xApp Writer's Guide

Abukar Mohamed, Shraboni Jana, Matti Hiltunen, Zhe Huang,
Ron Shacham, Tommy Carpenter, and Scott Daniels

June 22, 2021

Abstract

This document describes how to write xApps and how to deploy them on the RIC platform.

1 Introduction

Logically, an xApp is a entity that implements a well-defined function. Mechanically, an xApp is a K8s pod that is currently (Amber) restricted to have one container. In order for an xApp to be deployable, it needs to have an xApp descriptor (JSON) that describes the xApp's configuration parameters and information the RIC platform needs to configure the RIC platform for the xApp. The xApp writer will also need to provide a JSON schema for the descriptor.

In addition to these basic requirements, an xApp may do any of the following:

- Read initial configuration parameters (passed in the xApp descriptor).
- Receive updated configuration parameters.
- Send and receive messages.
- Read and write into a persistent storage (key-value store).
- Receive A1-P policy guidance messages - specifically operations to create or delete a policy instance (JSON payload on an RMR message) related to a given policy type.
- Define a new A1 policy type.
- Make subscriptions via E2 interface to the RAN, receive E2 INDICATION messages from the RAN, and issue E2 POLICY and CONTROL messages to the RAN.
- Report metrics related to its own execution or observed RAN events.

The lifecycle of an xApp consists of the following states:

- Development: Design, implementation, local testing.
- Released: The xApp code and xapp descriptor are committed to LF Gerrit repo and included in an O-RAN release. The xApp is packaged as Docker container and its image released to LF Release registry.
- On-boarded/Distributed: The xApp descriptor (and potentially helm chart) is customized for a given RIC environment and the resulting customized helm chart is stored in a local helm chart repo used by the RIC environment' s xApp Manager.
- Run-time Parameters Configuration: Before the xApp can be deployed, run-time helm chart parameters will be provided by the operator to customized the xApp Kubernetes deployment instance. This procedure is mainly used to configure run-time unique helm chart parameters such as instance UUID, liveness check, east-bound and north-bound service endpoints (e.g., DBAAS entry, VES collector endpoint) and so on.
- Deployed: The xApp has been deployed via the xApp Manager and the xApp pod is running on a RIC instance. For xApps where it makes sense, the deployed status may be further divided into additional states controlled via xApp configuration updates. For example,
  - Running
  - Stopped

2 xApp Specification and LocaL Testing

2.1 xApp development

XApps can be written using the RIC utility libraries (RMR, SDL, logging, etc) directly or by utilizing the xApp frameworks (in go, C++, and Python). The xApp frameworks are described in 4.

For RIC xApps to be deployable, they need to have a proper docker im- age generated and available in a accessible docker registry, and a valid xApp descriptor.

2.2 xApp Descriptor

The xApp descriptor is provided by the xApp developer. xApp Descriptor includes all the basic and essential information for the RIC platform to manage the life cycle of the xApp. Information and configuration included in the xApp descriptor will be used to generate the xApp helm charts and define the data flows to the north and south bound traffics. xApp developer can also include self-defined internal parameters that will be consumed by the xApp in the xApp descriptor.

The fields in the xApp descriptor is defined by the xApp JSON schema. A valid xApp descriptor will have to pass the xApp JSON schema validation. Please see the schema section for the details.

The xApp descriptor follows a JSON structure. The following are the key sections that defines an xApp.

- **name**: (REQUIRED) this is the unique identifier to address an xApp. A valid xApp descriptor must includes the xapp_name attribute. The following is an example.

```
"name":  "example_xapp",
```

- **version**: (REQUIRED) this is the semantic version number of the xApp descriptor. It defines the version numbers of the xApp artifacts (e.g., xApp helm charts) that will be generated from the xApp descriptor. Together with the xapp_name, they defines the unique identifier of an xApp artifact that can be on-boarded, distributed and deployed. The following is an example.

```
"version": "1.0.0",
```

- **containers**: (REQUIRED) This section defines a list of containers that the xApp will run. For each container, a structure that defines the container name, image registry, image name, image tag, and the command that it runs is defined. The name and images are REQUIRED. The command and argument lists are optional. The following is an example that defines two containers.

```
"containers": [
    {
        "name":  "example_container_1",
        "image":  {
            "registry":  "example_image_registry_1",
            "name": "example_image_name_1",
            "tag":  "example_image_tag_1"
        },
        "command": ["example_command_1"],
        "args": ["example_argument_1"]
        "resources": {
                    "limits": {},
                     "requests": {}
                     }
    },
    {
        "name": "example_container_2",
        "image":  {
            "registry":  "example_image_registry_2",
            "name": "example_image_name_2",
            "tag":  "example_image_tag_2"
        },
        "command": ["example_command_2"],
        "args": ["example_argument_2"]
        "resources": {
                    "limits": {},
                     "requests": {}
                     }
```

}
],

- **controls**: (Optional) The control section holds the internal configuration of the xApp. Therefore, this section is xApp specific. This section can include arbitrary number of xApp defined parameters. The xApp consumes the parameters by reading the xApp descriptor file that will be injected into the container as a JSON file. An environment variable XAPP DESCRIPTOR PATH will point to the directory where the JSON file is mounted inside the container. If the controls section is not empty, the xApp developer must provide the schema file for the controls section. Please refer to Schema for xApp Descriptor for creating such schema file. The following is an example for the controls section.

```
"controls": {
    "active": True,
    "requestorId": 66,
    "ranFunctionId": 1,
    "ricActionId": 0,
    "interfaceId": {
        "globalENBId": {
            "plmnId": "310150",
            "eNBId": 202251
        }
    }
},
```

- **metrics**: (Optional) The metrics section of the xApp descriptor holds information about metrics provided by the xApp. Each metrics item requires the *objectName*, *objectInstance*, *name*, *type* and *description* of each counter. The metrics section is required by VESPA manager (RIC platform component) and the actual metrics data are exposed to external servers via Prometheus/VESPA interface. The following is an example.

```
"metrics": [
    {
        "objectName": "UEEventStreamingCounters",
        "objectInstance": "SgNBAdditionRequest",
        "name": "SgNBAdditionRequest",
        "type": "counter",
        "description": "Total num. of SG addition req. events processed"
    },
    {
        "objectName": "UEEventStreamingCounters",
        "objectInstance": "SgNBAdditionRequestAcknowledge",
        "name": "SgNBAdditionRequestAcknowledge",
        "type": "counter",
        "description": "Total num. of SG addition req. ack. events processed"
    }
]
```

- **messaging**: (Optional) this section defines the communication ports for each containers. It may define list of RX and TX message types, and the

A1 policies for RMR communications implemented by this xApp. Each defined port will creates a K8S service port that are mapped to the container at the same port number. This section requires ports that contains the port name, port number, which container it is for. For RMR port, it also requires tx and rx message types, and A1 policy list.

```
"messaging": {
    "ports": [
        {
            "name": "http",
            "container": "mcxapp",
            "port": 8080,
            "description": "http service"
        },
        {
            "name": "rmrdata",
            "container": "mcxapp",
            "port": 4560,
            "txMessages":
            [
                "RIC_SUB_REQ",
                "RIC_SUB_DEL_REQ"
            ],
            "rxMessages":
            [
                "RIC_SUB_RESP",
                "RIC_SUB_FAILURE",
                "RIC_SUB_DEL_RESP",
                "RIC_INDICATION"
            ],
            "policies": [1,2],
            "description": "rmr data port for mcxapp"
        },
        {
            "name": "rmrroute",
            "container": "mcxapp",
            "port": 4561,
            "description": "rmr route port for mcxapp"
        }
    ]
},
```

Choosing port numbers:
In the bronze release onwards appmgr is not consuming the port name defined in the messaging section yet. Please choseto use the default 4560 port for rmr-data and 4561 for rmr-route.

Port naming convention:
Kubernetes requires the port name to be DNS compatible. Therefore, please choose a port name that contains only alphabetical characters (A-Z), numeric characters (0-9),the minus sign (-), and the period (.). Period characters are allowed only when they are used to delimit the components of domain style names.

- **liveness probes**: (Optional) The liveness probe section defines how liveness probe is defined in the xApp helm charts. You can provide ether a command or a http helm liveness probe definition in JSON format. This section requires initialDelaySeconds, periodSeconds, and either httpGet or exec. The following is an example for http-based liveness probes.

```
"livenessProbe": {
    "httpGet": {
        "path": "ric/v1/health/alive",
        "port": "8080"
    },
    "initialDelaySeconds": "5",
    "periodSeconds": "15"
},
```

The following is an example for RMR-based liveness probes

```
"livenessProbe": {
    "exec": {
        "command": ["/usr/local/bin/rmr_probe"]
    },
    "initialDelaySeconds": "5",
    "periodSeconds": "15"
},
```

- **readiness probes**: (Optional) The readiness probe section defines how readiness probe is defined in the xApp helm charts. You can provide ether a command or a http helm readiness probe definition in JSON for- mat. This section requires initialDelaySeconds, periodSeconds, and ei- ther httpGet or exec.The following is an example for http-based readiness probes.

```
"readinessProbe": {
    "httpGet": {
        "path": "ric/v1/health/alive",
        "port": "8080"
    },
    "initialDelaySeconds": "5",
    "periodSeconds": "15"
},
```

The following is an example for RMR-based readiness probes

```
"readinessProbe": {
    "exec": {
        "command": ["/usr/local/bin/rmr_probe"]
    },
    "initialDelaySeconds": "5",
    "periodSeconds": "15"
},
```

2.3 Schema for the xApp Descriptor

JSON schema is used to describe the attributes and values in the xApp descriptor JSON file. The xApp onboarding process verifies the types and values of the

xApp parameters in the descriptor. If mismatches are found, xApp onboarding will return failure. The schema file consists of two parts: sections that are static and cannot be changed for different xApp, and xApp specific controls section. When an operator is onboarding an xApp that defines a control section, he/she will provide the controls section schema with together with the xApp descriptor. The xapp onboarder will combine the schema files into one.

2.3.1 How to Create Schema for the Controls Section

You can submit arbitrary schema for the controls section. However, if the xApp descriptor contains a controls section, you have to provide the correct schema that describes it. If the xApp does not require a control section, you can ignore the control section schema. It is highly recommended to use draft-07 schema. The following is a skeleton schema that you can use

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "$id": "#/controls",
    "type": "object",
    "title": "Controls Section Schema",
    "required": [
        "REQUIRED_ITEM_1",
        "REQUIRED_ITEM_2"
    ],
    "properties": {
        "REQUIRED_ITEM_1": {REQUIRED_ITEM_1_SUB_SCHEMA},
        "REQUIRED_ITEM_2": {REQUIRED_ITEM_2_SUB_SCHEMA}
    }
}
```

This section will be replaced by the submitted control section schema. If the controls section schema is not provided, the following will beused to make sure that the xApp onboarder will reject an xApp descriptorwith an undefined control section.

```
"controls": {
    "required": [
        "_empty_control_section_"
    ]
}
```

2.4 Local Testing

To test onboard an xApp, you can utilize either the DMS CLI tool. The DMS CLI tool will return errors that can help you pinpoint problems in your xApp descriptor.

2.4.1 Prerequisites

The DMS CLI tool requires a host with docker daemon and local helm repo installed.We recommend using helm version v3.5.x and above. If you are using a Ubuntu host, you can use the following commands to prepare your environment for the DMS CLI tool.

```
sudo apt-get update
```

```
sudo apt install docker.io
```

2.4.2 Create a local helm repo

You can create a local helm repo by running the following command

```
docker run -d -p 8080:8080 -e DEBUG=1 -e STORAGE=local \
-e STORAGE_LOCAL_ROOTDIR="/charts" chartmuseum/chartmuseum:latest
```

2.5 Test On-boarding using xApp-onboarder/DMS CLI Tools

Please replace the blue text with the correct values for your xApp.

1. Install xapp-onboarder DMS CLI

```
git clone "https://gerrit.o-ran-sc.org/r/ric-plt/appmgr"
dev/xapp_onboarder
pip3 install ./
```

2. Set up the environment variables for DMS CLI connection

```
export CHART_REPO_URL="http://0.0.0.0:8080"
# It should return True if your DMS CLI tool is properly connected to the RIC
instancecli health
```

3. Onboard your xApp. Please refer to xApp descriptor for preparing for the xApp descriptor

```
# Make sure that you have the xapp descriptor
# config file and the schema file at your local file system
cli onboard CONFIG_FILE_PATH SHCEMA_FILE_PATH
```

If onboarding fails, the DMS CLI tool will return you messages that indicatewhere the errors are in the descriptor.

4. (OPTIONAL) Download the xApp helm charts

```
dms_cli download_helm_chart --xapp_chart_name=XAPP_CHART_NAME --version=VERSION --output_path=OUTPUT_PATH
```

5. (OPTIONAL) Download the xApp override values.yaml file

```
dms_cli download_values_yaml --xapp_chart_name=XAPP_CHART_NAME --version=VERSION --output_path=OUTPUT_PATH
```

2.6 xApp deployment and undeployment

To deploy an xApp named ” EXAMPLE-XAPP” , you need a fully functioning near real-time RIC platform. Please refer to the other guide about how to create one.

To deploy xApp using the DMS CLI , run the following commands.

1. Run DMS CLI to install xapp

```
dms_cli install --xapp_chart_name=XAPP_CHART_NAME --version=VERSION --namespace=NAMESPACE
```

2. Run DMS CLI to uninstall xapp

```
dms_cli uninstall --xapp_chart_name=XAPP_CHART_NAME --namespace=NAMESPACE
```

3 Functions

The RIC platform provides a set of functions that the xApps can use to accomplish their tasks.

3.1 Registering/De-registering Xapp

The xapp after deployment needs to be registered to the RIC platform by sharing its config to the platform.

Otherwise RMR and other functionalities described later will not be available for the xapp. This could be accomplished using REST API's described below.

1. Registration to xapp manager

Xapp indicates RIC platform i.e xapp manager to register itself.

```
curl -X 'POST' 'http://<appmgr_svcIP>:8080/ric/v1/register' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{
  "appName": "mcxapp",
  "appVersion": "1.0.0",
  "configPath": "/opt/ric/config",
  "appInstanceName": "mcxappinstance",
  "httpEndpoint": "10.103.5.170:8080",
  "rmrEndpoint": " 10.109.124.128:4560",
  "config": "xapp config in JSON"
}'
```

Here the parameters marked in bold are mandatory and are self-explanatory. The "configPath", is the http URL. If this parameter is not set the default will be set to "/ric/v1/config". This parameter indicates the xapp manager that when it queries for the xapp config, it needs to add this URL in its request.

The xapp can also send its config during registration itself, by populating "config" parameter. When this parameter is set, xapp manager will not query for the config again.

2. Get Config query from xapp to xapp manager

Xapp needs to implement this REST interface when xapp -manager queries for it. If during registration if the configuration is already sent, xapp manager would not request again for the configuration.

This request will be sent by xapp-manager.

```
curl -X 'GET' 'http://<xapp:8080>/ric/v1/config' -H 'accept: application/json'
```

In return the xapp must respond with the below JSON response.

```
[
  {
    "metadata": {
      "xappName": "string",
      "configType": "json"
    },
    "config": {}
  }
]
```

3. De-Registration from xapp manager

Xapp indicates to un-register from RIC platform

curl -X 'POST' 'http://<appmgr_svcIP>:8080/ric/v1/deregister' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{

"appName": "mcxapp",

"appInstanceName": "mcxappinstance"

}

NOTE: If golang based xapp-frame is used by xapps, the registration/un-registration of xapp and sending configurations to xapp-manager is a built-in feature. There is no extra implementation needed from xapp.

For xapp's that don't want to implement GET config API's, they can still register/unregister by sending REST API's 1 and 3 from the command line. To do that, ensure to fill the "config" parameter mandatorily. Here the idea is to send the registration and configuration in a single REST call.

Example:

curl -H "accept: application/json" -H "Content-Type: application/json" -d "@mcxapp.json" http://<appmgr_svcIP>:8080/ric/v1/register

where, mcxapp.json contains

{

"appName": "mcxapp",

"appVersion": "1.0.11",

"configPath": "",

"appInstanceName": "mcxapp",

"httpEndpoint": "1.2.3.4:8080",

"rmrEndpoint": "10.111.7.117:4560",

"config": " {\"messaging\":{\"ports\":[{\"container\":\"mcxapp\",\"description\":\"rmr send data port for mcxapp\",\"name\":\"rmr-data-out\",\"policies\":[],\"port\":4562,\"rxMessages\":[],\"txMessages\":[]},{\"container\":\"mcxapp\",\"description\":\"rmr receive data port for mcxapp\",\"name\":\"rmrdata\",\"policies\":[],\"port\":4560,\"rxMessages\":[\"RIC_UE_CONTEXT_RELEASE\",\"RIC_SGNB_ADDITION_REQ\",\"RIC_SGNB_ADDITION_ACK\...."}

3.2 Messaging - RMR

RMR is a message router library which an application can use to send messages to other RMR based applications. The destination of each message is governed by the message type and subscription ID, or just the message type. RMR is responsible for establishing and managing each connection freeing the application from any network connectivity management.The library is available in repo [1].

This repo contains the source and documentation for both the core RMR library and for the Python bindings which allow a Python (v3) application to use RMR to send and receive messages. The repo also contains examples which illustrate RMR use case. The README files for the repo and its directories are comprehensive for using and testing the library.

3.2.1 Using RMR from C++

RMR functionality is inherent to the Xapp Core System of C++ xApp framework. The section provides snippets of key RMR functions to be used.

```
//RMR context Initialization
void* xapp_rmr_ctx = rmr_init( listen_port, RMR_MAX_RCV_BYTES, RMRFL_NONE);

if( xapp_rmr_ctx == NULL ) {
  cout<< "Unable to initialise RMR Context" << endl;
  exit(1);
}

//Check if rmr context is ready before invoking sender/receiver
if( ! rmr_ready( xapp_rmr_ctx ) ) {
  cout<< "RMR Context Not ready" << endl;
}


// RMR Send Message
rmr_mbuf_t* send_msg = NULL;
// allocate the buffer
send_msg = rmr_alloc_msg(send_msg, RMR_DEF_SIZE);

char *msg = "MessagePayload 0";
//Copy the message to RMR Buffer.
memcpy(send_msg->payload, msg, msg_length);

//Send the message and check the status
send_msg = rmr_rcv_msg( xapp_rmr_ctx, send_msg );
if( send_msg->mtype < 0 || send_msg->state != RMR_OK )
  cout << "Bad Message State: " << send_msg->state << endl;
  exit( 1 );
}

//RMR Receive Message
rmr_mbuf_t* rcv_msg = NULL;
rcv_msg = rmr_rcv_msg( xapp_rmr_ctx, rcv_msg );
if( rcv_msg->mtype < 0 || rcv_msg->state != RMR_OK ) {
  cout << "Bad Message State: " << rcv_msg->state << endl;
  exit( 1 );
}
```

3.2.2 Using RMR from go

RIC xApps can communicate with other each and with other RIC platform components via RMR, which is a very thin but robust library that allow applications

to send and receives RMR messages. RMR message consists of message header and payload. The message header defines the type of the message to be sent, a subscription ID that identifies a subscription of routing entry, a MEID, which the global RAN name and a application specific transaction ID. The message payload carries the actual user data.

Below is a sample code that attempts to send RMR message:

```
    meid := &xapp.RMRMeid{RanName: RanName
mtype, _ := xapp.Rmr.GetRicMessageId(msgName)
data := []byte{1,2,3}
params := &xapp.RMRParams{Mtype: mtype, SubId: -1, Meid: meid, Xid: txid,
                          Payload: data, PayloadLen: len(data)}

if ok := xapp.Rmr.SendMsg(params); !ok {
  xapp.Logger.Error("Sending '%s' with txid=%s failed!", msgName, txid)
  if xapp.Rmr.IsRetryError(params) {
    // Retry or do something smart
  }
  return
}
xapp.Logger.Info("RMR message sent successfully!")
```

3.2.3 Using RMR from Python

There are two ways to use RMR in Python; you can use the rmr python bindings directly, or you can use the python xapp framework. Here, we discuss the direct bindings and refer the reader to the "Frameworks" section for the latter. Note that the xapp framework for python uses this rmr bindings library internally.

The rmr python bindings are available in the rmr package in [pypi]. They are a CTYPES wrapper around the C rmr shared object; it is ***not*** a re-implementation of the API in python. As such the underlying rmr C library needs to be installed. As of rmr v4 and greater, it uses SI95 rather than NNG.

The library provides a direct translation of the RMR API in the rmr.rmr module, and some higher level helper functions in the rmr.helpers module.

Please see: [rmr examples] for full examples of send, receive, and some helper functions such as rmr_rcvall_msgs. Also, the [rmr unit tests] are a great resource for snippets of direct rmr usage.

Send snippet:

```
# simultaneously alloc and set fields in it
pay = 'hello'
sbuf = rmr.rmr_alloc_msg(MRC,
      256,
      payload=pay,
      gen_transaction_id=True,
      mtype=14          ,meid='
      GNB10001',
      sub_id=654321)
summary = rmr.message_summary(sbuf). # generate a helpful summary
sbuf_send = rmr.rmr_send_msg(MRC SEND, sbuf_send)
send_summary = rmr.message_summary(sbuf_send)
```

```
assert send_summary["message_state"] == 0
```

Receive snippet:

```
# receive a single message
sbuf_rcv = rmr.rmr_alloc_msg(MRC_RCV, 256)
sbuf_rcv = rmr.rmr_torcv_msg(MRC_RCV, sbuf_rcv, 2000)

# get whole mailbox
for (msg, sbuf) in helpers.rmr_rcvall_msgs_raw(self.mrc):
    rmr.rmr_free(sbuf)
    print(summary)
    self.rmr_free(sbuf)
```

3.3 Shared data layer - SDL

Shared data layer provides xApps the capability to share data directly via database. SDL APIs provide simple yet flexible way to store and retrieve data while hiding all the unnecessary details such as type and location of database, all management operations of database layer such as high availability, scaling, load-balancing.

SDL uses following 4 environment variables to connect to Redis instance: DBAAS_SERVICE_HOST, DBAAS_SERVICE_PORT, DBAAS_SERVICE_SENTINEL_PORT and DBAAS_MASTER_NAME

Information about various SDL APIs and usage can be found at [].

3.3.1 Using SDL from C++

Depending on the synchronous or asynchronous transaction required APIs can be called accordingly. Sample code below:

```
#include <sdl/syncstorage.hpp>

//data type definitions from sdl
using Namespace = std::string;
using Key = std::string;
using Data = std::vector<uint8_t>;
using DataMap = std::map<Key, Data>;
using Keys = std::set<Key>;

void func() {
        std::unique_ptr<shareddatalayer::SyncStorage> sdl(shareddatalayer::SyncStorage::create())

        Namespace ns("xapp-dev"); //define namespace

          Keys K = sdl->findKeys(ns,...);
          sdl->set(ns,...);
          ...
}
```

3.4 Using R-NIB

RIC applications can obtain the list of GNBs connected/discovered, and as well as their connection status stored in RNIB.

3.4.1 Using R-NIB from go

Following example shows how the GO applications can access RNIB and extract GNB related information.

```
// Get the list of GNBs
gnbs, err := xapp.Rnib.GetListGnbIds()
if err != nil || len(gnbs) == 0
    if err != nil {
        xapp.Logger.Error("GetListGnbIds failed with error: %v", err)
    }
    if len(gnbs) == 0 {
        xapp.Logger.Info("gNBs not discovered yet!")
    }
    return
}

// Print the RAN name and connection status of each GNB
for _, gnb := range gnbs {
    ranName := gnb.GetInventoryName()
    info, err := xapp.Rnib.GetNodeb(ranName)
     if err != nil {
        log.Error("GetNodeb() failed for ranName=%s: %v", ranName, err)
         continue
    }
    xapp.Logger.Info("NodeB[%s] connection status = %d", ranName,
                        nodeInfo.ConnectionStatus)
}
```

3.5 Watching for Config Changes

After starting up with the initial configuration, xApps can watch and read live the config file while running. In other words, xApps don't need to be restarted to have their config file changes to take effect. To watch for config file changes, xApps provide a callback function for xApp-framework to run whenever a file change occurs.

3.5.1 Watching for config changes in go

```
// Define the callback
  func (e *MyExampleXApp) StatusCB(f string)
        if appReady {
            return true
        } else      // Application not ready yet, do something
            return false
        }
```

```
}

// Register the callback to xApp framwork
xapp.Resource.InjectStatusCb(u.StatusCB)
```

3.6 Logging

The RIC platform provides a logging library that ensures that the log entries generated by xApps will have a standard format and will be handled uniformly. The ORAN wiki describing logging best practices can be found at [2].

The logging repository used in RIC platform with detailed documentation is available at [3]. The logging library writes the logs to stdout. Each log entry is one line. Fields in the log entry

- ts – Timestamp, number of milliseconds since Unix Epoch (i.e. 1970-01-01 00:00:00 +0000 (UTC)), set by the logging library
- crit – Severity level of the log, given by the application process: DEBUG, INFO, WARNING, ERROR
- id – the name of the process, set by the logging library
- msg – log message given by the application process
- mdc – a list of key value pairs, both strings, unique key names, given by the application process

3.6.1 Using logging from C++

Library initialization is an optional step, which can be done using mdclog_init() function. By calling mdclog_init() the library user can define the logger identity tag, which is added to every log entry by the library. For example -

```
#include<mdclog/mdclog.h>
//logtest.cc
void initlog()
{
    mdclog_attr_t *attr;
    mdclog_attr_init(&attr);
    mdclog_attr_set_ident(attr, "logtest"); //logtest will be the "id"(identity)
    mdclog_init(attr);
    mdclog_attr_destroy(attr);
}
void func() {
    initlog();
    ...
    //Severity Level INFO
    mdclog_write(MDCLOG_INFO,"The info is from, file=%s, line=%d", __FILE__,__LINE__);
    ...
    //Severity Level ERROR
    mdclog_write(MDCLOG_ERR,"The error is from, file=%s, line=%d", __FILE__,__LINE__);
```

```
    ...
    //Severity Level WARNING
    mdclog_write(MDCLOG_WARN,"The error is from, file=%s, line=%d", __FILE__,__LINE__);
    ...
https://www.overleaf.com/project/5e2b4816b24ba10001217583    //Severity Level DEBUG
    mdclog_write(MDCLOG_DEBUG,"The error is from, file=%s, line=%d", __FILE__,__LINE__);
    ...
}
```

If the mdclog_init() function is not called,in other words, init_log() is not called in ***func***(), the library uses the program name as the identity.For detailed documentation, refer [3] and [2].

3.7 ASN.1 Encoding/Decoding

XApp development uses ASN1C library [4] for encoding and decoding of asn messages. The E2AP, E2SM, X2AP, F2AP etc. asn files can be compiled into a set of .c and .h files.

```
`asn1c -fincludes-quoted -fcompound-names -fno-include-deps -findirect-choice -gen-PER -no-gen
```

For more details regarding the APIs provided by the library, refer [5].

3.7.1 Using ASN.1 from C++

Sample code to encode using PER encoding rules for E2AP_PDU_t structure created in memory.

```
E2AP_PDU_t *e2pdu = calloc(1, sizeof(E2AP_PDU_t));
....
 asn_enc_rval_t res =
          asn_encode_to_buffer(0, ATS_ALIGNED_BASIC_PER, &asn_DEF_E2AP_PDU, e2pdu, data_buf, *

    if(res.encoded == -1){
        mdclog_write(MDCLOG_ERR, "Failed to encode: %s, %s",asn_DEF_E2AP_PDU.name, errmsg_buff
    } else
        if(*data_size < res.encoded)
          mdclog_write(MDCLOG_ERR, "Buffer assigned small to encode: %s",asn_DEF_E2AP_PDU.name
        }
    }
```

And for decoding the encoded buffer into E2AP_PDU_t data structure using :

```
  E2AP_PDU_t *e2pdu = 0;
  asn_dec_rval_t rval;
  ASN_STRUCT_RESET(asn_DEF_E2AP_PDU, e2pdu);

  rval = asn_decode(0,ATS_ALIGNED_BASIC_PER, &asn_DEF_E2AP_PDU, (void **)&e2pdu, data_buf, data
  if(rval.code == RC_OK)
  {
      ...
  }
```

3.8 Metrics

RIC applications can act as metrics providers and expose the metrics data to external centralized time-series DB servers. Prometheus interface is used to periodically collect metrics data and forward to ONAP via VES agent.

GO xApp-framework provides generic interfaces to register various metrics types. Following example code defines two counters, registers using Prometheus GO-client interface and updates the counters when the respective messages are processed.

```
// Define two metrics counters that the xApp provides
metrics := []xapp.CounterOpts{
    {Name: "RICIndicationsRx", Help: "The total number of RIC indication events"},
    {Name: "RICExampleMessageRx", Help: "The total number of RIC example events"},
}

// Register the metrics
stats := xapp.Metric.RegisterCounterGroup(metrics, "ExampleXapp")

// Update E2APIndicationsRx counter when RIC indication message is received
stats["E2APIndicationsRx"].Inc()

// Update RICExampleMessageRx counter when RIC example message is received
stats["RICExampleMessageRx"].Inc()
```

3.9 Health checking

An xApp has to provide a method for K8s to check the health of the xApp. The RIC supports two types of health checks: HTTP-based and RMR-based.

3.9.1 Registering Status Callback for Health Checking in go

xApp-framework normally handles responding Kubernetes health probes (aliveness and readiness) autonomously. However, xApps can register a status callback function for xApp-framework to call each time a HTTP health probe request is received, and determine what kind of status is returned to Kubernetes. This is useful in case that something goes wrong during startup, or the application waits external command to proceed with the completion of the start-up. In these particular cases, applications can simply return negative response and status will be visible in Kubernetes as not ready. For example:

```
// Define the callback
  func (e *MyExampleXApp) ConfigChangeHandler(f string)
                    // Do something useful here
}
```

3.10 Using E2 from xApps

In the RIC Amber release, xApps directly implement the E2 protocol messages, that is, they construct E2 subscription/control messages in ASN.1 format and receive the E2 indication messages in ASN.1 format. The E2 message payload is sent to the RAN using an RMR message where the E2 ASN.1 message is the message payload and the RMR meid field is populated with the target E2 node id.

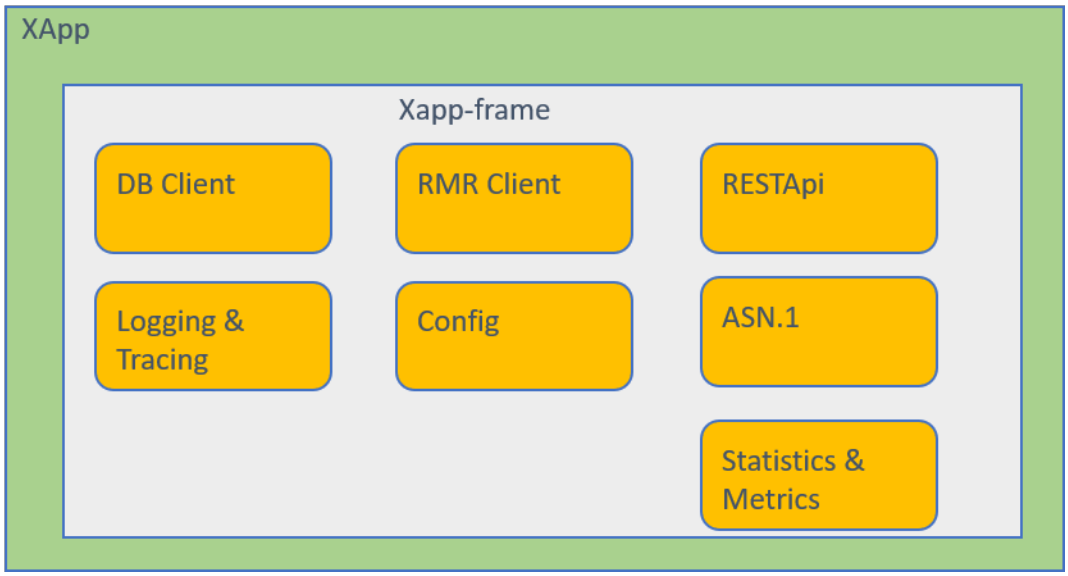
4 Frameworks

The RIC platform currently provides frameworks that make it easier to construct xApps in go and C++.

4.1 Go framework

xApp framework is a simple skeleton designed for rapid development of RIC xapps based on Go. Following figure depicts the high level architecture of xApp-framework, which consists of several loosely linked components that provide the common functions needed by the xApp developers.

For GO-based xApps, application developers don' t need to write xApp code from the scratch, but can use xApp-framework, which is designed to facilitate and rapidly build a full-fledged RIC xApps. The xApp-framework can be found ilink here¿ and a simple example-Xapp that illustrates how the framework can be used is located at ilink here¿. The dependencies and prerequisites of xApp development are also described in the above links.



The common functions and interfaces that the xApp-framework provides include:

- RESTful (new resources can be injected dynamically)
- RMR (with message filtering based on application rules)
- Database backend services (with configurable namespaces, etc)
- Watching and populating config-map changes in K8S environment
- Logging (via MDCLOG + tracing support in the future)
- ASN.1 decoding and encoding (only skeleton -¿ to be implemented)

  The GO xApp Framework supports various essential components such:

  - RESTful: HTTP services, health probes and injecting new resources can be dynamically
  - RMR client: Sending and receiving of RMR messages with message filtering based on application rules). See [xx]
  - DB client: Database backend services (with configurable namespaces, etc) for SDL, RNIB and UENIB. See [xx]
  - Config: Reading, watching and populating config-map changes in K8S environment on runtime
  - Logging and tracing: Generic system logging and tracing services (in the future). See [xx]
  - Metrics and statistics: Generating and publishing metrics to the metrics server (Prometheus interface used to collect (or "scrap") metrics and counters. See [xx]
  - ASN.1 decoding and encoding (only skeleton -¿ to be implemented). See [xx]

  For more information about the xApp Framework and other used library services, check ¡link here¿

4.1.1 Getting Started – A simple example xApp

The following is a sample xApp that receives RMR messages, stores message payload to SDL and echo the message back to the sender:

```
package main

import (
    "gerrit.o-ran-sc.org/r/ric-plt/xapp-frame/pkg/xapp"
)

type SampleXapp struct {
```

```
    appReady bool
}

func (e *SampleXapp) Consume(rp *xapp.RMRParams) (err error) {
    if err := xapp.Sdl.Store("myKey", r.Payload); err != nil {
        xapp.Logger.Info("Sdl.Store failed with error: %v", err)
    }

    if ok := xapp.Rmr.SendMsg(r); !ok {
        xapp.Logger.Info("Rmr.SendMsg failed ...")
    }
}

func NewSampleXapp(appReady bool) *SampleXapp {
    return &SampleXapp{
        appReady: appReady
    }
}

func main() {
    NewSampleXapp(true, false).Run()
}
```

Detailed explanation of the above code:

- We imported the package: "gerrit.o-ran-sc.org/r/ric-plt/xapp-frame/pkg/xapp". Note! that GO initializes imported packages and executes init() function in every package
- We defined a GO struct called SampleXapp with only one Boolean field appReady. The SampleXapp has only one method called Consume, which is mandatory.
- We wrote the logic of the Consume method:
  - it reads the payload from the message, stores the data to SDL using xApp-framework SDL interface: xapp.Sdl.Store("myKey", r.Payload)
  - ends back the message using xApp-framework RMR interface: xapp.Rmr.SendMsg(r)
- We created an instance of SampleXapp struct by using pointer address operator
- Finally, we defined the main function. All applications in Go use main as their entry point like C does

4.1.2 Compiling and running the sample xApp locally

Save the sample code as sample-xapp.go, build and run it locally using following commands:

```
GO111MODULE=on GO ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o sample-xapp sample-xa

./sample-xapp -f config/config-file.yaml
```

4.1.3 Generating Docker image and running the image in RIC environment

¡ add here some basic stuff of how to compile and generate a docker image for xApp, and run in RIC environement¿

4.2 C++ framework

The C++ framework allows the programmer to create an xApp object instance, and to use that instance as the logic base. The xApp object provides a message level interface to the RIC Message Router (RMR), including the ability to register callback functions which the instance will drive as messages are received; much in the same way that an X-windows application is driven by the window manager for all activity. The xApp may also choose to use it' s own send/receive loop, and thus is not required to use the callback driver mechanism provided by the framework.

4.2.1 C++ Framework API

The C++ framework API consists of the creation of the xApp object, and invoking desired functions via the instance of the object. The following paragraphs cover the various steps involved to create an xApp instance, wait for a route table to arrive, send a message, and wait for messages to arrive.

4.2.2 Creating the xApp instance

The creation of the xApp instance is as simple as invoking the object' s constructor with two required parameters:

port A C string (char *) which defines the port that RMR will open to listen for connections.

wait A Boolean value which indicates whether or not the initialization process should wait for the arrival of a valid route table before completing. When true is supplied, the initialization will not complete until RMR has received a valid route table (or one is located via the RMR_SEED_RT environment variable).

The following code sample illustrates the simplicity of creating the instance of the xApp object.

```
#include <memory>
#include <ricxfcpp/xapp.hpp>
int main( ) {
    std::unique_ptr<Xapp> xapp;
    char* listen_port = (char *) "4560";      //RMR listen port
    bool wait4table = true;                   // wait for a route table

    xapp = std::unique_ptr<Xapp>(
```

```
        new Xapp( listen_port, wait4table ) );
}
```

From a compilation perspective, the following is the simple compiler invocation string needed to compile and link the above program (assuming that the sample code exists in a file called man_ex1.cpp).

```
g++ man_ex1.cpp -g -o man_ex1 -lricxfcpp -lrmr_si -lpthread -lm
```

The above program, while complete and capable of being compiled, does nothing useful. When invoked, RMR will be initialized and will begin listening for a route table; blocking the return to the main program until one is received. When a valid route table arrives, initialization will complete and the program will exit as there is no code following the instruction to creeate the object.

4.2.3 Listening For Messages

The program in the previous example can be extended with just a few lines of code to enable it to receive and process messages. The application needs to register a callback function for each message type which it desires to process. Once registered, each time a message is received the registered callback for the message type will be invoked by the framework.

4.2.4 Callback Signature

As with most callback related systems, a callback must have a well known function signature which generally passes event related information and a "user" data pointer which was registered with the function. The following is the prototype which callback functions must be defined with:

```
void cb_name( Message& m, int mtype, int subid,
        int payload_len, Msg_component payload,
        void* usr_data );
```

The parameters passed to the callback function are as follows:

- m: A reference to the Message that was received.
- mtype: The message type (allows for disambiguation if the callback is registered for multiple message types).
- subid: The subscription ID from the message.
- payload len: The number of bytes which the sender has placed into the payload.
- payload: A direct reference (smart pointer) to the payload. (The smart pointer is wrapped in a special class in order to provide a custom destruction function without burdening the xApp developer with that knowledge.)

- user data: A pointer to user data. This is the pointer that was provided when the function was registered.

To illustrate the use of a callback function, the previous code example has been extended to add the function, register it for message types 1000 and 1001, and to invoke the Run() function in the framework (explained in the next section).

```
#include <memory>
#include <ricxfcpp/xapp.hpp>
long m1000_count = 0;      // message counters, one for each type
long m1001_count = 0;

// callback function that will increase the appropriate counter
void cbf( Message& mbuf, int mtype, int subid, int len,
            Msg_component payload, void* data ) {
    long* counter;

    if( (counter = (long *) data) != NULL ) {
        (*counter)++;
    }
}

int main( ) {
    std::unique_ptr<Xapp> xapp;
    char* listen_port = (char *) "4560";
    bool wait4table = false;

    xapp = std::unique_ptr<Xapp>(
          new Xapp( listen_port, wait4table ) );

    // register the same callback function for both msg types
    xapp->Add_msg_cb( 1000, cbf, (void *) &m1000_count );
    xapp->Add_msg_cb( 1001, cbf, (void *) &m1001_count );

    xapp->Run( 1 );          // start the callback driver
}
```

As before, the program does nothing useful, but now it will execute and receive messages. For this example, the same function can be used to increment the appropriate counter simply by providing a pointer to the counter as the user data when the callback function is registered. In addition, a subtle change from the previous example has been to set the wait for table flag to false. For an xApp that is a receive only application (never sends) it is not necessary to wait for RMR to receive a table from the Route Manager.

4.2.5 Registering A Default Callback

The xApp may also register a default callback function such that the function will be invoked for any message that does not have a registered callback. If the xAPP does not register a default callback, any message which cannot be

mapped to a known callback function is silently dropped. A default callback is registered by providing a message type of xapp->DEFAULT_CALLBACK.

4.2.6 The Framework Callback Driver

The Run() function within the Xapp object is invoked to start the callback driver, and the xApp should not expect the function to return under most circumstances. The only parameter that the Run() function expects is the number of threads to start. For each thread requested, the framework will start a listener thread which will allow received messages to be processed in parallel.By supplying a value greater than one, the xApp must ensure that the callbackfunctions are thread safe as it is very likely that the same callback function will be invoked concurrently from multiple threads.

4.2.7 Sending Messages

It is very likely that most xApps will need to send messages and will not operate in "receive only" mode. Sending the message is a function of the message object itself and can take one of two forms:

- Replying to the sender of a received message
- Sending a message (routed based on the message type and subscription ID)

When replying to the sender, the message type and subscription ID are not used to determine the destination of the message; RMR ensures that the message is sent back to the origin xApp. The xApp may still need to change the message type and/or the subscription ID in the message prior to using the reply function. To provide for both situations, two reply functions are supported by the Message object as illustrated with the following prototypes.

```
bool Send_response( int mtype, int subid, int response_len,
    std::shared_ptr<unsigned char> response );

bool Send_response( int response_len,
    std::shared_ptr<unsigned char> response );
```

In the first prototype the xApp must supply the new message type and subscription ID values, where the second function uses the values which are currently set in the message. Further, the new payload contents, and length, are supplied to both functions; the framework ensures that the message is large enough to accommodate the payload, reallocating it if necessary, and copies the response into the message payload prior to sending. Should the xApp needto change either the message type, or the subscription ID, but not both, the NO_CHANGE constant can be used as illustrated below.

```
msg->Send_response( Message::NO_CHANGE, Message::NO_SUBID,
    pl_length, (unsigned char *) payload );
```

In addition to the two function prototypes for Send_response() there are two additional prototypes which allow the new payload to be supplied as a shared smart pointer. The other parameters to these functions are identical to those illustrated above, and thus are not presented here.

The Send_msg() set of functions supported by the Message object are identical to the Send_response() functions and are shown below.

```
bool Send_msg( int mtype, int subid, int payload_len,
    std::shared_ptr<unsigned char> payload );
bool Send_msg( int mtype, int subid, int payload_len,
    unsigned char* payload );
bool Send_msg( int payload_len,
    std::shared_ptr<unsigned char> payload );
bool Send_msg( int payload_len, unsigned char* payload );
```

Each send function accepts the message, copies in the payload provided, sets the message type and subscription ID (if provided), and then causes the message to be sent. The only difference between the Send_msg() and Send_response() functions is that the destination of the message is selected based on the mapping of the message type and subscription ID using the current routing table known to RMR.

4.2.8 Direct Payload Manipulation

For some applications, it might be more efficient to manipulate the payload portion of an Xapp Message in place, rather than creating it and relying on a buffer copy when the message is finally sent. To achieve this, the xApp must either use the smart pointer to the payload passed to the callback function, or retrieve one from the message using Get_payload() when working with a message outside of a callback function. Once the smart pointer is obtained, the pointer's get() function can be used to directly reference the payload (unsigned char) bytes.

When working directly with the payload, the xApp must take care not to write more than the actual payload size which can be extracted from the Message object using the Get_available_size() function.

When sending a message where the payload has been directly altered, and no extra buffer copy is needed, a NULL pointer should be passed to the Message send function. The following illustrates how the payload can be directly manipulated and returned to the sender (for simplicity, there is no error handling if the payload size of the received message isn't large enough for the response string, the response is just not sent).

```
Msg_component payload;    // smart reference
int pl_size;              // max size of payload

payload = msg->Get_payload();
pl_size = msg->Get_available_size();
if( snprintf( (char *) payload.get(), pl_size,
```

```
    "Msg Received\n" ) < pl_size ) {
  msg->Send_response( M_TYPE, SID, strlen( raw pl ), NULL );
}
```

4.2.9 Sending Multiple Responses

It is likely that the xApp will wish to send multiple responses back to the process that sent a message that triggered the callback. The callback function may invoke the Send_response() function multiple times before returning. After each call, the Message retains the necessary information to allow for a subsequent invocation to send more data. It should be noted though, that after the first call to {Send_response() the original payload will be lost; if necessary, the xApp must make a copy of the payload before the first response call is made.

4.2.10 Message Allocation

Not all xApps will be "responders," meaning that some xApps will need to send one or more messages before it can expect to receive any messages back. To accomplish this, the xApp must first allocate a message buffer, optionally initializing the payload, and then using the message's Send_msg() function to send a message out. The framework's Alloc_msg() function can be used to create a Message object with a desired payload size.

4.2.11 Framework Provided Callbacks

The framework itself may provide message handling via the driver such that the xApp might not need to implement some message processing functionality. Initially, the C++ framework will provide a default callback function to handle the RMR based health check messages. This callback function will assume that if the message was received, and the callback invoked, that all is well and will reply with an OK state. If the xApp should need to override this simplistic response, all it needs to do is to register it's own callback function for the health check message type.

5 Examples

In this section we will present actual code from example xApps.

5.1 Hello World xApp

The hello world xapp demonstrates how an xapp uses the o1, a1, and e2 interfaces. specifically, the xapp uses a "hello world sm" and implements a "hello world" a1 policy.

5.2 rmr dump xapp

The RMR dump application is an example built on top of the C++ xApp framework to both illustrate the use of the framework, and to provide a useful diagnostic tool when testing and troubleshooting xApps.

The RMR dump xApp isn't a traditional xApp inasmuch as it's goal is to listen for message types and to dump information about the messages received to the TTY much as tcpdump does for raw packet traffic. The full source code, and Makefile, are in the examples directory of the C++ repo (link?).

When invoked, the RMR dump program is given one or more message types to listen for. A callback function is registered for each, and the framework Run() function is invoked to drive the process. For each recognised message, and depending on the verbosity level supplied at program start, information about the received message(s) is written to the TTY. If the forwarding option, -f, is given on the command line, and an appropriate route table is provided, each received message is forwarded without change. This allows for the insertion of the RMR dump program into a flow, however if the ultimate receiver of a message needs to reply to that message, the reply will not reach the original sender, so RMR dump is not a complete "middle box" application.

5.2.1 Code for rmr_dump

The following is the code for this xAPP. Several sections, which provide logic unrelated to the framework, have been omitted. The full code is in the framework repository.

```
#include <stdio.h>
#include <unistd.h>
#include <atomic>

#include "ricxfcpp/xapp.hpp"

/*
  Information that the callback needs outside
  of what is given to it via params on a call
  by the framework.
*/
typedef struct {
  int    vlevel;       // verbosity level
  bool   forward;      // if true, message is forwarded
  int    stats_freq;   // header/stats after n messages
  std::atomic<long> pcount; // messages processed  std
  ::atomic<long> icount; // messages ignored
  std::atomic<int>  hdr;    // number of messages before next header
} cb_info_t;

//      ------------------------------------------------------
```

```
// Dump bytes to tty.
void dump( unsigned const char* buf, int len ) {
  // omitted for brevity
}

/*
  generate stats when the hdr count reaches 0. Only one active
  thread will ever see it be exactly 0, so this is thread safe.
*/
void stats( cb_info_t& cbi ) {
  int curv;          // current stat trigger value

  curv = cbi.hdr--;

  if( curv == 0 ) {          //stats when we reach 0
    fprintf( stdout, "ignored: %ld  processed: %ld\n",
      cbi.icount.load(), cbi.pcount.load() );
    if( cbi.vlevel > 0 ) {
      fprintf( stdout, "\n     %5s %5s %2s %5s\n",
        "MTYPE", "SUBID", "ST", "PLLEN" );
    }

    cbi.hdr = cbi.stats_freq;    // reset must be last
  }
}

// Callback registere for all msgs we are interested in
void cb1( Message& mbuf, int mtype, int subid, int len,
          Msg_component payload,  void* data ) {
  cb_info_t*  cbi;
  long total_count;

  if( ( cbi = ( cb_info_t *) data) == NULL ) {
    return;
  }

  cbi->pcount++;
  stats( *cbi );       // gen stats & header if needed

  if( cbi->vlevel > 0 ) {
      fprintf( stdout, "<RD> %-5d %-5d %02d %-5d \n",
        mtype, subid, mbuf.Get_state(), len );

    if( cbi->vlevel > 1 ) {
      dump(  payload.get(), len > 64 ? 64 : len );
    }
  }

  if( cbi->forward ) {
    // forward with no change to len or payload
    mbuf.Send_msg( Message::NO_CHANGE, NULL );
  }
}

/*
  registered as the default callback; it counts the
```

```
  messages that we aren't giving details about.
*/
void cbd( Message& mbuf, int mtype, int subid, int len,
        Msg_component payload,  void* data ) {
  cb_info_t*  cbi;

  if( ( cbi = ( cb_info_t *) data) == NULL ) {
    return;
  }

  cbi->icount++;
  stats( *cbi );

  if( cbi->forward ) {
    // forward with no change to len or payload
    mbuf.Send_msg( Message::NO_CHANGE, NULL );
  }
}

int main( int argc, char** argv ) {
  std::unique_ptr<Xapp> x;
  char* port = (char *) "4560";
  int ai = 1;          // arg processing index
  cb_info_t*  cbi;
  int    mtype;
  int    nthreads = 1;

  cbi = (cb_info_t *) malloc( sizeof( *cbi ) );
  cbi->pcount = 0;
  cbi->icount = 0;
  cbi->stats_freq = 10;

  // very simple flag parsing (no error/bounds checking)
  while( ai < argc ) {
    // code ommited for brevity
    ai++;
  }

  cbi->hdr = cbi->stats_freq;
  fprintf( stderr, "<RD> listening on port: %s\n", port );

  // create xapp, wait for route table if forwarding
  x = std::unique_ptr<Xapp>( new Xapp( port, cbi->forward ) );

  // register callback for each type on the command line
  while( ai < argc ) {
    mtype = atoi( argv[ai] );
    ai++;
    fprintf( stderr, "<RD> capturing messages for type %d\n", mtype );
    x->Add_msg_cb( mtype, cb1, cbi );
  }
  x->Add_msg_cb( x->DEFAULT_CALLBACK, cbd, cbi );   // register default cb

  fprintf( stderr, "<RD> starting driver\n" );
  x->Run( nthreads );   // return from Run() is not expected
}
```

Acknowledgments

We would like to thank the RIC team at AT&T and Nokia for the information and suggestions.

References

[1] S. Daniels, "Ric message routing." Available at https://gerrit.o-ran-sc.org/r/ric-plt/lib/rmr.

[2] Logging, "Best practices." Available at https://wiki.o-ran-sc.org/display/ORAN/Logging.

[3] R. Platform, "Logging." Available at https://gerrit.o-ran-sc.org/r/com/log.

[4] L. Walkin, "Asn1c." Available at http://lionet.info/asn1c.

[5] ASN, "Git repo." Available at https://github.com/vlm/asn1c.