# xApp Mock

# 1. Requirements

## 1.1 Summary

Many messages are sent and received from external x-apps to E2 Terminator (not through E2 Manager) or are sent from RAN to xApps through E2 terminator

Until such xApps will be developed, a simple lightweight x-app mock will be developed to test these flows.

Request from x-app

1. Enable user to trigger sending messages.
2. Send ASN1 encoded messages (from configuration) via RMR to E2 Terminator,
3. Receive responses from E2 Terminator if any
4. Write to stdout (don't decode them)
5. Receive RAN requests from E2 Terminator and send configured encoded responses, if required.
6. The flows:
    a. Send RIC SUBSCRIPTION REQUEST and receive RIC SUBSCRIPTION RESPONSE
    b. Send RIC SUBSCRIPTION DELETE REQUEST and receive RIC SUBSCRIPTION DELETE RESPONSE

Request from RAN

- The mock should receive request from E2 Terminator and write to stdout and send encoded configured response, if required.
- Supported requests:
    1. RESOURCE STATUS UPDATE
    2. RIC INDICATION

- RIC CONTROL

1. RIC CONTROL ACKNOWLEDGE
2. RIC CONTROL FAILURE
3. ERROR INDICATION

# 2. Solution

## 2.1 Proposed Solution

- The mock xApp is implemented as a command line tool which is started per scenario and terminates when the scenario is completed.
- The mock should be configurable to support various messages and to, possibly, allow responding with different kinds of messages for the same received message.
- The Mock shall accept a request on the command, combine the request with configuration to construct a message and send it to the E2 Term.
- The Mock shall also listen for messages received on the RMR endpoint log them and may respond to them.
- The Mock shall be delivered as a docker image to manage its dependencies.

## 2.2 Phases

### 2.2.1 Phase one

- The Mock shall support RIC SUBSCRIPTION REQUEST/RESPONSE in phase 1.
- Prepare ASN1 packed representation of the supported requests and responses.

## 2.3 Fail/Recovery (Error Handling)

- Errors will be written using Go's built in log package to stderr.

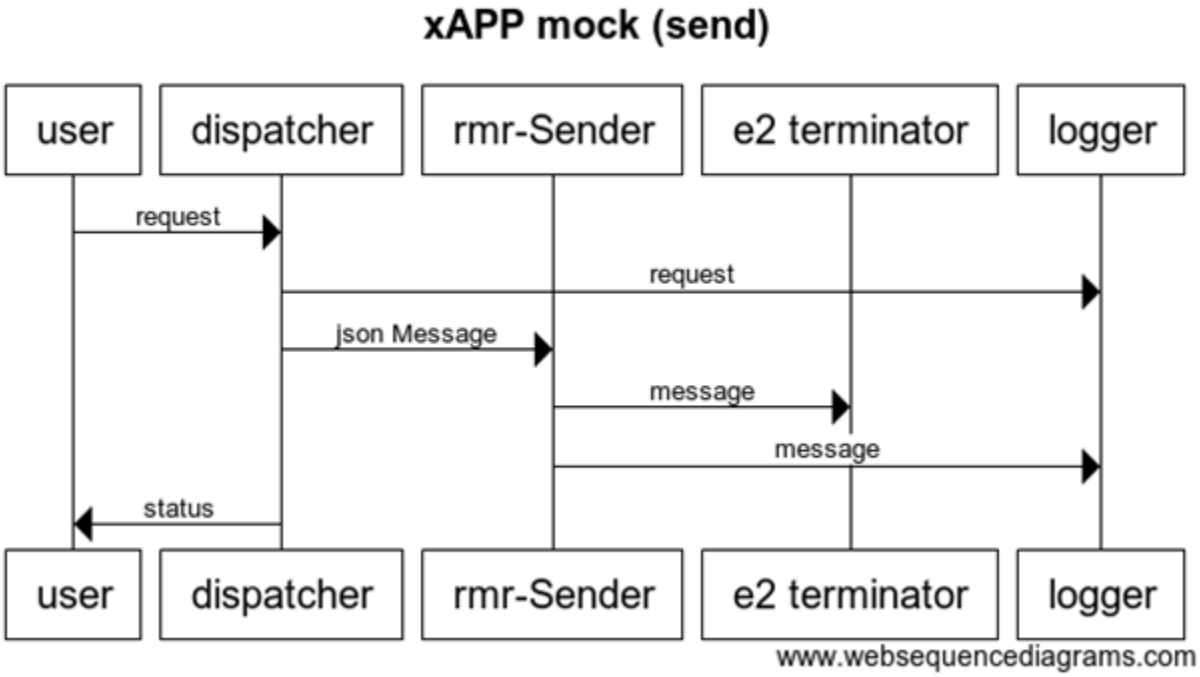
## 2.4 Security

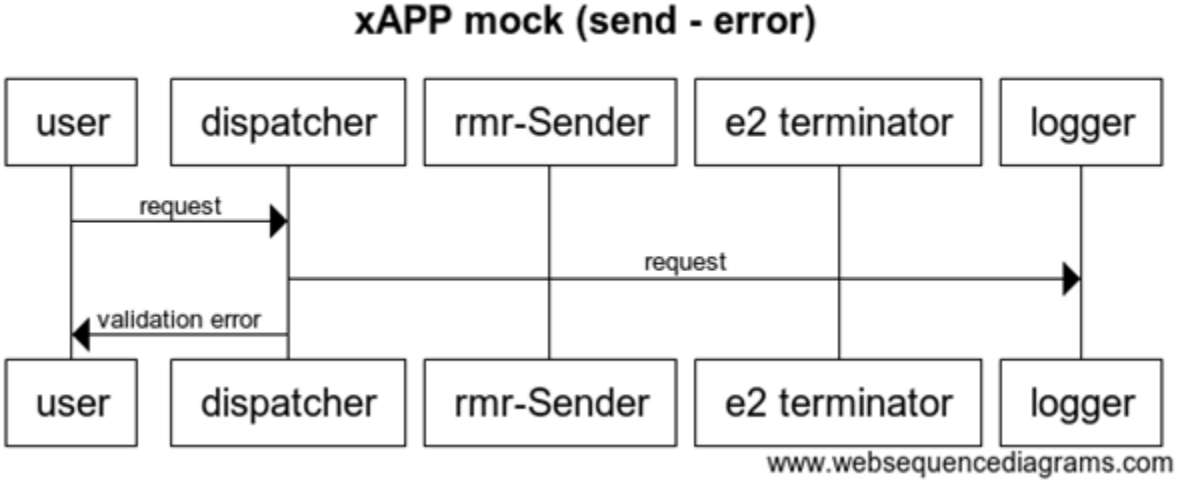
- N/A

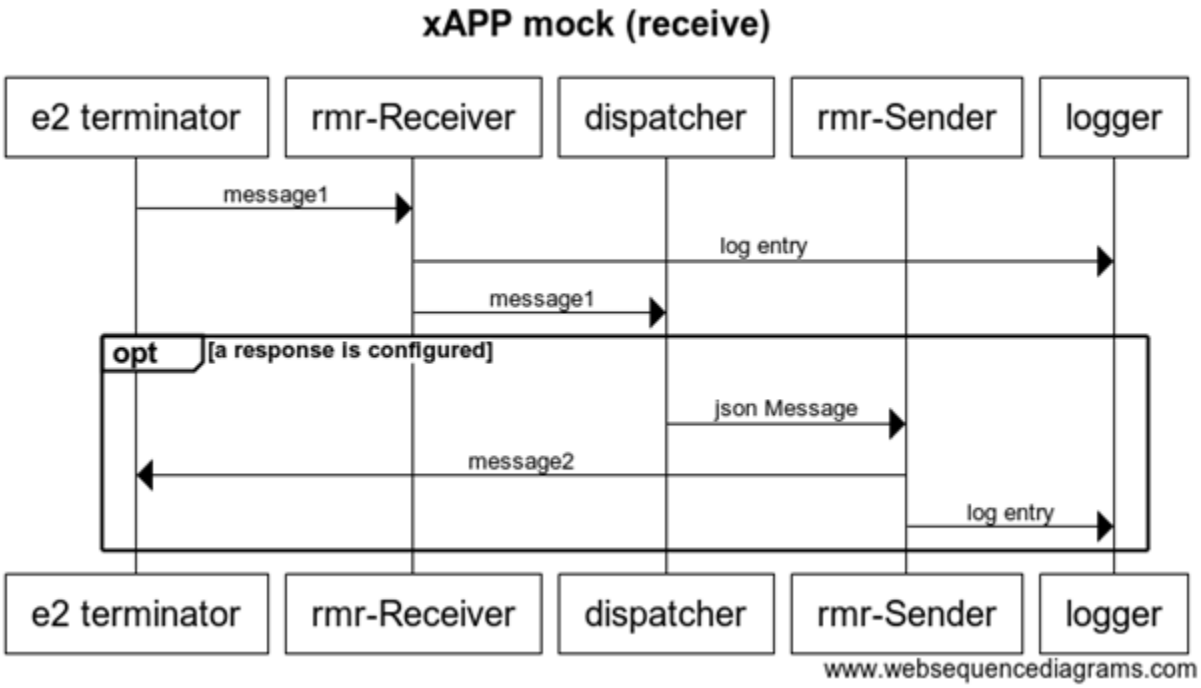
## 2.5 Logging

- All messages sent to the E2 Term and received from the E2 Term will be written using Go's built in log package to stdout. The format of the entries shall be a json object with at least the following members:
    - date: A human readable date and time.
    - timestamp: The number of milliseconds since the epoch.
    - id: The value of "id" from the configuration file for the sent message.
    - transactionId: The transaction id of the messages involved in the flow.
    - source: The source(sender) of the message.
    - destination: The destination(receiver) of the message.
    - payload: The payload of the message.
    - Request: The user's request.
    - Additional…

## 2.6 High Level Flow






# 3. File formats

## 3.1 Configuration file

The format of the configuration file is an array of Json objects, each object has the following format

{ “id”: “<id>” , “rmrMessageType”: <integer|reserved name>, “transactionId”: “<tid>”, “ranName”: “<name>”, “ranIp”: “<ip>”, “ranPort”: <integer>,

“payloadHeader”: “<string>”, “packedPayload”: “<sequence as produced by %x>”, “payload”: “text”}

- id (Mandatory): The id is a free form value and may serve to identify the command/flow. It may be the same as the rmrMessageType.
- rmrMessageType (Mandatory): The id number of the RMR message.
- payloadHeader: A prefix to combine with the payload that will be the message’s payload. The value may include variables of the format $<name> or #<name> where:
  - $<name> expands to the value of <name> if it exists or the empty string if not.
  - #<name> expands to the length of the value of <name> if it exists or omitted if not. The intention is to allow the Mock to construct the payload header required by the setup messages (ranIp|ranPort|ranName|payload len|<payload>).
- transactionId (xAction): The value may have a fixed value or $ or <prefix>$. $ is replaced by a value generated at runtime (possibly unique per message sent). If the tag does not exist, then the mock shall use the value taken from the incoming message.
- msgType, ranName, ranIp, ranPort and packedPayload (ASN1 payload) are the known members of the rmr message.

The Mock shall load the configuration file, on startup, and use the id to associate them to rmr message types or to the id specified in the user request to the Mock.

Failure to process the configuration file or no configuration file will trigger a panic (the process will fail to start).

## 3.2 Output file

N/A (see 3.5 Logging).

## 3.3 Configration (Docker level)

The following environment variables are required:

LD_LIBRARY_PATH=/usr/local/lib  - location of the rmr shared libraries

RMR_SEED_RT=router.txt  - routing data for the rmr infrastructure.

RMR_PORT = The port of the Mock's RMR endpoint.

# 4. Application sub-systems

## 4.1 Rmr-Sender

The *rmr-Sender* accepts a json object from the dispatcher and converts it to an RMR message and passes it to the remote. The sub-system supports the plain RMR message and the E2Term RMR message.

## 4.2 Rmr-Receiver

The *rmr-receiver* receives a message from the remote and passes a representation of it to the log and to the dispatcher (which may trigger the sending of a new RMR message).

Failure to establish the receiver will trigger a panic (the process will fail to start).

## 4.3 Dispatcher

The *dispatcher* executes the business logic of the Mock. On the one end it accepts the user request and consults the configuration file to decide which RMR message to send. It combines the parameters passed in the user request and relevant configuration data and construct the object that will be passed to the rmr-Sender.

On the other end, it accepts a message received by the RMR endpoint, consults the configuration file to decide which RMR message to send by searching for a match between the rmr message type and the id in the configuration, if any, and construct the object that will be passed to the rmr-Sender or returned to the user.

Example #1:

- Send x2 setup request and wait for a response.

{"action": "send", "id": "RIC_X2_SETUP_REQ", "ranName": "ran", "ranIp": "ip", "ranPort": port, "waitForRmrMessageType": "RIC_X2_SETUP_RESP"},

Example #2:

- Wait for ENB update and send a positive ack

{"action": "receive", rmrMessageType: "RIC_ENB_CONF_UPDATE", "id": "RIC_ENB_CONF_UPDATE_ACK_positive", "ranName": "ran", <additional parameter>},

Configuration:

{"id": "RIC_X2_SETUP_REQ" , "type": "rmrMessage",  "rmrMessageType": 10060, "transactionId": "e2e$", "payloadHeader": "$ranIp|$ranPort|$ranName|#packedPayload|",  "packedPayload": …., }

{"id": "RIC_ENB_CONF_UPDATE_ACK_positive" , "type": "rmrMessage",  "rmrMessageType": "RIC_ENB_CONF_UPDATE_ACK", "transactionId": "e2e$", "payloadHeader": "$ranIp|$ranPort|$ranName|#packedPayload|",  "packedPayload": …., }

## 4.4 Command line interface

The Mock shall expect a single json object, an array of json objects will be rejected, on the command line that it will process as a single flow.

- action (Mandatory): Possible values “send” and “receive”. When the action is of type “receive” the Mock shall wait until terminated.
- id (Mandatory): Must match one of the “id” tags in the Mock’s configuration file.
- rmrMessageType(Mandatory for “receive”)
- waitForRmrMessageType: Wait, indefinitely, for the specified RMR message and terminate when it is received.
- ranName: May override the value of the same tag in the configuration file.
- ranIp: May override the value of the same tag in the configuration file.
- ranPort: May override the value of the same tag in the configuration file.
- Other(?): let the request override all the matching tags in the configuration file.