O-RAN Module

Release oran-1.0

National Institute of Standards and Technology (NIST)

Apr 14, 2025

CONTENTS

CHAPTER

ONE

O-RAN MODEL DESCRIPTION

The `oran` module for ns-3 implements the classes required to model a network architecture based on the O-RAN specifications. These models include a RAN Intelligent Controller (RIC) that is functionally equivalent to O-RAN's Near-Real Time (Near-RT) RIC, and reporting modules that attach to simulation nodes and serve as communication endpoints with the RIC in a similar fashion as the E2 Terminators in O-RAN.

These models have been designed to provide the infrastructure and access to data so that developers and researchers can focus on implementing their solutions, and minimize the time and effort spent on handling interactions between models. With this in mind, all the components that contain logic that may be modified by end users have been modeled hierarchically (so that parent classes can take care of common actions and methods, and leave child models to focus on the logic itself), and at least one example is provided, to serve as reference for new models.

The RIC model uses a data repository to store all the information exchanged between the RIC and the modules, as well as to serve as a logging endpoint. This release provides an SQLite storage backend for the data repository. The database file is accessible after the simulation and can be accessed by any SQLite-compatible tool and interface.

Modeling of the reporting and communication models for the simulation nodes has been implemented using existing traces and methods, which means there is no need to modify the models provided by the ns-3 distribution to make use of the full capabilities of this module.

1.1 Features

This release of the `oran` module contains the following features:

- Near-RT RIC model, including:
  - Data access API independent of the data repository backend.
  - SQLite database repository implementation for Reports, Commands, and logging.
  - Support for Logic Modules that serve as O-RAN's `xApps`.
  - Separation of Logic Modules into `default` (only one, mandatory) and `additional` (zero to many, optional).
  - Support for addition and removal of Logic Modules during the simulation.
  - Conflict Mitigation API.
  - Logic Module and Conflict Mitigation logic logging to the data repository.
  - Periodic or report-triggered invocation of the Logic Module's algorithms.
  - Simulated processing delay for each Logic Module.
  - Integration with ONNX Runtime and PyTorch to support Machine Learning (ML)
- Periodic reporting of metrics from the simulation nodes to the Near-RT RIC.

- Periodic or event-triggered generation of reports.
- Reporting capabilities for node location (any simulation node) and LTE cell attachment (LTE UE nodes).
- Generation and execution of LTE-to-LTE handover Commands.
- Activation and deactivation for individual components and RIC.
- Simulated transmission delay between the E2 Nodes and the Near-RT RIC.
- E2 Node Keep-Alive mechanism.

CHAPTER

TWO

O-RAN MODEL ARCHITECTURE

2.1 Design

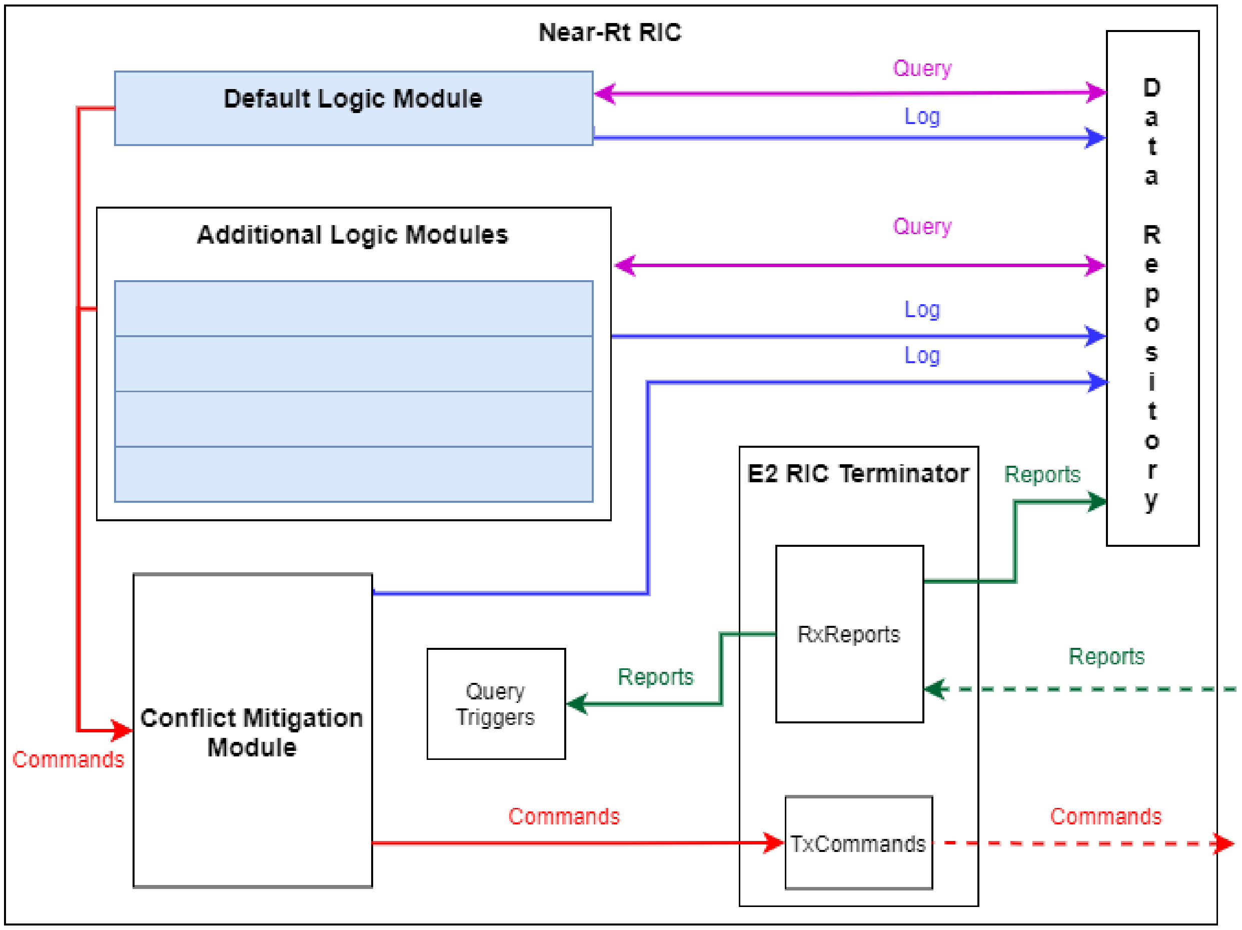
The design of the `oran` model can be divided into the components that comprise the Near-RT RIC, and the components that attach to the simulation nodes (E2 Nodes) for Reporting and Command processing.

2.1.1 Near-RT RIC Design

The Near-RT RIC is modeled as a container that houses all the individual components that provide the different functionalities of the RIC:

- Data storage.
- Logic processing modules. A `default` module must always be defined, and any number of additional modules can be deployed.
- Conflict mitigation.A
- Report triggers to start querying the logic processing modules based on report types and contents.
- Communication with the simulation nodes.

The block diagram of the Near-RT RIC model can be seen in this figure:



The Near-RT RIC is used to define and configure the specific instance of each of the components that will be used. By doing this, each of the components can be instantiated independently, and the Near-RT RIC will take care of either providing them with the pointers to the components they need to use (in the case of the data storage), or pass the information from one to another (like passing the Commands generated by the logic processing modules to the Conflict Mitigation instance). Additionally, the Near-RT RIC can activate or deactivate all the instantiated components when the RIC itself gets activated or deactivated.

The data repository model serves as the access point to the actual storage instance. This approach serves to abstract the details of the storage implementation, so the rest of the components of the Near-RT RIC do not need to know whether they are accessing a database, a text file, or any other storage structure. This abstraction also serves as a single-point definition of the data access API that will be implemented by the storage instances. This ensures that all the other components will work the same from one simulation to another, regardless of the storage mechanism used.

The operations defined by the data access API can be grouped into the following categories:

- **Report Storage**: for storing the registration requests and Reports sent by the nodes.
- **Network Status Query**: for retrieving the information stored from the nodes' Reports.
- **Logging**: for storing logic log messages and outputs generated by the logic processing and conflict mitigation modules.

The logic processing modules (abbreviated as Logic Modules, or LMs) are the components that implement the intelligence of the RIC. Each LM implements a different logic for issuing Commands to nodes depending on the status of the network retrieved from the data storage. Different LMs may try to optimize different metrics, or they may use different approaches and techniques for addressing the same specific issue. LMs use the data storage to retrieve the information reported by the network nodes, and based on the logic they implement, they generate sets of Commands for the network nodes that will help achieve the goal defined in their logic. LMs can optionally log the relevant steps of their logic to the data repository for later evaluation and / or debugging.

The LMs do not decide when their code is run. This is the decision of the Near-RT RIC that will query all the deployed LMs for the sets of Commands they want to issue. The Near-RT RIC may perform this invocation periodically at fixed intervals, or it may be triggered by a report received from an E2 Node. A query trigger is a filter that can specity a type of report and a value (or values) that, when received by the Near-RT RIC, can cause the LMs to be queried. When this happens, the timer for the periodic querying is restarted.

Each LM simulates the processing time required to run its logic using a Random Variable. When an LM is queried, the model runs its logic, generates the relevant Commands, and then waits the specified amount of time before sending a signal that indicates to the Near-RT RIC that it has finished processing and any generated commands are ready for retrieval. Meanwhile, after initiating the LM query (or queries), the Near-RT RIC starts a timer that specifies the maximum amount of time it will wait for the LMs to run their logic. If all the LMs complete their runs before the timer expires, the Near-RT RIC cancels the timer, and sends the collected Commands to the Conflict Mitigation Module. On the other hand, if the timer expires and some LMs are still in the `processing` state, then the Near-RT RIC sends the Commands that were collected from the LMs that finished on time (if any) to the Conflict Mitigation Module. If an LM finishes processing after the timer expires but before the next query starts, then the Near-RT RIC will save the Commands for the next run or discard them based on a configurable policy. Note that if the Commands are saved for the next run, it is possible that the set of Commands sent to the Conflict Mitigation Module contains Commands from a single LM generated from two different runs. It is the Conflict Mitigation module's reponsibility to handle these Commands as desired.

The Near-RT RIC must always instantiate at least one of these Logic Modules, which serves as the `default` LM for the RIC. Additional LMs can be deployed as needed as long as the `default` LM is always present. LMs can be managed dynamically during the simulation, and they can be added, removed, replaced, and reconfigured. In the case of the `default` LM, removing the existing LM must be followed by the deployment of a new instance, or the RIC will abort the simulation due to not having a `default` LM.

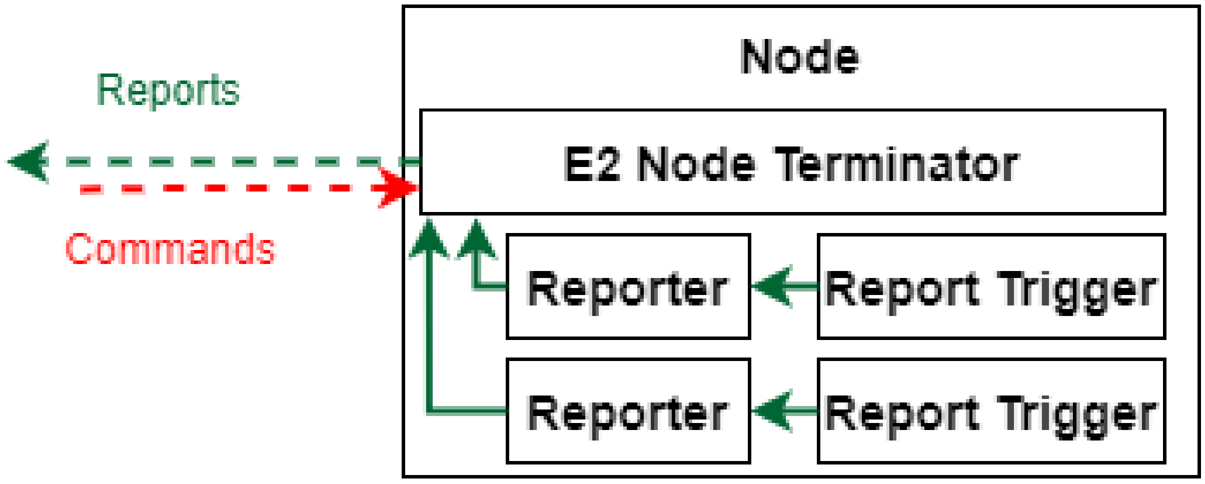
The Conflict Mitigation Module is a component that processes all the Commands generated by the deployed LMs each time the RIC invokes their logic, in order to minimize the potential conflicts between them. This module is especially important when there is more than one LM deployed, as each LM generates its own set of Commands independently, without interacting with other LMs. The logic in this module can be as complex or simple as desired, ranging from simple checks regarding the types of Commands or the nodes affected by them, to deep evaluations of the nodes and neighbors affected by the changes triggered by the Commands. The output of this process is a single set of Commands that will be sent to the relevant nodes in the network.

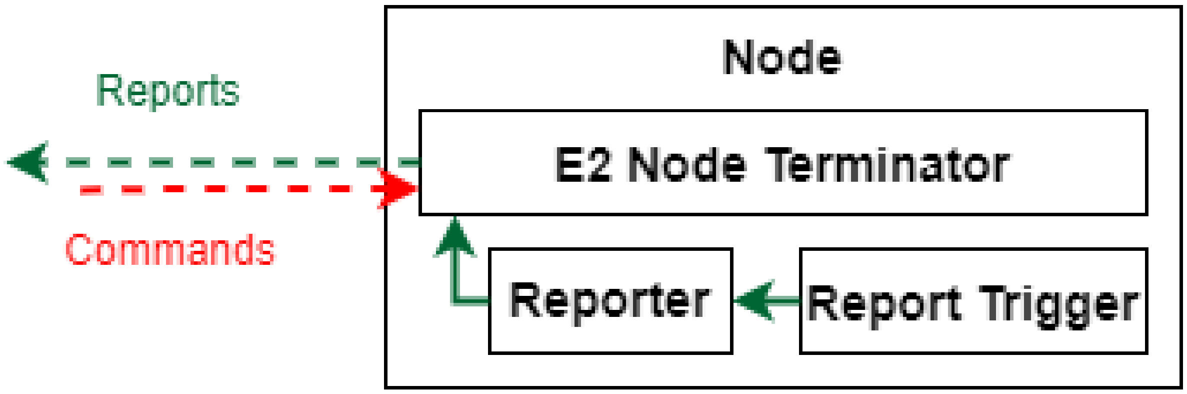
The final component of the RIC is the E2 Terminator. This module receives its name from the O-RAN architecture design, as the interface defined in those specifications is named `E2`, and the components in the RIC and nodes that use said interface are called `Terminators`. In the RIC model, the E2 Terminator is the entity that interacts with the simulation nodes, receiving the Reports sent by these nodes, and sending each of the Commands output by the Conflict Mitigation Module to the target network node. Both the received Reports and the sent Commands will be stored in the data repository, so that they can be retrieved later. For example, this information could be accessed by an LM to perform its logic or by a user in order to monitor or debug certain behavior.

All of these components can be activated and deactivated individually, or as a group. When a component is deactivated and another component tries to interact with it, the simulation is not aborted, but a `no action` approach is followed: the models will skip the actions they would normally perform and if any output is expected, empty sets or invalid values will be returned. For example, attempting to store a Report in a deactivated data storage will result in the Report being silently ignored, while passing sets of Commands to a deactivated Conflict Mitigation Module will result in all those Commands being returned right away, without any evaluation of possible conflicts.

2.1.2 Node Design

The modules that enable a simulation node to interact with the Near-RT RIC are: An E2 Terminator, and Reporters. This is depicted in the next figure:





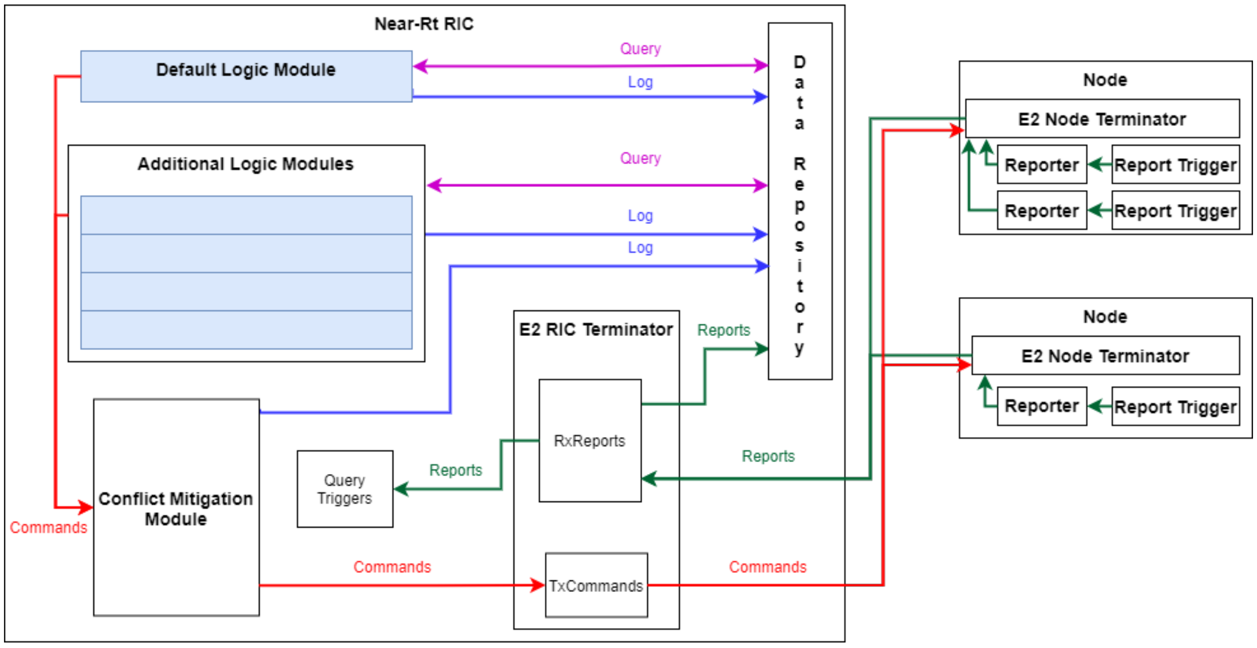
The Reporters are modules that attach to existing traces in the node, build Reports with the values being traced, and send them to the E2 Terminator in the node for transmission to the RIC. Each Reporter has an associated Report Trigger that tells the Reporter when to collect the information and generate the Report. These triggers may be periodic, or based on events or traces. Each Report includes the identity of the node, and the time at which it is generated, so even if the transmission to the RIC is delayed, the RIC will know the time at which the values in the Report were captured. When a Reporter is instantiated, it is linked to an E2 Terminator in the node, which allows the Reporter to obtain the node identity from the Terminator, and the Terminator to collect the Reports generated by the Reporter.

The E2 Terminator for the simulation nodes is analogous to the E2 Terminator in the RIC, in that it is the entity that communicates with the RIC by exchanging Reports and Commands. All of the E2 Terminators periodically send the Reports generated by the Reporters associated with it to the Near-RT RIC E2 Terminator. Additionally, the Node E2 Terminators are also in charge of receiving Commands from the RIC, and translating them into calls to the methods of the appropriate models. For example, an LTE handover Command will be translated into a call to start an X2 Handover in the associated eNB NetDevice instance. Due to the disparity of the Commands that may be accepted by each type of node, there are multiple instances of E2 Terminators, each one of them capable of processing a different set of Commands.

2.1.3 Communications Design

The communication between the Node E2 Terminators and the Near-RT RIC E2 Terminator is modeled using virtual interfaces. This means that there is no network link between these models, and instead, during the configuration, the E2 Terminators in the nodes are given a pointer to the RIC. With this pointer, the node Terminators will contact the RIC Terminator when they are activated, and this allows the RIC to keep track of all the node Terminators that are sending Reports and accepting Commands. Transmission delays for both Reports and Commands are simulated using Random Variables: one Random Variable in the E2 Terminator of the Near-RT RIC generates the delays for the Commands being sent to the E2 Nodes, and a Random Variable in each E2 Node Terminator generates the delays for the Reports being transmitted from the E2 Nodes to the Near-RT RIC.

The next figure shows all the components including the communication exchanges.

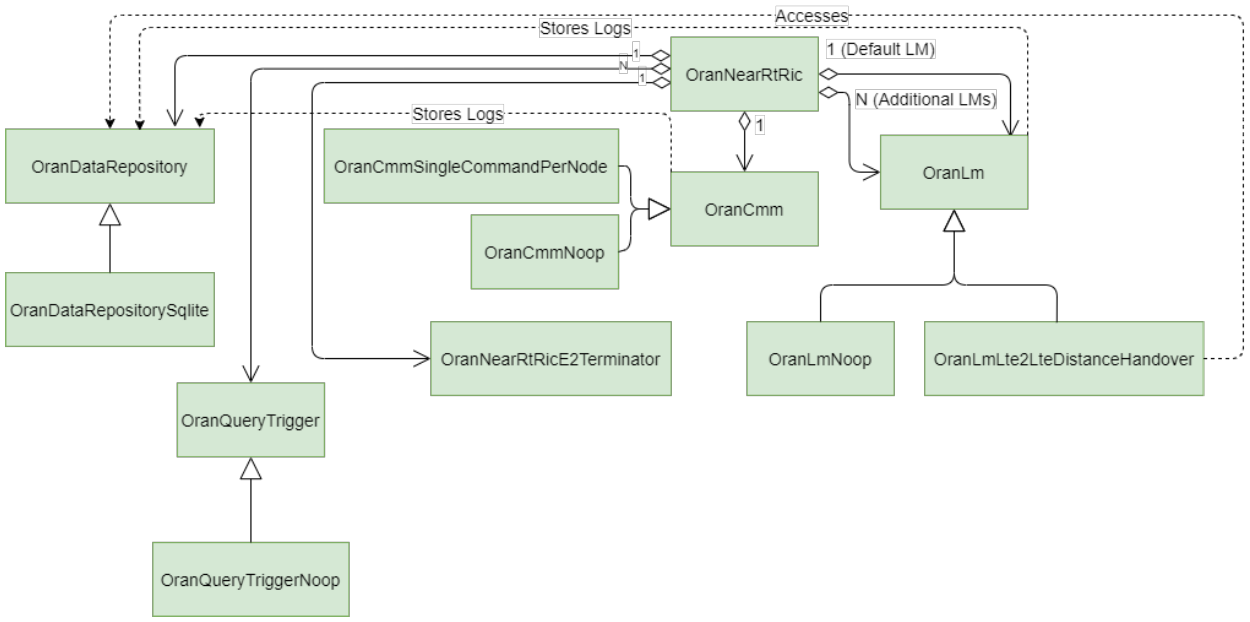


2.2 Class Organization

In this Section we will present the classes that have been used to model the O-RAN architecture according to the design previously presented. We will look at the classes in three groups: The RIC classes, the node classes, and the information exchange classes.

2.2.1 Near-RT RIC

As we can see in the next figure



the class diagram can be easily mapped to the block diagrams presented earlier. Each functional module has been modeled with a parent class, that defines the API and interactions with other classes, and inheriting from the parent class are one or more child classes that provide specific implementations for each module.

The Data Repository class (`OranDataRepository`) defines the methods used by other components in the RIC to store and retrieve information in the RIC storage. An implementation of the storage module that uses SQLite as the backend (`OranDataRepositorySqlite`) inherits from this base class and implements all the data access methods by building up SQL commands and executing them against the database.

The Logic Module classes follow a similar principle, although the parent class (`OranLm`) actually implements methods that will be the same for all the implementations of LMs. For example, the methods used for activating and deactivating the module, retrieving the name, and logging messages, are all implemented in the parent class. This allows the instances to implement only the constructor, destructor, and logic method, as every other task is already taken care of. LMs make use of the Data Repository for retrieving information about the state of the network, and storing log messages and the generated Commands. In this release there are two specific instances of LMs: a 'No Operation' LM that does nothing (`OranLmNoop`), but serves to instantiate an LM when we must provide one, and an 'LTE handover' LM that issues Commands to handover an LTE UE from one LTE cell to another based on the distance from the LTE UE to the eNBs (`OranLmLte2LteDistanceHandover`).

A similar approach is taken for the Conflict Mitigation Module: the parent class (`OranCmm`) provides the implementation for all the common methods, and the specific implementations only need to implement their specific logic. The Conflict Mitigation modules access the Data Repository to log messages about their logic. Two implementations are provided in this release: a 'No Operation' implementation (`OranCmmNoop`), that does nothing, and a 'Single Command' implementation (`OranCmmSingleCommandPerNode`) that makes sure that in a single set we do not have more than one Command affecting the same node (if more than one Command affects the same node, the Command issued by the default LM takes precedence; otherwise, the first processed Command takes precedence).

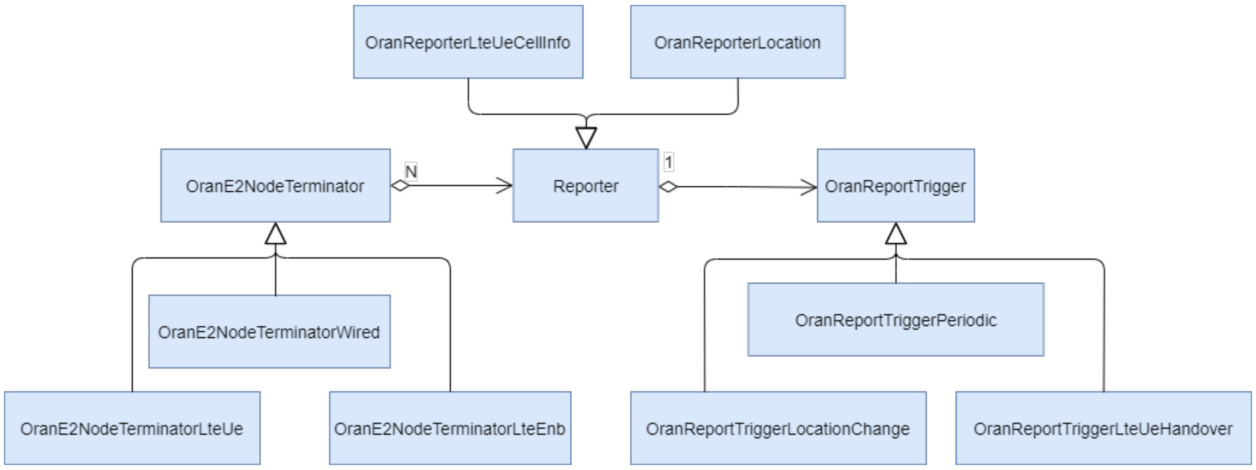
The Near-RT RIC also contains a collection (implemented as a C++ map) of Query Triggers that are used to start querying the LMs as soon as Reports with certain criteria reach the Near-RT RIC. The parent class for these Query Triggers is `OranQueryTrigger`, and currently the only specific implementation is a No-Operation Trigger (`OranQueryTriggerNoop`) that never initiates the LM querying. The examples provided show how one can implemenet a custom Query Trigger based, for example, on Location Reports.

The Near-RT RIC E2 Terminator class (`OranNearRtRicE2Terminator`) is the only functional module that does not follow the same hierarchical design, as there are no different implementations to be provided. Therefore, the Near-RT RIC E2 Terminator class provides all the functionality needed for receiving Reports and node registration requests, storing these Reports and updating registration information in the Data Repository, receiving sets of Commands to send to nodes, logging these Commands in the Data Repository, and transmitting them to the appropriate nodes.

Finally, the Near-RT RIC class (`OranNearRtRic`) is the model that serves as a container for all the other functional models. A Near-RT RIC will contain an instance of the Data Repository, an LM instance marked as `default`, a vector of other additional LM instances, a Conflict Mitigation Module instance, and a Near-RT RIC E2 Terminator. Whenever one of these modules needs to access another, they will do so through the Near-RT RIC. This allows for dynamic replacement of the instances used during the simulation without having to update a significant number of references. Also, as the Near-RT RIC needs to ensure it keeps valid references to all the deployed instances, it will check that the references are valid whenever the RIC is activated and when it needs to invoke a method in the instances (for example, invoking the logic in the LMs). If any reference is found to be invalid or NULL, the simulation will be aborted.

2.2.2 O-RAN Nodes

The classes used to model the node components of the architecture are shown in this figure


OranReporterLteUeCellInfo
OranReporterLocation
OranE2NodeTerminator
N
Reporter
1
OranReportTrigger
OranE2NodeTerminatorWired
OranReportTriggerPeriodic
OranE2NodeTerminatorLteUe
OranE2NodeTerminatorLteEnb
OranReportTriggerLocationChange
OranReportTriggerLteUeHandover

We can see how both components (the E2 Terminator and the Reporters) have been implemented following the same hierarchical principle as the Near-RT RIC components. Therefore, we have a parent Reporter class (`OranReporter`) that provides the implementation of the activation, deactivation, and generation of Reports to the E2 Terminator. This class is inherited by:

- The Location Reporter (`OranReporterLocation`), which can be attached to any ns-3 node, periodically retrieves the node's position through the node's Mobility Model.
- The LTE UE Cell Information Reporter (`OranReporterLteUeCellInfo`), which attaches to LTE UE nodes, and uses the LTE UE NetDevice to obtain the ID of the cell the UE is attached to.

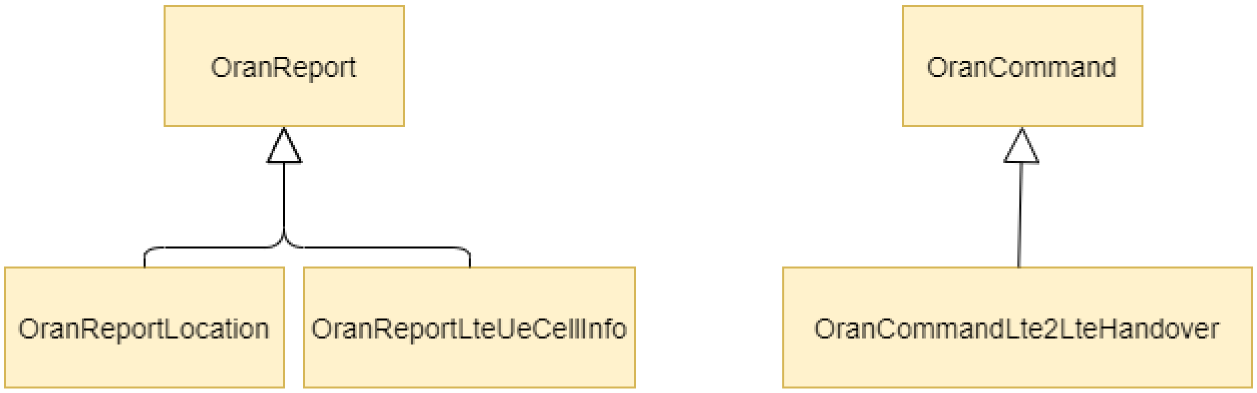
Each Reporter class must have one Report Trigger (parent class `OranReportTrigger`) that tells the Reporter when to collect the information. This is so that the Reporter knows how to get the information, and the Report Trigger knows when to get the information. The current release of the code includes a periodic Report Trigger (`OranReportTriggerPeriodic`), and two event-based Report Triggers: the `OranReportTriggerLocationChange` which is based on location events, and the `OranReportTriggerLteUeHandover`, which is based on successul LTE handover events in a UE.

Similarly, the parent Node E2 Terminator class (`OranE2NodeTerminator`) provides the implementation for activating and deactivating, attaching to a node, adding Reporter instances, and sending periodic registration requests and Reports to the Near-RT RIC. These operations are the same for all specific instances of the Terminator. Where these instances will differ is in the Commands that they can process. Currently, implementations are provided of E2 Terminators for wired nodes (`OranE2NodeTerminatorWired`), LTE UEs (`OranE2NodeTerminatorLteUe`), and LTE eNBs (`OranE2NodeTerminatorLteEnb`).

Regarding the E2 Node periodic registration process, it is important to note that the Near-RT RIC performs periodic checks to identify E2 Nodes that have not updated their registration recently. If the last registration for an E2 Node in the Near-RT RIC's data repository is older than a configured threshold, the E2 Node will be considered deregistered, its Reports will be ignored, and it will not be issued any commands. It is therefore important to configure the registration timing adequately.

2.2.3 Information Elements

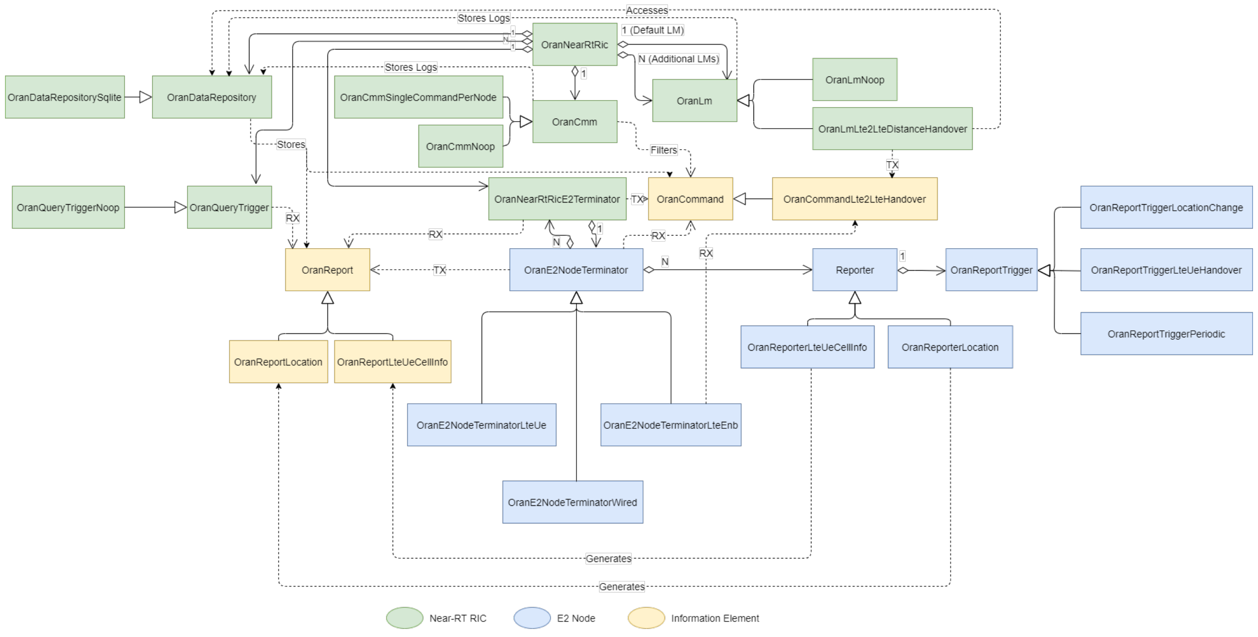
The final group of classes are the ones used for the information exchange between the RIC and the nodes. They represent the Reports (parent class `OranReport`) that the nodes generate and the RIC stores in the Data Repository, and the Commands (parent class `OranCommand`) issued by the RIC's LMs and processed by the E2 Terminators in the nodes. These are shown in this figure:



The current release provides a Location Report (`OranReportLocation`), which contains node locality information, and an LTE UE Cell Information Report (`OranReportLteUeCellInfo`), which contains information about the cell to which the issuing LTE UE is attached.

Conversely, there is only one instance of Command distributed in this release, and that is the 'LTE to LTE Handover' Command (`OranCommandLte2LteHandover`). This Command instructs an eNB to handover one of the UEs attached to it (identified by its Radio Network Temporary Identifier (RNTI)) to another eNB. This Command must be processed by an LTE eNB E2 Terminator.

The next figure shows the interaction of all classes, as well as the Reporter instances that generate each Report instance, and the LTE eNB E2 Terminator being the one Terminator capable of processing the only available Command.



2.2.4 Utilities and Helpers

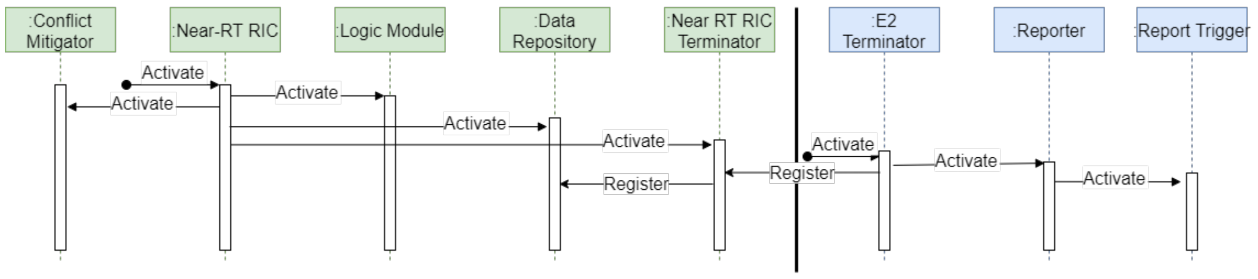
In addition to all the classes presented, the current distribution includes a Helper (`OranHelper`) that eases the code burden on the scenarios for configuring and deploying the individual models, and a Node E2 Terminator Container (`OranE2NodeTerminatorContainer`) that allows the activation and deactivation of multiple Node E2 Terminators with a single Command.

2.3 O-RAN Model Operation

This Section documents the sequence of events generated by the most common operations of the O-RAN models during a simulation: activation of the models, reporting from the nodes to the RIC, LM logic invocation in the RIC, and Command issuing from the RIC to the nodes.

2.3.1 Activation

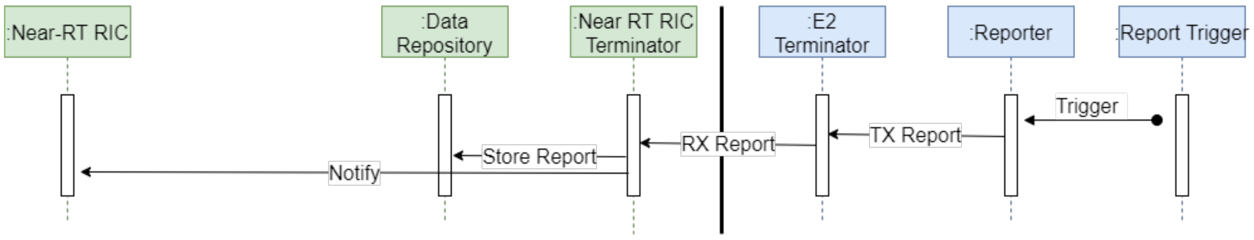
As shown in the next figure, when the Near-RT RIC is activated using its public API, the activation Command is propagated to the rest of the components: Data Repository, Logic Modules (default and additional), Conflict Mitigation Module, and E2 Terminator.



On the nodes' side, the activation of the E2 Terminator will trigger the activation of all the Reporters and Report Triggers attached to it, and it also triggers a registration request to the RIC. If the RIC has already been activated, the registration request will be recorded in the Data Repository. If the RIC is not active at this time, the registration request will be ignored, and periodic retransmissions of the request will be scheduled in the Node E2 Terminator.

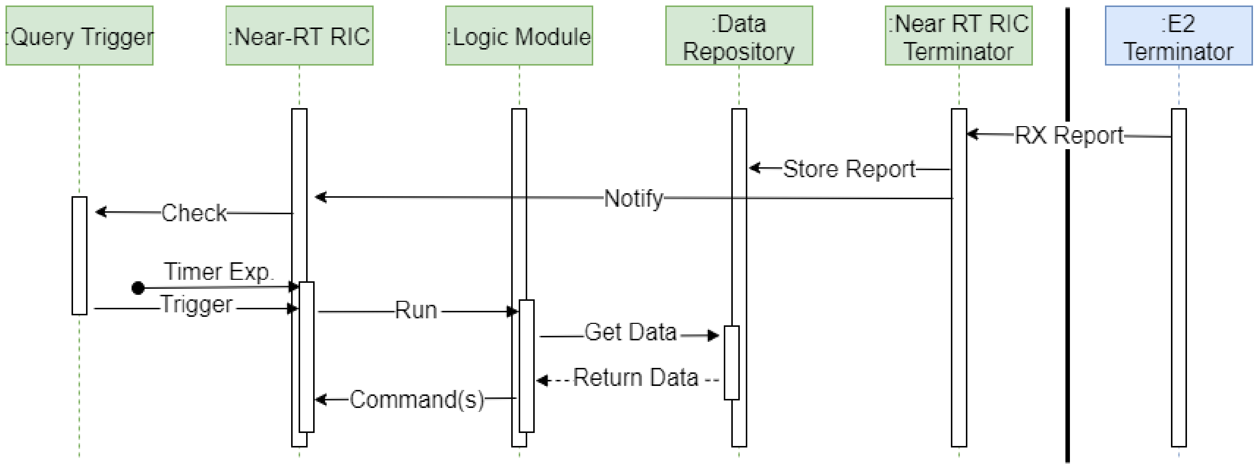
2.3.2 Report Generation

Once the RIC and a Node E2 Terminator have been activated, and a registration request has been successfully recorded in the Data Repository, the Report Triggers in the node will signal when the Reporters should generate Reports and send them to the Node E2 Terminator. In this Node E2 Terminator, a timer will periodically send the collected Reports to the RIC. When the RIC's E2 Terminator receives these Reports, they will be stored in the Data Repository, as shown in the next figure, and the Near-RT RIC will be notified so that the Query Triggers can be checked against the received Reports.



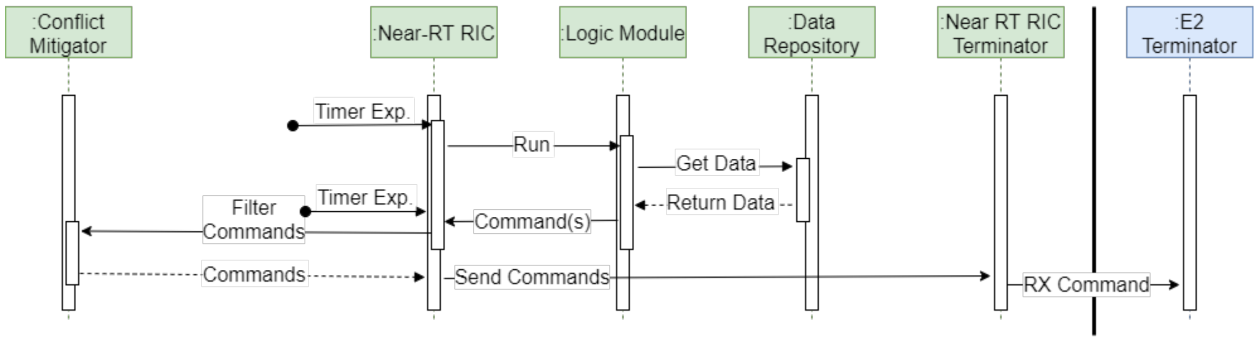
2.3.3 RIC Logic Execution

The next figure shows how, the run of the logic of the LMs in the Near-RT RIC can be triggered by a periodic timer, or by a Query Trigger matching a received Report. In either case, the RIC will request to all the deployed LMs (default and additional) to run their logic and generate Commands for the network nodes. For each LM to process this request, it will retrieve information from the Data Storage about the network it is interested in (e.g. the positions of all the nodes in the last 10 seconds, the average traffic loss rate in the last 5 seconds, or the list of all the cells that each UE has been attached to), and use this information to decide, based on the logic and goals implemented, if any change needs to be made to the network, and if so, the appropriate Commands are issued. These commands will be passed to the Near-RT RIC once a processing timer expires in the LM.



2.3.4 Issuing of Commands

After all the LMs have run their logic and generated their sets of Commands, or a maximum processing timer has expired in the Near-RT RIC, all collected Commands are sent to the Conflict Mitigation Module, which filters all of the Commands according to the logic implemented, and issues a single set of Commands ready to be sent to the nodes. This set is passed to the Near-RT RIC E2 Terminator which will log the individual Commands in the Data Storage, and then send each one of them to the appropriate E2 Terminator in the nodes. This sequence is show in the next figure.


:Conflict Mitigator
:Near-RT RIC
:Logic Module
:Data Repository
:Near RT RIC Terminator
:E2 Terminator
Timer Exp.
Run
Get Data
Return Data
Timer Exp.
Filter Commands
Command(s)
Commands
Send Commands
RX Command

CHAPTER

THREE

O-RAN USAGE

In this Chapter we will review how to use the provided O-RAN models in an scenario, as well as how to implement our own models. The working examples provided with the release demonstrate all the required operations and configurations needed to use the models.

3.1 Deployment in Scenario

The operations required to use the O-RAN models in a simulation, are:

- Configuration and instantiation of the Near-RT RIC and all its modules, and
- Configuration and instantiation of the Node E2 Terminators and Reporters.

Instantiating and configuring the Near-RT RIC to the point where it is ready to be used in the simulation requires the previous instantiation of all the modules that make up the RIC. First, we need to instantiate a Data Repository. This can be done by simply creating an object of the appropriate class, and configuring its attributes. The next listing shows how this is done for the SQLite-backed Data Repository, which needs to know the name of the file we will use for the database, and optionally, the trace sink for the status report of all the SQL queries run against the database (method QueryRcSink in the listing):

```
void
QueryRcSink (std::string query, std::string args, int rc)
{
  std::cout << Simulator::Now ().GetSeconds () <<
    " Query " << ((rc == SQLITE_OK || rc == SQLITE_DONE) ? "OK" : "ERROR") <<
    "(" << rc << "): \"" << query << "\"";

  if (! args.empty ())
    {
      std::cout << " (" << args << ")";
    }
  std::cout << std::endl;
}
// ....
Ptr<OranDataRepository> dataRepository = CreateObject<OranDataRepositorySqlite> ();
dataRepository->SetAttribute ("DatabaseFile", StringValue (dbFileName));
dataRepository->TraceConnectWithoutContext ("QueryRc", MakeCallback (&QueryRcSink));
```

Once the Data Repository has been configured, the next step is to instantiate and configure the Logic Modules. We need a pointer to the Near-RT RIC to complete the configuration of these modules, so we will create the pointer to the RIC without any further configuration, and then we will focus on the LMs, as shown in the next listing. All the LMs, regardless of their type, must be provided with a pointer to the Near-RT RIC, a Random Variable that will provide the processing time (in seconds) when the LM is queried, and optionally (but recommended), a name. This name will be used in the logging messages. Finally, LMs can be configured to be verbose, which makes the LMs log information to the Data Repository each time it is run:

```
Ptr<OranNearRtRic> nearRtRic = CreateObject<OranNearRtRic> ();
// ....
Ptr<OranLm> myLm = CreateObject<OranLmNoop> ();
myLm->SetName ("MyNoopLm");
myLm->SetAttribute ("NearRtRic", PointerValue (nearRtRic));
myLm->SetAttribute ("Verbose", BooleanValue (true));
myLm->SetAttribute ("ProcessingDelayRv",
    StringValue ("ns3::ConstantRandomVariable[Constant=0.005]"));
```

The same process and attributes to instantiate and configure the LMs applies to the Conflict Mitigation Modules:

```
Ptr<OranNearRtRic> nearRtRic = CreateObject<OranNearRtRic> ();
// ....
Ptr<OranCmm> cmm = CreateObject<OranCmmSingleCommandPerNode> ();
cmm->SetName ("MyConflictMitigationModule");
cmm->SetAttribute ("NearRtRic", PointerValue (nearRtRic));
cmm->SetAttribute ("Verbose", BooleanValue (true));
```

The last module we need to instantiate is the E2 Terminator for the RIC. The configuration of this module is very simple, as it just needs to be provided pointers to the Near-RT RIC, the Data Repository, and the Random Variable that will simulate the transmission delay (in seconds) for the issued Commands:

```
Ptr<OranNearRtRic> nearRtRic = CreateObject<OranNearRtRic> ();
Ptr<OranDataRepository> dataRepository = CreateObject<OranDataRepositorySqlite> ();
// ....
Ptr<OranNearRtRicE2Terminator> nearRtRicE2Terminator = CreateObject
↪<OranNearRtRicE2Terminator> ();
nearRtRicE2Terminator->SetAttribute ("NearRtRic", PointerValue (nearRtRic));
nearRtRicE2Terminator->SetAttribute ("DataRepository", PointerValue (dataRepository));
nearRtRicE2Terminator->SetAttribute ("TransmissionDelayRv",
    StringValue ("ns3::ConstantRandomVariable[Constant=0.001]"));
```

All of these modules can also be activated and deactivated individually, but in most cases we will want to activate and deactivate the whole RIC at once, which can be done once we configure the object created earlier.

The next block shows the code necessary to configure the Near-RT RIC. Lines 1 to 8 serve as a reminder that we need to have all the modules instantiated as previously shown. After that, lines 10 to 16 show how to link these models to the RIC. Special attention should be given to lines 10 and 11, where we can see that the Default LM is set using an attribute (as this LM is mandatory), but the additional LMs are added to the RIC using the method `AddLogicModule`. Lines 18 to 20 configure the maximum time for an E2 Node to refresh its registration, and the Random Variable that will define how often the Near-RT RIC will check for nodes that have not registered recently. Lines 22 and 23 configure the maximum time the Near-RT RIC will wait for the LMs to run their logic, and the policy that will define what to do with the reports generated by LMs that take longer than the maximum time (in this case, the policy is 'DROP', so they will be discarded). Line 25 adds a Query Trigger to the Near-RT RIC.

Finally, line 27 shows the scheduling of the activation and start of operations 1 second into the simulation. At this time, when the `Start` method is invoked in the RIC, all the modules linked to the RIC will also be activated, and the timer to periodically run the LMs will also start:

```
Ptr<OranNearRtRic> nearRtRic = CreateObject<OranNearRtRic> ();
// ...
Ptr<OranDataRepository> dataRepository = CreateObject<OranDataRepositorySqlite> ();
```

(continues on next page)

(continued from previous page)

```
Ptr<OranLm> defaultLm = CreateObject<OranNoop> ();
Ptr<OranLm> otherLm = CreateObject<OranLmLte2LteDistanceHandover> ();
Ptr<OranCmm> cmm = CreateObject<OranCmmNoop> ();
Ptr<OranNearRtRicE2Terminator> nearRtRicE2Terminator = CreateObject
↪<OranNearRtRicE2Terminator> ();
Ptr<OranQueryTrigger> myQueryTrigger = CreateObject<OranQueryTriggerNoop> ();
// ...
nearRtRic->SetAttribute ("DefaultLogicModule", PointerValue (defaultLm));
nearRtRic->AddLogicModule (otherLm);
nearRtRic->SetAttribute ("LmQueryInterval", TimeValue (Seconds (5)));

nearRtRic->SetAttribute ("E2Terminator", PointerValue (nearRtRicE2Terminator));
nearRtRic->SetAttribute ("DataRepository", PointerValue (dataRepository));
nearRtRic->SetAttribute ("ConflictMitigationModule", PointerValue (cmm));

nearRtRic->SetAttribute ("E2NodeInactivityThreshold", TimeValue (Seconds (2)));
nearRtRic->SetAttribute ("E2NodeInactivityIntervalRv",
    StringValue ("ns3::ConstantRandomVariable[Constant=2]"));

nearRtRic->SetAttribute ("LmQueryMaxWaitTime", TimeValue (Seconds (0.5)));
nearRtRic->SetAttribute ("LmQueryLateCommandPolicy", EnumValue (OranNearRtRic::DROP));

nearRtRic->AddQueryTrigger ("MyNoopQueryTrigger", myQueryTrigger);

Simulator::Schedule (Seconds (1), &OranNearRtRic::Start, nearRtRic);
```

Once we have finished configuring the RIC, we can start deploying E2 Terminators in the simulation nodes. The next listing shows that the Node E2 Terminators themselves need to be provided a pointer to the Near-RT RIC (note that the Near-RT RIC must be instantiated before configuring the Node E2 Terminator; however, the listing does not include the code for instantiating the Near-RT RIC for clarity purposes; previous listings demonstrate how to instantiate all the required models), as well as the random variables that will be used for triggering periodic registration events, periodic transmission of Reports to the Near-RT RIC (lines 6 to 8), and the delay for the transmission of said Reports (line 9). Once these attributes have been configured, and the Reporters in this node created, these Reporters must be added to the E2 Terminator using the `AddReporter` method, as shown on line 11. Additionally we need to attach the Terminator to the simulation node, so IDs can be retrieved for registration purposes, and the listing shows how to do this on line 12. Finally, we need to schedule a time for the activation of the Terminator and all the attached Reporters, as is done on line 14 of the listing:

```
Ptr<Node> myWiredNode = CreateObject<Node> ();
Ptr<OranNearRtRic> nearRtRic = CreateObject<OranNearRtRic> ();
Ptr<OranE2NodeTerminatorWired> wiredNodeTerminator = CreateObject
↪<OranE2NodeTerminatorWired> ();
Ptr<OranReporterLocation> locationReporter = CreateObject<OranReporterLocation> ();
// ....
wiredNodeTerminator->SetAttribute ("NearRtRic", PointerValue (nearRtRic));
wiredNodeTerminator->SetAttribute ("RegistrationIntervalRv", StringValue (
↪"ns3::ConstantRandomVariable[Constant=5]"));
wiredNodeTerminator->SetAttribute ("SendIntervalRv", StringValue (
↪"ns3::ConstantRandomVariable[Constant=1]"));
wiredNodeTerminator->SetAttribute ("TransmissionDelayRv", StringValue (
↪"ns3::ConstantRandomVariable[Constant=0.001]");

```

(continues on next page)

(continued from previous page)

```
wiredNodeTerminator->AddReporter (locationReporter);
wiredNodeTerminator->Attach (myWiredNode);

Simulator::Schedule (Seconds (2), &OranE2NodeTerminatorWired::Activate,␣
↪wiredNodeTerminator);
```

After the Node E2 Terminator has been configured, the next step is to instantiate and configure any Reporter that will be operating from that node. The next listing shows the instantiation and configuration of an LTE UE Cell Information Reporter. All Reporters need to be provided a pointer to the Node E2 Terminator that they will be reporting to (line 4) and a Report Trigger that will indicate when to generate these reports (lines 5 and 6). As mentioned earlier, once the Reporter is fully configured, we need to add it to the Node E2 Terminator (line 8):

```
Ptr<OranE2NodeTerminatorWired> lteUeTerminator = CreateObject<OranE2NodeTerminatorLteUe>␣
↪();
// ....
Ptr<OranReporterLocation> lteCellInfoReporter = CreateObject<OranReporterLteUeCellInfo>␣
↪();
lteCellInfoReporter->SetAttribute ("Terminator", PointerValue (lteUeTerminator));
lteCellInfoReporter->SetAttribute ("Trigger",
    StringValue ("ns3::OranReportTriggerLteUeHandover[InitialReport=true]"));

lteUeTerminator->AddReporter (lteCellInfoReporter);
```

3.1.1 Helper

The previous listings show the code that is necessary to include in a scenario in order to use the O-RAN models. However, it is possible to use the helper provided with the distribution to simplify and reduce the code.

The next listing shows all the code necessary to instantiate and configure the Near-RT RIC using the `OranHelper`. This helper provides methods for each component, so that a single method can be used to specify the type of the component instance, along with any attributes that we want to configure, as shown on lines 11 (Data Repository), 12 (Default Logic Module), 13 (Additional Logic Module), 14 (Conflict Mitigation Module), and 15 (a Query Trigger). Near-RT RIC attributes can be configured through attributes of the helper, as on lines 4 to 9. When all the modules have been configured, the method `CreateNearRtRic` will instantiate all the modules, associate them with the Near-RT RIC, and return a pointer to the Near-RT RIC instance (line 17). The helper can also be used to activate and deactivate the RIC with the methods `ActivateAndStartNearRtRic` and `DeactivateAndStopNearRtRic`:

```
Ptr<OranNearRtRic> nearRtRic = nullptr;
Ptr<OranHelper> oranHelper = CreateObject<OranHelper> ();

oranHelper->SetAttribute ("Verbose", BooleanValue (false));
oranHelper->SetAttribute ("LmQueryInterval", TimeValue (Seconds (1)));
oranHelper->SetAttribute ("E2NodeInactivityThreshold", TimeValue (Seconds (2)));
oranHelper->SetAttribute ("E2NodeInactivityIntervalRv", StringValue (
↪"ns3::ConstantRandomVariable[Constant=2]"));
oranHelper->SetAttribute ("LmQueryMaxWaitTime", TimeValue (Seconds (0.5)));
oranHelper->SetAttribute ("LmQueryLateCommandPolicy", EnumValue (OranNearRtRic::DROP));

oranHelper->SetDataRepository ("ns3::OranDataRepositorySqlite", "DatabaseFile",␣
↪StringValue (dbFileName));
oranHelper->SetDefaultLogicModule ("ns3::OranLmNoop", "Name", StringValue ("defaultLm"));
oranHelper->AddLogicModule ("ns3::OranLmLte2LteDistanceHandover", "Verbose",␣
↪BooleanValue (true), "Name", StringValue ("MyAdditionalLm"));
```

(continues on next page)

(continued from previous page)

```
oranHelper->SetConflictMitigationModule ("ns3::OranCmmSingleCommandPerNode", "Verbose",␣
↪BooleanValue (true));
oranHelper->AddQueryTrigger ("Crossover", "ns3::OranQueryTriggerCustom", "CustomCallback
↪", CallbackValue (MakeCallback(&CustomLmQueryTriggerCallback)));

nearRtRic = oranHelper->CreateNearRtRic ();
```

The configuration and instantiation of the nodes using the helper must be done in groups of nodes that will share the same type of Node E2 Terminator and Reporters. The next listing shows the configuration of Terminators for a group of LTE UE nodes (lines 11 through 20) and a group of wired nodes (lines 23 through 30). Each block of code has the same structure: First, the E2 Node Terminator is configured by defining its type and any attribute that we may need (lines 11-13 and 23-25). Then the method `AddReporter` is called as many times as different types of Reporters will be deployed on these nodes, specifying the type of Reporter and the attributes needed to configure it (lines 15-18 configure two Reporters for the LTE UEs, and lines 27-28 configure one Reporter for the wired nodes). Finally, the method `DeployTerminators` is invoked with the pointer to the Near-RT RIC and the NodeContainer with the nodes where the Terminators and Reporters will be deployed (lines 20 and 30), and an `OranE2NodeTerminatorContainer` that has all the Node E2 Terminators is returned:

```
NodeContainer ueNodes, wiredNodes;

Ptr<OranNearRtRic> nearRtRic = nullptr;
OranE2NodeTerminatorContainer e2NodeTermsUes, e2NodeTermsWired;
Ptr<OranHelper> oranHelper = CreateObject<OranHelper> ();
oranHelper->SetAttribute ("Verbose", BooleanValue (true));
// ....
nearRtRic = oranHelper->CreateNearRtRic ();
// ....
// LTE UEs get Location Reporter and Cell ID Reporter
oranHelper->SetE2NodeTerminator ("ns3::OranE2NodeTerminatorLteUe",
    "RegistrationIntervalRv", StringValue ("ns3::ConstantRandomVariable[Constant=5]"),
    "SendIntervalRv", StringValue ("ns3::ConstantRandomVariable[Constant=1]"));

oranHelper->AddReporter ("ns3::OranReporterLocation",
    "Trigger", StringValue ("ns3::OranReportTriggerPeriodic"));
oranHelper->AddReporter ("ns3::OranReporterLteUeCellInfo",
    "Trigger", StringValue ("ns3::OranReportTriggerLteUeHandover[InitialReport=true]"));

e2NodeTermsUes.Add (oranHelper->DeployTerminators (nearRtRic, ueNodes));

// Wired nodes get only Location Reporter
oranHelper->SetE2NodeTerminator ("ns3::OranE2NodeTerminatorWired",
    "RegistrationIntervalRv", StringValue ("ns3::ConstantRandomVariable[Constant=5]"),
    "SendIntervalRv", StringValue ("ns3::ConstantRandomVariable[Constant=1]"));

oranHelper->AddReporter ("ns3::OranReporterLocation",
    "Trigger", StringValue ("ns3::OranReportTriggerPeriodic"));

e2NodeTermsWired.Add (oranHelper->DeployTerminators (nearRtRic, wiredNodes));
```

3.2 Modeling New Implementations

In this Section we will present shells of custom modules (Logic Module, Query Trigger, Conflict Mitigation Module, Reporter, Report Trigger, and Node E2 Terminator) that show that custom modules only need to focus on the logic they want to implement, as most of the overhead work is being taken care of by the parent classes. These shell classes can also be used as the starting point for the development of new models.

3.2.1 Logic Module

The next listing shows the shell of a custom Logic Module. Besides the constructor (line 24), destructor (line 32), and `GetTypeId` methods (line 13), we see that a custom LM only needs to implement the public method `Run`, that returns a vector of Reports. In this `Run` method the basic workflow is described in the comments on lines 48-52, and it consists of retrieving information from the Data Storage (using the pointer to the Near-RT RIC to get access to the instance), run the algorithm that we want this LM to implement, and generate the Commands that are deemed necessary. These Commands are stored in a vector and this vector is returned:

```
#include <ns3/log.h>
#include <ns3/abort.h>
#include <ns3/simulator.h>

#include "oran-data-repository.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("MyLm");

NS_OBJECT_ENSURE_REGISTERED (MyLm);

TypeId
MyLm::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MyLm")
    .SetParent<OranLm> ()
    .AddConstructor<MyLm> ()
  ;

  return tid;
}

MyLm::MyLm (void)
  : OranLm ()
{
  NS_LOG_FUNCTION (this);

  m_name = "MyLm";
}

MyLm::~MyLm (void)
{
  NS_LOG_FUNCTION (this);
}

std::vector<Ptr<OranCommand> >
```

(continues on next page)

(continued from previous page)

```
MyLm::Run (void)
{
  NS_LOG_FUNCTION (this);

  std::vector<Ptr<OranCommand> > commands;

  if (m_active)
    {
      NS_ABORT_MSG_IF (m_nearRtRic == nullptr, "Attempting to run LM (" + m_name + ")␣
↪with NULL Near-RT RIC");

      // Step 1: Retrieve relevant data from the Data Repository

      // Step 2: Run our algorithm

      // Step 3: Create the needed Commands and put them in the 'commands' vector
    }

  // Return the commands.
  return commands;
}

} // namespace ns3
```

3.2.2 Query Trigger

The next listing shows the shell of a custom Query Trigger. The core of a Query Trigger is the method `QueryLms`, that receives a Report and based on the type and contents of the report, returns a boolean value that, if true, will initiate the querying of the LMs:

```
#include <ns3/log.h>
#include <ns3/abort.h>

#include "oran-report.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("MyQueryTrigger");

NS_OBJECT_ENSURE_REGISTERED (MyQueryTrigger);

TypeId
MyQueryTrigger::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MyQueryTrigger")
    .SetParent<OranQueryTrigger> ()
    .AddConstructor<MyQueryTrigger> ()
  ;

  return tid;
}

```

(continues on next page)

(continued from previous page)

```
MyQueryTrigger::MyQueryTrigger (void)
  : OranReporter ()
{
  NS_LOG_FUNCTION (this);
}

MyQueryTrigger::~MyQueryTrigger (void)
{
  NS_LOG_FUNCTION (this);
}

bool
MyQueryTrigger::QueryLms (Ptr<OranReport> report)
{
  NS_LOG_FUNCTION (this);

  bool queryLms = false;

  // Check type and contents of report, and, if needed, change the
  // value of queryLms

  return queryLms;
}

} // namespace ns3
```

3.2.3 Conflict Mitigation

Similar to the Logic Modules, a custom Conflict Mitigation Module (like the one shown in the next listing) only needs to implement the constructor (line 24), destructor (line 32), Dispose (line 38), and `GetTypeId` methods (line 13), and the method that houses the logic specific to this CMM: the `Filter` method (line 45):

```
#include <ns3/log.h>
#include <ns3/abort.h>

#include "oran-command.h"
#include "oran-near-rt-ric.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("MyCmm");

NS_OBJECT_ENSURE_REGISTERED (MyCmm);

TypeId
MyCmm::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MyCmm")
    .SetParent<OranCmm> ()
    .AddConstructor<MyCmm> ()
  ;
```

(continues on next page)

(continued from previous page)

```
  return tid;
}

MyCmm::MyCmm (void)
  : OranCmm ()
{
  NS_LOG_FUNCTION (this);

  m_name = "My Custom CMM";
}

MyCmm::~MyCmm (void)
{
  NS_LOG_FUNCTION (this);
}

void
MyCmm::DoDispose (void)
{
  NS_LOG_FUNCTION (this);

  OranCmm::DoDispose ();
}

std::vector<Ptr<OranCommand> >
MyCmm::Filter (
  std::map<std::tuple<std::string, bool>,
  std::vector<Ptr<OranCommand> > > inputCommands)
{
  NS_LOG_FUNCTION (this);

  NS_ABORT_MSG_IF (m_nearRtRic == nullptr, "Attempting to run Conflict Mitigation Module␣
↪with NULL Near-RT RIC");

  std::vector<Ptr<OranCommand> > commands;

  if (m_active)
    {
      // TODO: Decide which commands we really want to transmit, and add them to the
↪'commands' vector
    }
  else
    {
      // This module is not active. Just copy the same set of commands as output
      for (auto commandSet : inputCommands)
        {
          commands.insert (commands.end (), commandSet.second.begin (), commandSet.
↪second.end ());
        }
    }

  return commands;
```

(continues on next page)

(continued from previous page)

```
}

} // namespace ns3
```

The `Filter` method receives a map in which the key is a tuple with the name of the LM that generated the associated Commands, and a boolean flag indicating if this is the Default LM of the RIC. The value associated with each tuple is a vector of all the Commands generated by that LM.

With all the Commands sorted this way, the CMM can implement the desired logic to filter them, as described in the comment on line 58. The Commands that pass this filter shall be stored in a vector, which will be returned by the method.

3.2.4 Node Reporter

The next listing presents the shell for custom Reporters. Once again, we need to implement the constructor (line 23), destructor (line 29), and `GetTypeId` methods (line 12). For Reporters, the method that implements the specific data capture for this instance is the `GenerateReports` method (line 34). This method is expected to return a vector of Reports. The comments on lines 44-48 describe the usual workflow for generating these Reports, namely capture the information from the node, create Reports with the appropriate values filled in, and store these Reports in the vector that will be returned:

```
#include <ns3/log.h>
#include <ns3/abort.h>

#include "oran-report.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("MyReporter");

NS_OBJECT_ENSURE_REGISTERED (MyReporter);

TypeId
MyReporter::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MyReporter")
    .SetParent<OranReporter> ()
    .AddConstructor<MyReporter> ()
  ;

  return tid;
}

MyReporter::MyReporter (void)
  : OranReporter ()
{
  NS_LOG_FUNCTION (this);
}

MyReporter::~MyReporter (void)
{
  NS_LOG_FUNCTION (this);
}
```

(continues on next page)

(continued from previous page)

```
std::vector<Ptr<OranReport> >
MyReporter::GenerateReports (void)
{
  NS_LOG_FUNCTION (this);

  std::vector<Ptr<OranReport> > reports;
  if (m_active)
    {
      NS_ABORT_MSG_IF (m_terminator == nullptr, "Attempting to generate reports in␣
↪reporter with NULL E2 Terminator");

      // Step 1: Capture the information needed from the node methods or traces

      // Step 2: Instantiate and populate the values of a Report of the appropriate type

      // Step 3: Add the report to the 'reports' vector
    }
  return reports;
}

} // namespace ns3
```

3.2.5 Report Trigger

The next listing shows a shell code for custom Report Triggers. The main component of a Report Trigger is a method that will monitor a trace, and based on that trace, it will invoke the method `TriggerReports`. This method is shown in the listing on lines 74 to 81. In order for this method to work, we use the `Activate` method to connect the trace of interest to our method (lines 34 to 46), and the `Deactivate` method to disconnect the trace (lines 48 to 59, and 83 to 90):

```
#include <ns3/log.h>
#include <ns3/simulator.h>

#include "oran-reporter.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("MyReportTrigger");

NS_OBJECT_ENSURE_REGISTERED (MyReportTrigger);

TypeId
MyReportTrigger::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MyReportTrigger")
    .SetParent<OranReportTrigger> ()
    .AddConstructor<MyReportTrigger> ()
  ;

  return tid;
}
```

(continues on next page)

(continued from previous page)

```
MyReportTrigger::MyReportTrigger (void)
  : OranReportTrigger ()
{
  NS_LOG_FUNCTION (this);
}

MyReportTrigger::~MyReportTrigger (void)
{
  NS_LOG_FUNCTION (this);
}

void
MyReportTrigger::Activate (Ptr<OranReporter> reporter)
{
  NS_LOG_FUNCTION (this << reporter);

  if (! m_active)
    {
      traceofinterest->TraceConnectWithoutContext ("TraceName",
        MakeCallback (&MyReportTrigger::CheckTriggeredTrace, this));
    }

  OranReportTrigger::Activate (reporter);
}

void
MyReportTrigger::Deactivate (void)
{
  NS_LOG_FUNCTION (this);

  if (m_active)
    {
      DisconnectSink ();
    }

  OranReportTrigger::Deactivate ();
}

void
MyReportTrigger::DoDispose (void)
{
  NS_LOG_FUNCTION (this);

  if (m_active)
    {
      DisconnectTrace ();
    }

  OranReportTrigger::DoDispose ();
}
```

(continues on next page)

(continued from previous page)

```
void
MyReportTrigger::CheckTriggeredTrace (...params...)
{
  NS_LOG_FUNCTION (this);

  NS_LOG_LOGIC ("Triggering report");
  TriggerReport ();
}

void
MyReportTrigger::DisconnectTrace (void)
{
  NS_LOG_FUNCTION (this);

  traceofinterest->TraceDisconnectWithoutContext ("TraceName",
    MakeCallback (&MyReportTrigger::CheckTriggeredTrace, this));
}

} // namespace ns3
```

3.2.6 Node E2 Terminator

Finally, the next listing shows how, similarly to the other custom modules, custom Node E2 Terminators need to implement the constructor (line 20), destructor (line 26), and `GetTypeId` methods (line 9). Additionally, the `GetNodeType` method must be implemented to indicate what type of node this E2 Node Terminator can be attached to:

```
#include <ns3/log.h>

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("MyE2NodeTerminator");

NS_OBJECT_ENSURE_REGISTERED (MyE2NodeTerminator);

TypeId
MyE2NodeTerminator::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MyE2NodeTerminator")
    .SetParent<OranE2NodeTerminator> ()
    .AddConstructor<MyE2NodeTerminator> ()
  ;

  return tid;
}

MyE2NodeTerminator::MyE2NodeTerminator (void)
  : OranE2NodeTerminator ()
{
  NS_LOG_FUNCTION (this);
}

MyE2NodeTerminator::~MyE2NodeTerminator (void)
```

(continues on next page)

(continued from previous page)

```
{
  NS_LOG_FUNCTION (this);
}

OranNearRtRic::NodeType
MyE2NodeTerminator::GetNodeType (void) const
{
  NS_LOG_FUNCTION (this);

  return OranNearRtRic::WIRED;
}

void
MyE2NodeTerminator::ReceiveCommand (Ptr<OranCommand> command)
{
  NS_LOG_FUNCTION (this << command);

  if (m_active)
    {
      // TODO: Filter the commands we can process using the TypeId
      // TODO: Add processing of the receive commands.
    }
}

} // namespace ns3
```

The differentiating aspect between Node E2 Terminators is the type of Commands they can process. This specificity is implemented in the `ReceiveCommand` method (line 40). This method, when the Terminator is active, checks that it knows how to process the Command (by checking the `TypeId` of the Command), and then proceeds to interpret and act as directed by the Command.

CHAPTER

FOUR

O-RAN EXAMPLES AND TESTS

This Section presents the examples distributed with the release, and gives a quick overview of the scenario in each one.

4.1 Random Walk Example

The first and simplest example is the Random Walk Example, distributed in the example file `oran-random-walk-example.cc`. This example presents a very simple topology in which ns-3 nodes move randomly and periodically reports its position to the Near-RT RIC. This scenario shows how the RIC can be configured to collect and gather information from the simulation, without any logic processing or Command issuing taking place.

This example provides two methods for optionally printing the position of a node to the terminal when its direction is altered (method `CourseChange`), and the database queries used to store and access data in the RIC along with their result (method `QueryRcSink`). These two methods can be enabled or disabled with the command line switch `--verbose`.

The example uses the `OranHelper` to reduce the amount of code needed to configure and deploy working models of the full O-RAN architecture. This helper is instantiated on line 107, and starting at line 116 the RIC is being configured: First a data repository that uses the SQLite backend, and then an instance of the 'No Operation' Logic Module to be used as the default LM (line 118), and an instance of the 'No Operation' Conflict Mitigation Module (line 119). This configuration ensures that no actual processing will take place, and no Commands will be issued.

The Node E2 Terminator is configured starting at line 124, with a Location Reporter using a periodic Report Trigger attached to it (line 128). These models are instantiated and deployed on line 131.

Finally, lines 140 and 141 activate and start the operation of the RIC and the Node E2 Terminator.

4.2 LTE to LTE Distance Handover Example

The LTE to LTE Distance Handover Example, distributed in the example file `oran-lte-2-lte-distance-handover-example.cc`, is a more complex scenario that shows how to configure and deploy each of the models manually, without using the `OranHelper`. This scenario consists of 2 LTE eNBs and 1 LTE UE that moves back and forth between both eNBs. The UE is initially attached to the closest eNB, but as it moves closer to the other eNB, the Logic Module in the RIC will issue a handover Command and the UE will be attached to the other eNB.

This example provides two methods for optionally printing a message to the terminal when a handover has been successfully completed (method `NotifyHandoverEndOkEnb`), and the database queries used to store and access data in the RIC along with their result (method `QueryRcSink`). These two methods can be enabled or disabled with the command line switch `--verbose`. Additionally, the method `ReverseVelocity` is invoked periodically to reverse the direction in which the UE is moving.

The configuration of the O-RAN models begins at line 178. Firstly, the components of the RIC are instantiated and configured:

- Data Repository, instantiated on line 182 and configured on lines 188 through 192.
- Default Logic Module, instantiated as `OranLmLte2LteDistanceHandover` on line 183 and configured on lines 194 to 196.
- Conflict Mitigation Module, instantiated as an `OranCmmNoop` on line 184 and configured on lines 198 and 199.
- E2 Terminator, instantiated on line 186 and configured on lines 201 through 203.

After all these modules have been configured, the Near-RT RIC is configured with references to them (shown on lines 205 through 213), and the activation and start of operations is scheduled for RIC to happen one second into the simulation, on line 200.

The next step is to configure the Terminators and Reporters for the nodes. The loop on lines 217 to 240 configures the LTE UE nodes in the simulation with an LTE UE E2 Node Terminator (instantiated on line 221, and configured on lines 229 to 232), a Location Reporter with a periodic trigger (instantiated on line 219, configured on lines 223 and 224), and a Cell Attachment Reporter with a handover event trigger (instantiated on line 220, configured on lines 226 and 227). The Reporters are added to the Terminator on lines 234 and 235, and the Terminator is attached to the node on line 237. Finally, the Terminator is scheduled to start working on line 239.

Similarly the loop on lines 242 to 260 configures the LTE eNB nodes almost the same, with the only differences being that the Terminator instance used is the LTE eNB E2 Node Terminator, and that only the Location Reporter is configured for these nodes.

4.3 LTE to LTE Handover With Helper Example

The LTE to LTE Distance Handover Helper Example, distributed in the example file `oran-lte-2-lte-distance-handover-helper-example.cc`, is functionally the same scenario as the previous example. However, in this scenario the `OranHelper` is used to configure and deploy the models.

The Helper is instantiated on line 181. After that, the Helper is configured with several attributes that will be passed to the models as they are instantiated (lines 183 to 189), the model names and parameters of the RIC modules that we want to use (lines 197 to 201), and the RIC is instantiated (line 203).

The instantiation and configuration of the models for the LTE UE nodes is achieved with lines 206 to 217, and the LTE eNB nodes are setup on lines 220 to 228.

Finally, the activation and start of all the models is scheduled on lines 237 to 239.

This example is used as the base for others, that will show additional functionality with the same topology.

4.4 LTE to LTE Handover With LM Processing Delay Example

The LTE to LTE Distance Handover With LM Processing Delay Example, distributed in the example file `oran-lte-2-lte-distance-handover-lm-processing-delay-example.cc`, is functionally the same scenario as the one in the previous example. However, in this scenario the LM is configured with a processing delay.

The processing delay is defined with a Random Variable. By default the scenario uses a Normal Random Variable with a mean of 5 ms and a variance of 0.031 ms, but this can be overriden using the `processing-delay-rv` command line parameter.

The relevant lines for configuring the processing delay for LMs are:

- Lines 208 and 209, where LMs are created with the `ProcesingDelayRv` attribute set to some non-default value.
- Line 197, where the maximum time to wait for LMs while processing is configured (this value will be passed to the Near-RT RIC).
- Line 198, where the Near-RT RIC policy for handling LMs that do not complete the calculations in time is defined.

4.5 LTE to LTE Handover With LM Query Trigger Example

The LTE to LTE Distance Handover With LM Query Trigger Example, distributed in the example file `oran-lte-2-lte-distance-handover-lm-query-trigger-example.cc`, is functionally the same scenario as the one in the previous example. However, in this scenario the Near-RT RIC is configured with a custom Query Trigger (provided in the scenario) that may initiate the LM querying process as soon as Reports with certain characteristics are received by the Near-RT RIC.

In this scenario the function that will be used to analyze the Reports received by the Near-RT RIC and decide whether to start the LM Querying process is defined on lines 93 to 127 (function `CustomLmQueryTriggerCallback`. This function:

- Receives a Report.
- **Runs its logic, which in this case is:**
  - Check if the Report is a Location Report. If so,
  - Check if the neighbor LTE eNB is closer. If so,
  - Prepare to return `true`.
- Return `true` if the LM querying process should be started right away.

Once the function is defined, the only other step required for deploying the custom Query Trigger is to add it to the Helper before instantiating the Near-RT RIC, as done on lines 234 and 235. When adding this Query Trigger to the Helper we specify a name, the Query Trigger class that will be used (`OranQueryTriggerCustom` in this case), and the function to use for evaluation, specified as a Callback.

4.6 Keep-Alive Example

The Keep-Alive Example, distributed in the example file `oran-keep-alive-example.cc`, showcases how the Keep-Alive mechanism works. This example uses a single node moving in a straight line and periodically reporting its position to the Near-RT RIC. However, not all of these Reports will be accepted by the Near-RT RIC, because the configuration of the timing for sending Registration messages in the node means that the node will frequently be marked as 'inactive' by the Near-RT RIC.

The configuration parameters relevant in this example are:

- **In the Near-RT RIC:**
  - The maximum time since a previous registration before marking an E2 Node as 'Deregistered'. Configured on line 83 to be 1 second.
  - The Random Variable for the inteval between periodic checks of E2 Nodes to see if any of them should be marked as 'Deregistered'. Configured on line 84 to be a constant interval of 0.5 seconds.
- **In the E2 Node Terminator:**
  - The Random Variable for the inteval between sending periodic Registration messages to the Near-RT RIC. Configured on line 101 to be a uniform interval between 2 and 3 seconds.

Note that the values used for the Random Variables and timings have been chosen to showcase the workings of the Keep-Alive mechanism and the effects of E2 Nodes being deregisted from the Near-RT RIC. These values are not recommended for use in actual scenarios.

Running this scenario with WARNING logs enabled for the E2 Node Terminator like this:

```
NS_LOG="OranE2NodeTerminator=prefix_time|warn" ./waf --run oran-keep-alive-example
```

will show how many of the Registration messages were sent too late.

4.7 Data Repository Example

The Data Repository Example, distributed in the example file `oran-data-repository-example.cc`, showcases how the Data Repository API can be used to store and retrieve information. This example performs all these operations from the scenario for simplicity, but the same methods can be used by any model in the simulation that can access the RIC. This will be important when developing custom Logic Modules, and can also be used in scenarios for debugging, testing and validation.

4.8 LTE to LTE ML Handover Example

The LTE to LTE ML Handover Example, distributed in the example file `oran-lte-2-lte-ml-handover-example.cc`, is a scenario that showcases how pretrained ONNX and PyTorch ML Models can be used to initiate handovers based on location and packet loss data. It consists of four UEs and two eNBs, where UE 1 and UE 4 are configured to move only within coverage of eNB 1 or eNB 2, respectively, while UE 2 and UE 3 move around in an area where the coverage of eNB 1 and eNB 2 overlap. As the simulation progresses with UEs moving and receiving data, the distances of all four UEs as well as the recorded packet loss for each UE are fed to an ML model that returns a desired configuration that indicates which eNB UE 2 and UE 3 should be attached to to minimize the overall packet loss. The models that we provide are for demonstrate purposes only and have not been thoroughly developed. It should also be noted that "saved_trained_model_pytorch.onnx" is the same trained model as "saved_trained_model_pytorch.pt" only it has been exported to the ONNX format.

Note that in order to run this example using the flag, `--use-onnx-lm`, the ONNX libraires must be found during the configuration of ns-3, and it is assumed that the ML model file `saved_trained_model_pytorch.onnx` has been copied from the example directory to the working directory. In order to run this example using the flag, `--use-torch-lm` the PyTorch libraires must be found during the configuration of ns-3, and it is assumed that the ML model file `saved_trained_model_pytorch.pt` has been copied from the example directory to the working directory.

The configuration of the O-RAN models begins at line 305. The `OranLmLte2LteOnnxHandover` LM is instantiated and configured on lines 320 to 324, while the `OranLmLte2LteTorchHandover` LM is instantiated and configured on lines 325 to 329. These LMs use a pretrained ML model that takes UE distance and application loss as an input and then outputs a desired configuration that the LM can then use to determine if any handovers need to take place.

There is a script included in the examples folder called, `oran-lte-2-lte-ml-handover-example-generate-training-data.sh` that can be used to generate data using this same example to create and train an ML model. This script essentially runs a simulation for each possible combination of UE-to-cell configuration, and then parses the data to determine which confiugration provides the lowest average packet loss. Using this information, the script then provides traning data that can be used by the PyTorch classifier defined in `oran-lte-2-lte-ml-handover-example-classifier.py`, which takes the outputs from the script as input. This input essentially tells the ML model which configuration will provide the lowest average packet loss for the next one second, given the inputs we described earlier. After runing the script to generate the training data and feeding that training data to the python classifier, a new `saved_trained_model_pytorch.pt` should exist that can now be used by the `OranLmLte2LteTorchHandover` LM.

4.9 Tests

The Test Suite provided with the models includes a single test (`OranTestCaseMobility1`). This test creates a single node which starts to move two seconds into the simulation and stops moving 12 seconds into the simulation. During all this time the node moves with a constant velocity of (2, 2, 0). The scenario uses the `OranHelper` to deploy a RIC with 'No Operation' LMs and CMM, so no attempts to modify the topology is made by the RIC, and a Location Reporter is attached to the node.

After 14 seconds of simulation the Data Storage in the RIC is queried to retrieve the first and last positions reported by the node, and they are compared with the pre-computed values to verify their correctness.

CHAPTER

FIVE

O-RAN ADAPTATIVE SQLITE DESCIPTION

5.1 New funcionaties

Modifications were made to the ns3-oran extension to implement support for managing arbitrary reports. These changes only add new functionality to the simulator without impacting existing features. As a result, the added tools are backward-compatible, ensuring that all code written using the standard extension remains fully functional.

The database system, `OranAdaptativeSqlite`, dynamically manages incoming reports based on their type:

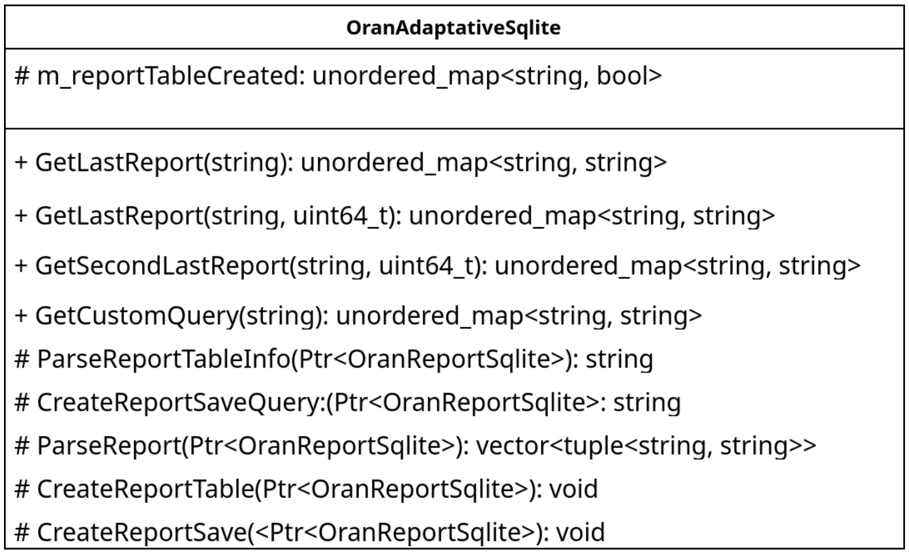
- For default reports (predefined in the simulator), data is added to the SQLite database as usual.
- For custom reports (inheriting from `OranReportSqlite`): The database checks if a report of this type has been received before by verifying the table name provided by the report's GetTableName method. If the table does not exist: A new table is created using the schema defined by the report's GetTableInfo method. This table includes columns for the sender node's ID and the report's timestamp. If the table exists: The report's data is added to the existing table.

The OranAdaptativeSqlite class provides four public methods for querying data:

- `GetLastReport`:
  - One version takes a table name (`string`) and returns the most recent entry from that table.
  - A second version takes a table name (`string`) and a node ID (`uint64_t`), returning the last report of that type from the specified node.
- `GetSecondLastReport`: Functions like `GetLastReport` but retrieves the second-most recent report, useful for analyzing changes between consecutive reports.
- `GetCustomQuery`: Accepts any SQLite query (as a `string`) and returns the results.

All methods return data in an `unordered_map` format, where keys represent report fields and values hold the corresponding data. If a method fails (e.g., invalid query, missing table, or node ID), it throws an exception.

The figure below show the new implemented methods of `OranAdaptativeSqlite`. Further details of this class can be found in the Doxygen documentation



OranAdaptativeSqlite

m_reportTableCreated: unordered_map<string, bool>

+ GetLastReport(string): unordered_map<string, string>
+ GetLastReport(string, uint64_t): unordered_map<string, string>
+ GetSecondLastReport(string, uint64_t): unordered_map<string, string>
+ GetCustomQuery(string): unordered_map<string, string>
ParseReportTableInfo(Ptr<OranReportSqlite>): string
CreateReportSaveQuery:(Ptr<OranReportSqlite>: string
ParseReport(Ptr<OranReportSqlite>): vector<tuple<string, string>>
CreateReportTable(Ptr<OranReportSqlite>): void
CreateReportSave(<Ptr<OranReportSqlite>): void

5.2 Creating new reports

To create a custom report, the user must specialize the `OranReportSqlite` class. This class has three public virtual methods that must be overridden:

- `GetTableInfo`: Returns a vector of string tuples.
- `GetTableName`: Returns a string.
- `ToString`: Returns a string.

The `GetTableName` method must return the name of the table to be used in the database. This name allows xApps to access data inserted into the table. The `GetTableInfo` method returns information about the field names and data types for each metric included in the report. These details are used to create new tables for custom reports. Finally, the `ToString` method should return a string containing the report name followed by the data to be inserted into the database, enclosed in parentheses and separated by semicolons.

The listing below demonstrates how to override the methods required to create a custom report containing the IPv4 address of a network node, as used in one of the examples:

```
OranReportUeIpv4::OranReportUeIpv4() : OranReportSqlite()
{
    m_tableInfo.emplace_back("ipv4", "TEXT");
}

std::string OranReportUeIpv4::ToString() const
{
    std::stringstream ss;
    Time time = GetTime();

    ss << "OranReportIpv4(" << "nodeid=" << GetReporterE2NodeId() << ";time=" << time.
↪GetTimeStep() << ";ipv4=" << "\"" << m_ipv4 << "\"" << ")";

    return ss.str();
}

std::vector<std::tuple<std::string, std::string>> OranReportUeIpv4::GetTableInfo()
```

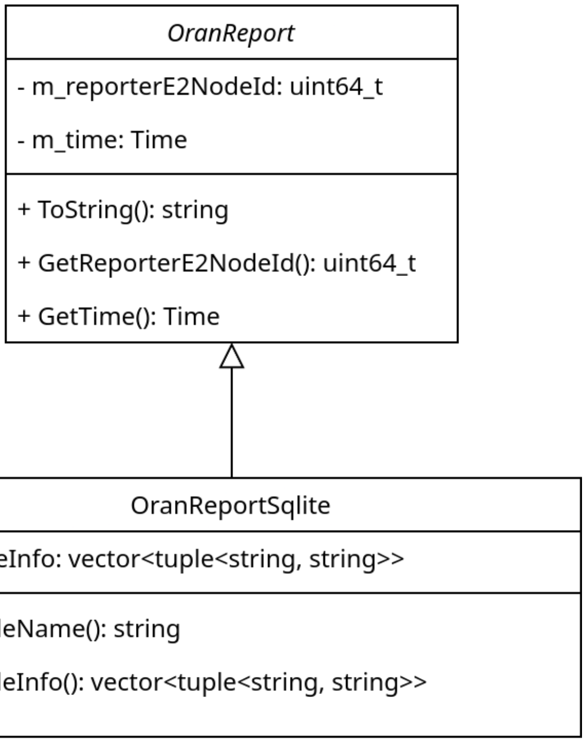
(continues on next page)

(continued from previous page)

```
{
    return m_tableInfo;
}

std::string OranReportUeIpv4::GetTableName()
{
    return "UeIpv4";
}
```

The figure below shows a simplified UML diagram of the `OranReportSqlite` class. Further details can be found in the Doxygen documentation.


OranReport
- m_reporterE2NodeId: uint64_t
- m_time: Time
+ ToString(): string
+ GetReporterE2NodeId(): uint64_t
+ GetTime(): Time
OranReportSqlite
# m_tableInfo: vector<tuple<string, string>>
+ GetTableName(): string
+ GetTableInfo(): vector<tuple<string, string>>

CHAPTER

SIX

O-RAN ADAPTATIVE SQLITE EXAMPLES

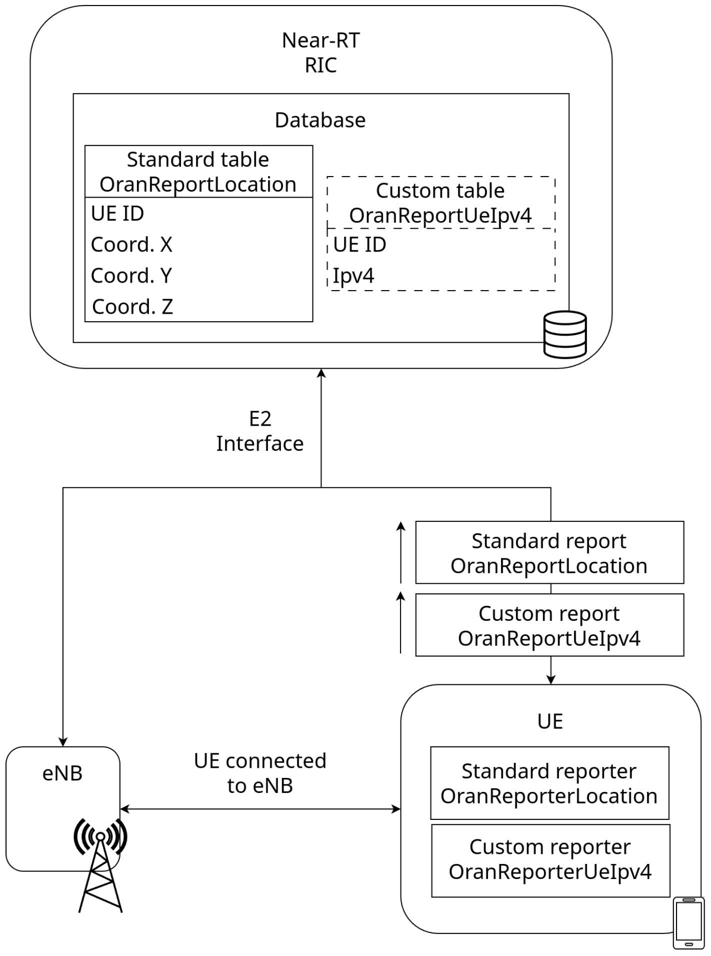
6.1 Simple-db-example

This example aims to illustrate the use of the new data storage system in conjunction with pre-existing reports. It is a simple example simulating the Near-RT RIC, involving only one user device and one eNB. Both the user device and the eNB are nodes in the O-RAN network. The example consists of five files, namely `oran-report-ue-ipv4.cc`, `oran-report-ue-ipv4.h`, `oran-reporter-ue-ipv4.cc`, `oran-reporter-ue-ipv4.h` and `simple-db-usage.cc`.

The files `oran-reporter-ue-ipv4.cc` and `oran-reporter-ue-ipv4.h` implement the class containing the logic for a reporter. The purpose of the reporter is to generate a report. This class is added to a user device and is used whenever the device prepares a report to be sent to the Near-RT RIC. The function of this class is to create a report of type OranReportUeIpv4 and write data to it. The complete implementation code for these two files can also be found in Appendix A.

The file `simple-db-usage.cc` is the main file and consists of simulating an LTE network with one eNB and one user device. A Near-RT RIC is also created, and both the user device and the eNB are E2 nodes of this controller. A constant position is defined for the eNB, and a constant-speed mobility model is added to the UE. A location reporter (which generates reports already implemented by default in ns3-oran) and an IPv4 reporter (which generates custom reports, as described earlier) are also added to the UE. The simulation lasts 10 seconds, and periodically, every second, reports are sent from the UE to the Near-RT RIC. When running the simulation, all SQLite database queries are displayed on the screen.

The figure below illustrates the example. The figure shows an eNB, a UE, and the Near-RT RIC. The UE is connected to the eNB, and both the UE and the eNB are connected to the Near-RT RIC via the E2 interface. The UE has two reporters that generate periodic location and IPv4 address reports. The reports are sent from the UE to the Near-RT RIC via the E2 interface (indicated by the arrow next to the report). The Near-RT RIC contains a database with two tables. The OranReportLocation table, shown with a solid line, corresponds to the location reports implemented by default in ns3-oran. This table is initialized by default at the start of the simulation. The OranReportUeIpv4 table, shown with a dotted line, corresponds to the custom report. This table, as well as any custom report tables, is dynamically generated during the simulation when the first report of that type arrives at the Near-RT RIC.



6.2 RL-handover-example

The second example provided with the tool aims to demonstrate how the implemented database model can be used to train a reinforcement learning model. This example implements a logic module that trains a reinforcement learning model using Proximal Policy Optimization (PPO) to manage handover decisions, with the goal of minimizing packet loss rates. For demonstration purposes, the example simulates handover management in a scenario with one UE and two eNBs.

This simulation example additionally uses the ns3-ai module, which enables communication between ns-3 simulation scripts and an external Python process. With ns3-ai, a Gymnasium environment can be defined in C++ within ns-3 simulation scripts. This environment is then accessible from a Python script, allowing the ns-3 simulation to act as a training scenario for reinforcement learning.

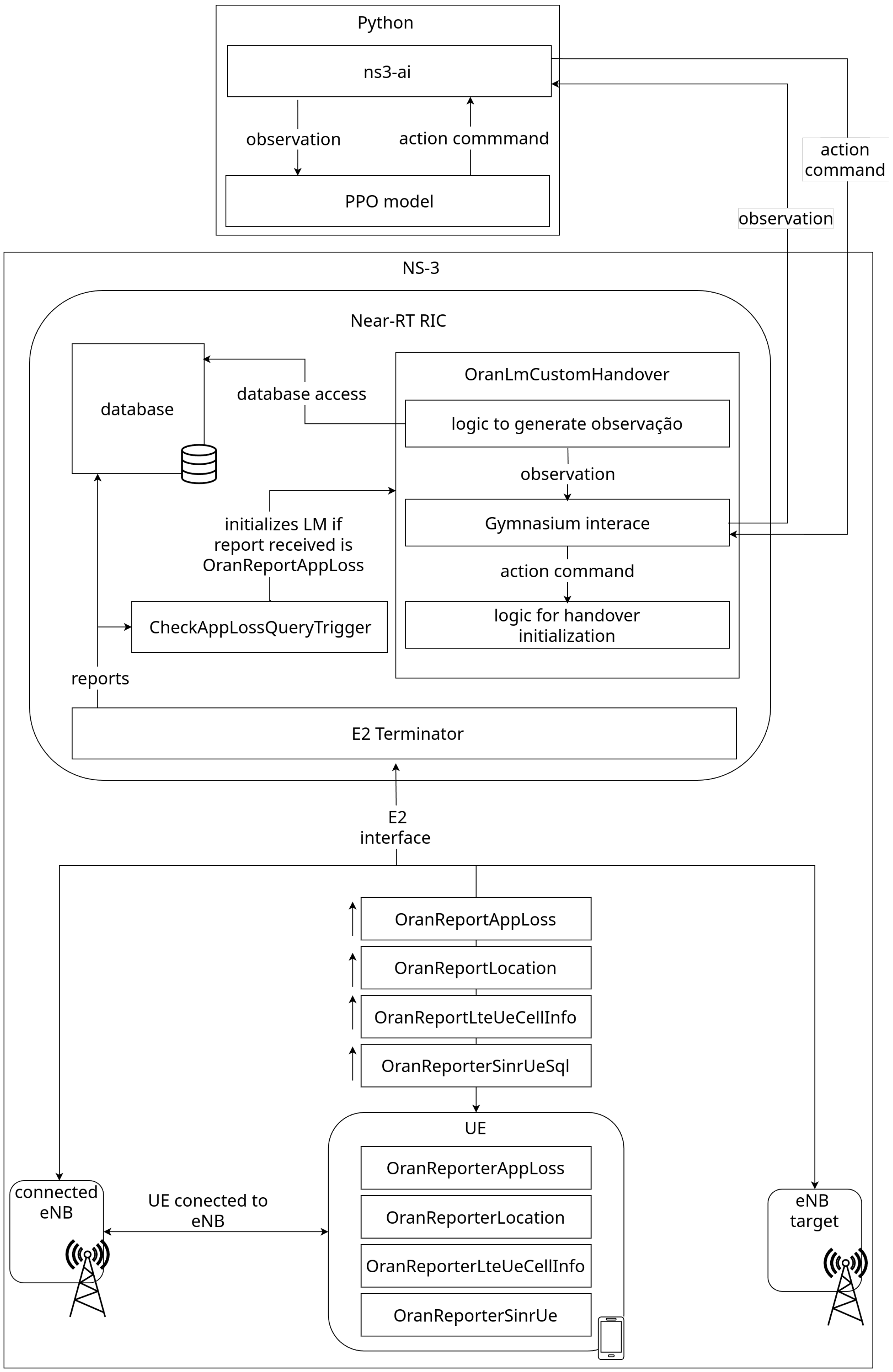
The eNBs have fixed positions in all simulation instances, while the UE starts at a random position with a random constant velocity along the X-axis. The UE's velocity depends on its starting position to ensure it moves toward the farthest eNB, frequently triggering handover scenarios.

Both the UE and eNBs are connected to the Near-RT RIC. Periodically, the UE generates reports containing its spatial position, observed packet loss rate since the last report, and its signal-to-noise ratio (SINR) with the currently connected eNB. Whenever the Near-RT RIC receives a packet loss report from a UE, the main logic module (responsible for training the machine learning model for handover management) generates an observation for the model. This observation includes packet loss rate, SINR, and the UE's distance to each eNB. The model processes this data and returns the index of the eNB the UE should connect to. If the indicated eNB matches the UE's current connection, no action is taken; otherwise, a handover process begins.

The simulation is divided into 10 files:

- oran-report-sinr-ue.cc and oran-report-sinr-ue.h implement the OranReportSinrUeSql class, which defines a report containing SINR and RSRP (Reference Signal Received Power) data for UEs. This report creates a SQLite table with fields for UE ID, connected cell ID, SINR, RSRP, and RNTI (Radio Network Temporary Identifier).
- oran-reporter-sinr-ue.cc and oran-reporter-sinr-ue.h implement a reporter generator for UEs, capturing physical layer traces from ns-3 to generate OranReportSinrUeSql reports.
- open-gym.h and open-gym.cc define the Gymnasium environment used for training, providing interfaces for communication between the Python training script and the C++ simulation. These files include methods for accessing observation/action spaces, retrieving observations, calculating rewards, and sending actions.
- oran-logic-module-train-ml-handover.cc and oran-logic-module-train-ml-handover.h implement the logic module for training the handover model. This module runs when a packet loss report (indicating >1% loss) is received or periodically every second. It generates observations from the latest UE reports stored in the database and sends them to the Python training script via ns3-ai. The reward r is calculated as r = 1 - p, where p is the packet loss rate. If the model's action indicates a new eNB, a handover is initiated.
- oran-report-trigger-apploss.h and oran-report-trigger-apploss.cc define triggers for sending packet loss reports. A report is sent if the UE's packet loss exceeds 10%, with a 20ms cooldown before re-triggering.
- Finally, the run-handover-example.py file defines the neural network, training algorithm, and initializes the training loop. It uses the stable-baselines3 library for PPO implementation. The script runs the simulation in an infinite loop, saving the model periodically and displaying performance metrics. Training continues until manually interrupted.

The figure below illustrates the example scenario.


Python
ns3-ai
observation
action commmand
PPO model
action command
observation
NS-3
Near-RT RIC
database
database access
OranLmCustomHandover
logic to generate observação
observation
Gymnasium interace
action command
logic for handover initialization
initializes LM if report received is OranReportAppLoss
CheckAppLossQueryTrigger
reports
E2 Terminator
E2 interface
OranReportAppLoss
OranReportLocation
OranReportLteUeCellInfo
OranReporterSinrUeSql
UE
OranReporterAppLoss
OranReporterLocation
OranReporterLteUeCellInfo
OranReporterSinrUe
connected eNB
UE conected to eNB
eNB target