "RIC Request Sequence Number" of the original subscription request message.

What should the xAPP do with the sub_id (i.e how to use it in the future ?)

The RIC INDICATION messages related to the subscription will carry the same sub_id as the subscription response. If the xApp creates a RIC CONTROL message based on a RIC INDICATION, it should use the same sub_id value as the indication. Similarly, if the xApp forwards the RIC INDICATION, it should use the same sub_id value.

The xApp may choose to implement its logic based on the sub_id or may ignore the sub_id in its logic as illustrated below:

```
receive message;
switch (msg.sub_id) {
    case 1 :
        process message related to subscription 1;
        break;
    case 2:
        process message related to subscription 2;
}

OR

receive message;
if (msg.mtype == RIC INDICATION) {
    decode message;
    if msg.procedure_code = x and type_of_message = y ... {
        process message
```

Similarly, would a response to delete subscription request, or if the response is a failure also contain a subscription ID ?

Yes. The sub_id in these messages should be the same as the one returned in the original RIC SUBSCRIPTION RESPONSE message.

When the xAPP sends out a subscription, what should the gNodeB ID be (that is, the RMR "meid" field)?

ANSWER: The current guidance from use case team is that it would be the Global gNB ID (e.g., "NYN0001246"). For now, the xApp needs to know this name or read it from the R-NIB, but in the future we are planning of allowing an xApp to specify the meid with a special value denoting "ALL" and the Subscription Manager will send the subscription to all the gNBs that this xApp is allowed to manage (possible R2 feature).

When the xAPP sends a subscription request, the Subscription Manager is supposed to intercept the message, exchange it with RAN and then send back a response to the xAPP. How does the Subscription Manager know 'which' xAPP to send the response back to?

RMR has been extended with an operation (rmr_get_src) that allows any message receiver (including Subscription Manager) to read the sender information (hostname, port). Given this information, the Subscription Manager can populate RMR routes that will deliver the response to the original sender (routing based on message type and the sub_id).

In terms of xAPP behavior, if an xAPP dies and returns back, should it retry all subscriptions or is it the xAPP's responsibility to store the subscriptions in a persistent manner?

The fact that an xApp dies and then recovers does not cancel the subscriptions in the RAN. So, the recover xApp will automatically start receiving the messages that it previous subscribed to. The cleanest solution would likely be that the xApp persists its subscriptions and thus know what it subscribed to when it recovers.

Note that if the xApp accidentally reissues its subscriptions, the Subscription Manager should detect them as duplicates and simply merge into the existing subscriptions (no new subscriptions to RAN). Details to be ironed out in R2.

Also, are there any expiry times for subscriptions after which they need to be renewed ? Without an expiry time, I can see (in an unstable environment), new subscriptions quickly piling up.

To our knowledge, there has not been any discussion yet subscriptions expiring in the RAN.

Can I dump the contents of the route table that RMr is using?

Kind of. RMr supports a verbose option for the thread that collects route information from a static file, or from the Route Manager. Parsing information as it parses the entries, as well as a before and after summary of the table can be generated. It's not perfect, but it is useful for some testing. Details are on the RMr Route Table Dump page

What are the CMake configuration options that are supported?

When running the 'cmake' command to configure RMr and create the Makefile, it is possible to add one of several flags to the command line (e.g. cmake .. -DDEV_PKG). the following is a list of the supported flags:

- -DDEV_PKG=1 turn on development mode. This causes the header files, and the archive (.a) to be included in the resulting package(s) and/or installed when 'make install' is executed. If this flag is omitted, then a run-time package is generated which does not include the header files or the archives.
- -DBUILD_DOC=1 this causes the document formatting tool ({X}fm) to be loaded at configuration time, and the RMr man pages to be generated. If the development package option is set, the man pages will be installed such that commands like man rmr are available.
- -DPACK_EXTERNALS=1 Causes the Nanomsg and NNG libraries to be placed into the package and/or installed. (This is meant **only** for testing and should never be used when generating a deployable package.)
- -DSKIP_EXTERNALS=0 Causes the build process to assume that Nanomsg and NNG are currently installed in the developement environment and do not need to be explicitly built.

Can I use epoll_wait() to know when messages are ready to receive?

Yes, when the underlying transport mechanism supports it; NNG supports this. Calling the RMR function rmr_get_rcvfd() will return a file descriptor which will be "readable" whenever a message is pending and calling rmr_rcv_msg() will not block. The following code illustrates one way to set up for this.

```
if( (rcv_fd = rmr_get_rcvfd( mrc )) >= 0 ) {            // if a valid fd comes back, set up epoll stuff
     if( rcv_fd < 0 ) {
         fprintf( stderr, "<SNDR> unable to set up polling fd\n" );
         exit( 1 );
     }
     if( (ep_fd = epoll_create1( 0 )) < 0 ) {
         fprintf( stderr, "<SNDR> [FAIL] unable to create epoll fd: %d\n", errno );
         exit( 1 );
     }
     epe.events = EPOLLIN;
     epe.data.fd = rcv_fd;

     if( epoll_ctl( ep_fd, EPOLL_CTL_ADD, rcv_fd, &epe ) != 0 )  {
         fprintf( stderr, "<SNDR> [FAIL] epoll_ctl status not 0 : %s\n", strerror( errno ) );
         exit( 1 );
     }
 }
```

A word of caution: the EPOLLET (edge trigger) flag seems not to work with NNG. NNG will set the FD readable until the queue empties, but doesn't set the edge trigger.