ns-3 Public Safety Communications documentation

Release psc-7.0

National Institute of Standards and Technology (NIST)

Jan 03, 2024

CONTENTS

This is documentation for *ns-3* models relating to public safety communications, extracted from the overall *ns-3 Model Library* documentation.

This document is written in reStructuredText for Sphinx and is maintained in the `doc/psc-models` directory of the PSC repository.

CHAPTER

ONE

PUBLIC SAFETY COMMUNICATIONS

1.1 Public Safety Communications Overview

ns-3 support for public safety communications (PSC) is based on relatively new capabilities for 4G LTE systems introduced in 3GPP Release 12 and later releases. This includes Device-to-Device (D2D) communications over sidelink and mission-critical push-to-talk (MCPTT) application.

Support for public safety communications is distributed among the following five *ns-3* modules:

1) `psc`: (this module) Support for models and scenarios that are specific to public safety communications.
2) `lte`: Support for ProSe (sidelink communications).
3) `buildings`: Support for pathloss models including building effects, as defined by 3GPP with relevance to public safety scenarios.
4) `antenna`: Parabolic antenna model as described in 3GPP document TR 36.814.
5) `sip`: Session Initiation Protocol (SIP) model.

Documentation for the PSC features implemented in the *ns-3* `lte`, `buildings`, `antenna`, and `sip` modules is provided in the respective module documentation. This chapter documents the *ns-3* `psc` module.

At present, a large portion of the code related to public safety communications is found in the ProSe and UE-to-Network relay implementation in the `lte` module. This is because the ProSe services of sidelink communications, discovery, and synchronization are deeply connected to the LTE models and difficult to factor into a separate module. Features intrinsic to the low-level operation of ProSe in LTE are found in the `lte` module.

The `psc` module also includes:

1) an extensive model of mission-critical push-to-talk (MCPTT) for use in off-network or on-network scenarios.
2) a UDP-based application to generate many-to-many traffic models in a scenario.
3) an energy consumption model for flying a (single/multi)rotor unmanned aerial vehicle (UAV).
4) a new HTTP application.
5) a video streaming model.
6) a configurable generic client/server application
7) a framework to develop incident scenarios and a detailed large scale example

There is no support for legacy public safety communications such as land mobile radio system (LMRS). While LTE sidelink supports both IPv4 and IPv6, the UE-to-Network Relay feature only supports IPv6.

1.1.1 Release History

More complete release notes can be viewed also on GitHub.

Release	Date	Summary of features
v1.0	Aug 17, 2018	Release 1.0 based on ns-3.29
v1.0.1	Sep 10, 2018	Minor ns-3.29 release alignment
v1.0.2	Sep 17, 2018	Minor update for Python bindings
v2.0	Sep 6, 2019	Off-network MCPTT, UAV mobility energy, HTTP, discovery
v3.0	Apr 25, 2020	LTE UE-to-Network Relay, D2D partial coverage examples
v3.0.1	Jul 29, 2020	Aligned to ns-3.31, UE-to-Network Relay fixes
v4.0	Apr 16, 2021	On-network MCPTT, video streaming models
v5.0	Oct 20, 2021	Aligned to ns-3.35, large scale scenario and applications
v6.0	May 26, 2022	Aligned to ns-3.36.1, minor update to MCPTT helper
v7.0	Jan 4, 2024	Aligned to ns-3.40

1.1.2 Acknowledgments

Public safety communications features are based on development led by the Wireless Networks Division of the U.S. National Institute of Standards and Technology (NIST), described in publications ([NIST2016], [NIST2017], [NIST2019], and [NIST2021]). Users of the D2D features of *ns-3* are requested to cite [NIST2017] in academic publications based on these models. Users of the MCPTT features of *ns-3* are requested to cite [NIST2021] in academic publications based on these models.

The integration of the initial ProSe module with the LTE and buildings module, as well as the creation of additional examples and tests, and contributions to on-network MCPTT support, was assisted by CTTC and the University of Washington. This work was performed under the financial assistance award 70NANB17H170 and 70NANB20H179 from U.S. Department of Commerce, National Institute of Standards and Technology.

The following individuals are authors of the public safety communications extensions:

- Zoraze Ali (CTTC)
- Aziza Ben-Mosbah (NIST)
- Evan Black (NIST)
- Fernando J. Cintron (NIST)
- Samantha Gamboa (NIST)
- Wesley Garey (NIST)
- Tom Henderson (Univ. of Washington)
- Antonio Izquierdo Manzanares (NIST)
- Manuel Requena (CTTC)
- Richard Rouil (NIST)
- Raghav Thanigaivel (NIST)

1.2 Mission Critical Push-to-Talk (MCPTT)

An MCPTT service provides a reliable, always-on, always-available means of communication for first responders. This communication can be one-to-one or one-to-many. There are two modes of operation: on-network and off-

network. In on-network mode, communication takes place via a client/server setup, which means that, in LTE, a UE sends signals to an eNodeB, and talks with an MCPTT server via the core-network. But in off-network mode, communication is supported by UE devices in a peer-to-peer like fashion, where signals are only sent from UE to UE.

1.2.1 Model Description

The purpose of the MCPTT model is to enable research into the performance of MCPTT protocols in both off-network and on-network public safety scenarios. Users are expected to be interested in studying Key Performance Indicators (KPIs) such as the push-to-talk access time (KPI1) or the mouth-to-ear latency (KPI3) metrics defined in Section 6.15.3 of [TS22179].

Design

A call is the logical channel between MCPTT applications. Call control is the protocol used to create and manage these logical channels. There are three types of calls that can take place between a set of users: basic, broadcast, and private. A basic group call is one where a logical channel is setup for a group of users associated with a particular group ID so that members of the group can contend to talk in order to communicate with the other members of the group. A broadcast group call is like a basic group call, but only the initiator is allowed to speak, and once the initiator is done speaking the call is terminated. A private call exists when there is a logical channel between two applications for two users to communicate. Each type of call may be basic, imminent peril, or emergency, and this "call status" dictates which physical resources are used by this call.

In the specifications, on-network calls are described not only by their call type (basic, broadcast, and private) but also by how the supporting session is established (on-demand or pre-established), the group call model (pre-arranged or chat group (restricted)), and whether the call requires elevated access privileges to network resources (emergency or imminent peril calls). The mode of call commencement (automatic or manually with user intervention) also can be defined. The difference between an on-demand and pre-established session is that in the pre-established case, certain network setup steps are completed before a user initiates the call; these steps include negotiation of IP addresses and ports (e.g., Interactive Connectivity Establishment (ICE) procedures) and coordination of bearers in the IP Media Subsystem (IMS) core. In a prearranged group call, the group members will be defined a priori, while the chat group (restricted) call model allows members to join calls without being invited. The initial support in *ns-3* for on-network call types is the 'prearranged group call, using on-demand session, with automatic commencement,' defined in TS 24.379 [TS24379].

When off-network, all call control messages are sent to all other applications using the same socket. When a user wants to start or join a group call for a particular group ID, the UE sends out periodic "GROUP CALL PROBE" messages. If a UE with same group ID is already part of an on-going call, the UE that received the probe will respond with a "GROUP CALL ANNOUNCEMENT" message. When a UE who is not part of an on-going call receives this "GROUP CALL ANNOUNCEMENT" message, the UE then automatically joins the call or waits for the user to choose between joining the call or ignoring the call. This announcement message is also sent periodically by members of an on-going call. While part of a group call, floor control (described in the following paragraph) is used to control which UE has permission to send media, and thus, which user is allowed to talk, at any given time during a call. A user may also change the "status" of a call at any point and this is communicated via the "GROUP CALL ANNOUNCEMENT" message.

In an MCPTT group call, only a single member of the group is allowed to talk at a time (with the exception of the dual-speaker feature of on-network), and during this time all other group members need to listen. This is facilitated by the floor control protocol. The protocol consists of floor participants and a floor control server. The floor participants make requests to the floor control server, while the floor control server receives and handles these requests. In the on-network version there is a centralized MCPTT server that acts as the floor control server, but in the off-network case, the UE of the current speaker is the one acting as the floor control server (also known as the current arbitrator). It is optional to use floor control for a private call.

When off-network, all floor control messages are sent to all members of the group. When a user wants to talk, which is indicated by a PTT press during a call, the UE requests permission to send data by sending a "Floor Request" message. The current arbitrator of the call, which is the device of the user who is currently allowed to speak, will send a "Floor Granted" message to give permission to another UE to transmit media, allowing the requesting user to speak to the other members of the group, a "Floor Deny" message to deny the UE permission to send media, or a "Floor Queue Info" message if the UE must wait to transmit media. After sending a request, if the UE does not receive an appropriate message after a given amount of time, the UE will assume that there is currently no arbitrator and will send a "Floor Taken" message, giving itself permission to transmit media.

1.2.2 Design Documentation

The MCPTT model was implemented based on the 3GPP specifications TS 24.379 [TS24379] and TS 24.380 [TS24380] release 14.4.0. This includes off-network models for floor control, basic group calls, broadcast group calls, and private calls. There are also several test cases that were invented by NIST that are also included with this model. This model was created and is intended to be used with the Proximity Services (ProSe) which is provided by LTE.

The MCPTT model includes:

- A set of classes that can be used to model an MCPTT application using call control and floor control in the off-network case
- Helper classes to act as a sink for time-sensitive message and state change traces that can be used to capture the behavior of the different off-network MCPTT state machines (i.e., protocols)
- A helper class that allows users to configure and deploy MCPTT applications in a scenario with nodes
- A few examples that show how the model can be used
- Test cases produced by NIST to check state machine behavior

The MCPTT model does not take into account security. The model only goes as far as including dummy variables in appropriate messages that would carry security fields. Also, the MCPTT model currently does not make use of ProSe Per-Packet Priority (PPPP). Even though the MCPTT model includes and appropriately maintains PPPP as specified in the standards, there currently is no support in ProSe for this feature and, thus, it is not used.

Note: MCPTT was created specifically for LTE, and the main component that ties off-network MCPTT to LTE is the use of PPPP, which is provided to lower layers of LTE when sending messages for physical resource selection, but the current implementation of the off-network model does not make use of this feature since there is currently no support for it in the current ns-3 implementation of ProSe. This means that the current MCPTT model is capable of being used over more technologies than just LTE since it is just an application in *ns-3*.

Design

In *ns-3*, the `ns3::psc::McpttHelper` is the main class that a user should use to initialize and install applications to an `ns3::Node`. This helper can be used to initially configure some of the components that are associated with an MCPTT application, such as the application's data rate when generating media, the random variable that should be used to simulate a PTT, the IP address used by the application when addressing the group, etc. Once created and installed, it is up to the user to setup calls. This can be done before or after the application is started but not "dynamically" by the application itself. So if a user wanted a scenario with three UE's that will eventually take part in the same on-going group call, the user will have to first query the `ns3::psc::McpttPttApplication` objects of the three `ns3::Node`'s to create those calls. Once the calls are created, the user will also have to query the application to select the current call to reflect things such as a PTT or call release at any given time throughout the simulation.

In the following code snippet, a basic group call is created for N users with group ID 1. One second after the applications are started, the first user will push the PTT button.

```
ObjectFactory callFac;
callFac.SetTypeId(McpttCallMachineGrpBasic::GetTypeId());
callFac.Set("GroupId", UintegerValue(1));
ObjectFactory floorFac;
floorFac.SetTypeId(McpttOffNetworkFloorParticipant::GetTypeId());

for (uint32_t idx = 0; idx < clientApps.GetN(); idx++)
{
    Ptr<McpttPttApp> pttApp = DynamicCast<McpttPttApp, Application>(clientApps.
↪Get(idx));
    pttApp->CreateCall(callFac, floorFac);
    pttApp->SelectCall(0);

    if (idx == 0)
    {
        Simulator::Schedule(Seconds(start.GetSeconds() + 1.0),
                            &McpttPttApp::TakePushNotification,
                            pttApp);
    }
}
```

Identifiers

Several MCPTT identifiers are used throughout the code, as follows.

- `MCPTT User ID` The MCPTT User ID is, in practice, a SIP URI (i.e., a string value). In this ns-3 model, for simplicity, it is stored as a 32-bit unsigned integer. Clients for which the `ns3::psc::McpttHelper` installs an instance of the `ns3::psc::McpttPttApp` will be assigned a unique user ID value, starting from the initial value of 1. In addition, the MCPTT server is separately assigned a user ID, stored in the `ns3::psc::McpttServerCallMachine` instance. This is initialized by default to 0, and typically left at the default value.
- `MCPTT Group ID` The MCPTT Group ID pertains to group calls; it is also, in practice, a SIP URI (i.e., a string value). In this ns-3 model, for simplicity, it is stored as a 32-bit unsigned integer. On clients, the value is stored in the base class `ns3::psc::McpttCallMachineGrp` as an attribute, and similarly, on the server, it is stored in an attribute in the base class `ns3::psc::McpttServerCallMachineGrp`. The group ID is usually assigned explicitly by the simulation user as part of call definition.
- `SSRC (synchronization source)` This 32-bit identifier is defined for RTP (see RFC 3550) and is, in practice, supposed to be assigned randomly so as to avoid collisions. The SSRC is associated with a media stream component, and the identifier is carried in RTP and floor control messages. In on-network operation, an SSRC is also assigned to the server (included in server-originated floor control messages). The on-network arbitrator, `ns3::psc::McpttOnNetworkFloorArbitrator`, contains an attribute `TxSsrc` that defaults to the value 0. For simplicity, the clients reuse the MCPTT User ID value for SSRC; i.e., all media streams are identified as being from the same source, in the current version of this model.

MCPTT Application

The `ns3::psc::McpttPttApp` is the core component of the MCPTT model. It is the object used to manage calls and provide an API with functions that would be available to a user like, starting a call, releasing a call, entering an emergency alert, etc. It also houses a few entities to help simulate the behavior of how an MCPTT application may

be used. This class also implements the `ns3::psc::McpttMediaSink` interface, which allows for an instance of the `ns3::psc::McpttMediaSrc` class to generate and pass media messages for transmission.

The `ns3::psc::McpttMediaSrc` is simply a class that generates RTP media messages to be sent. It is used by the application to help model data traffic in the network. It is a rather simple traffic model but can be configured to alter data rate and message size.

The `ns3::psc::McpttMsgParser` is a class that reads the header of a packet to determine which message is being received. This class is used by the application as well as in some test cases.

The `ns3::psc::McpttPusher` is class used to simulate the pushing of a button. The application uses this class to simulate a user that is using the PTT button. This model is also simple but allows the user to configure it with a random variable to reflect how often a user may push and release the PTT button.

The `ns3::psc::McpttCall` class simply aggregates the components needed to have an MCPTT call. This includes a channel for both floor control and media messages, and state machines for call control and floor control.

The `ns3::psc::McpttChannel` class is a wrapper around the `ns3:Socket` class to provide an interface similar to the functions required by the specifications, as well as handle opening and closing sockets on the fly. This class is used by the `ns3::psc::McpttCall` and `ns3::psc::McpttPttApp` classes.

The `ns3::psc::McpttCallMachine` is an interface created to handle the different types of calls that are available. This includes functions for starting and leaving calls. There are many subclasses:

- `ns3::psc::McpttCallMachineGrpBasic` for basic group calls
- `ns3::psc::McpttCallMachineGrpBroadcast` for broadcast group calls
- `ns3::psc::McpttCallMachinePrivate` for private calls, and
- `ns3::psc::McpttCallMachineNull` to "turn-off" call control.

Each of the state machines except for the null state machine have several classes associated with each to represent the different states of the state machine. For example, the class `ns3::psc::McpttCallMachineGrpBroadcastStateB1` is a model of the "B1: start-stop" state of the Broadcast call control state machine from the 3GPP standard describing call control.

The `ns3::psc::McpttFloorParticipant` is an interface created to represent the floor control protocol. There are two subclasses:

- `ns3::psc::McpttOffNetworkFloorParticipant` for the floor control protocol
- `ns3::psc::McpttFloorParticipantNull` to "turn-off" floor control

Just like the state machines for call control, the `ns3:McpttOffNetworkFloorParticipant` class has an `ns3::psc::McpttOffNetworkFloorParticipantState` member which is derived by many classes such as `ns3:McpttOffNetworkFloorParticipantStateHasPerm` to represent the different states of the floor control state machine.

The `ns3::psc::McpttEmergAlertMachine` is an interface to represent the state machine used to maintain a user's emergency alert status. This can be found in TS 24.379 [TS24379]. There is currently one subclass:

- `ns3::psc::McpttEmergAlertMachineBasic` for the emergency alert protocol

In the standard, there is one emergency alert machine associated with one user, but in the current implementation there is an emergency alert machine associated with each `ns3::psc::McpttCallMachineGrpBasic` instance. Unlike the previous state machines, there is no "state" class associated with this machine, all the logic is self-contained.

The `ns3::psc::McpttCallTypeMachine` is an interface for the call type machines described in the standard. These state machines exist to maintain the call type (e.g., basic, emergency, etc.) of a call. There are two subclasses:

- `ns3::psc::McpttCallTypeMachineGrpBasic` for a basic group call, and
- `ns3::psc::McpttCallTypeMachinePriv` for a private call.

These state machines are also self-contained and are not associated with any "state" classes.

Most state machines, if not all, have many members that are of the types `ns3::psc::McpttCounter` and `ns3::psc::McpttTimer`. These classes are used to provide an API to mirror the actions that can be requested of counters and timers that are defined throughout the standards. The `ns3::psc::McpttCounter` class provides an interface that can be used as counters are described in the standard with functions like, "Increment" and "Reset". The `ns3::psc::McpttTimer` class does the same for timers described in the standard with functions like "Start", "Stop", and "Restart". The `ns3::psc::McpttTimer` is simply a wrapper around the `ns3::Timer` class.

The `ns3::psc::McpttMsg` class is a base class for all MCPTT off-network messages. This class is derived from by the `ns3::psc::McpttCallMsg` class for call control messages, the `ns3::psc::McpttFloorMsg` for floor control, and the `ns3::psc::McpttMediaMsg` for RTP media messages. The `ns3::psc::McpttCallMsg` and `ns3::psc::McpttFloorMsg` classes both have many subclasses to represent each floor control or call control message used in the off-network portion of the standard. For example, the `ns3::psc::McpttFloorMsgRequest` class represents the "Floor Request" message described in TS 24.380 [TS24380]. These classes are sent between the MCPTT applications and are consumed by the appropriate state machines throughout the simulation.

As mentioned above, the `ns3::psc::McpttMediaMsg` class is used to represent an RTP media message, but the actual header for an RTP packet is modeled by the `ns3::psc::McpttRtpHeader` class. This class just defines the fields needed for a basic RTP header which is used by the media message.

The `ns3::psc::McpttCallMsgField` and `ns3::psc::McpttFloorMsgField` classes represent call control and floor control message fields, respectively. These classes are also just used as a base and have many child classes. For example, the `ns3::psc::McpttFloorMsgRequest` class, which represents a "Floor Request" message contains a member of type `ns3::psc::McpttFloorMsgFieldUserId` that describes the ID of the MCPTT user making the floor request.

The `ns3::psc::McpttEntityId` class only exist for simulation to associate an ID with various entities described in the standard. One may find that counters, timers, and various states have an ID, and this ID just helps distinguish between multiple objects of the same type.

The `ns3::psc::McpttFloorQueue` class is a wrapper around the `std::vector` class that provides an interface for the floor queue that is described in TS 24.380 [TS24380]. This class is only associated with a floor machine for use when queuing is enabled in floor control. One can enable queuing in floor control for a particular call, simply by setting the capacity of this queue to a size greater than zero.

The `ns3::psc::McpttQueuedUserInfo` class is used by the `ns3::psc::McpttFloorQueue` class to represent the structure of information needed to store for a user when they are placed in the queue, and made to wait to transmit during floor control.

Pusher Behavior

Supplement to the MCPTT application, a pusher model exists to simulate the PTT behavior for a group of MCPTT users on a call. This model consists of a centralized entity derived from the `ns3::psc::McpttPusherOrchestrator` class that determines the interarrival time (IAT) and duration of PTT events for a set of `ns3::psc::McpttPusher` objects. The value of this model comes from its ability to simulate the PTT behavior for a group based on data that was collected and generalized from real public safety call logs. Using the data from the call logs two main elements were extracted and realized by the model, and they are, PTT and session durations. In essence, a session is a time span when users (i.e., instances of the `ns3::psc::McpttPusher` class) are actively being scheduled to push and release the PTT button, while the PTT duration determines how long a PTT push will last. These durations are generated using instances of the `ns3::EmpiricalRandomVariable` class that interpolate bins taken from the Cumulative Distribution Function (CDF) curves that were computed after parsing the call log data to determine PTT and session durations. While the durations of PTTs and sessions will resemble the actual data more precisely, the IATs of both PTTs and sessions are based on an activity factor that is set by the user. They are computed using instances of the `ns3::ExponentialRandomVariable`, which is assumed to be the

distribution of IATs. This means that the following sequence of non-overlapping steps is typical for generating PTT and session events: IAT, duration, IAT, duration, and so on.

The `ns3::psc::McpttPusherOrchestratorInterface` is a base class for the subclasses:

- `ns3::psc::McpttPusherOrchestrator`,
- `ns3::psc::McpttPusherOrchestratorSpurtCdf`,
- `ns3::psc::McpttPusherOrchestratorSessionCdf`, and
- `ns3::psc::McpttPusherOrchestratorContention`.

The class `ns3::psc::McpttPusherOrchestrator` simply selects a random pusher from the set of pushers being orchestrated and uses two instances of an `ns3::RandomVariableStream`. One for deciding push durations and another for deciding the IAT of push events. The `ns3::psc::McpttPusherOrchestratorSpurtCdf` is a wrapper around the `ns3::psc::McpttPusherOrchestrator` class that initializes the random variables used by `ns3::psc::McpttPusherOrchestrator` to match CDFs from the call logs based on the activity factor that was provided. The `ns3::psc::McpttPusherOrchestratorSessionCdf` class is a decorator that simply starts and stops an underlying orchestrator to create sessions based on the CDFs from the call log and the activity factor set by the user. Finally, the `ns3::psc::McpttPusherOrchestratorContention` class is also a decorator that can be used to introduce contention, by using a threshold and an instance of the `ns3::UniformRandomVariable` class to determine if and when an additional PTT event should occur to collide with another active PTT event.

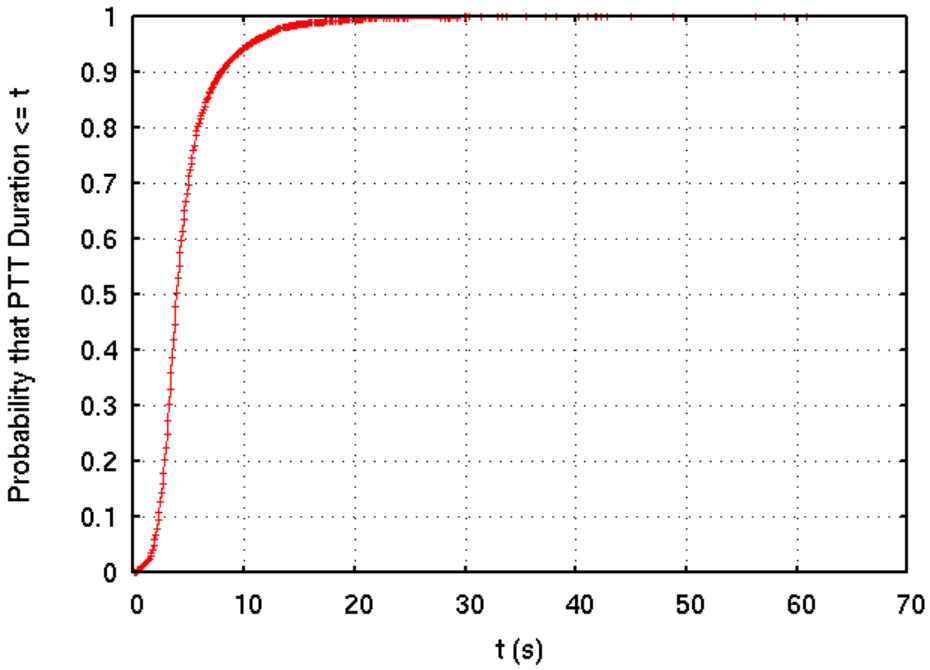


Fig. 1: PTT Duration CDF

For `ns3::psc::McpttPusherOrchestratorSpurtCdf` the PTT duration is based on the CDF of PTT durations in figure *PTT Duration CDF*, while the mean value given to the exponential random variable responsible for generating the IAT is computed as:

Average PTT IAT = (Average PTT Duration) * ((1 / Voice Activity Factor) - 1)

Here, "Average PTT IAT" is the mean value given to the exponential random variable, "Average PTT Duration" is 4.69 (which is the average PTT duration computed from the CDF data), and "Voice Activity Factory" is a value between (0, 1] that is set by the user.

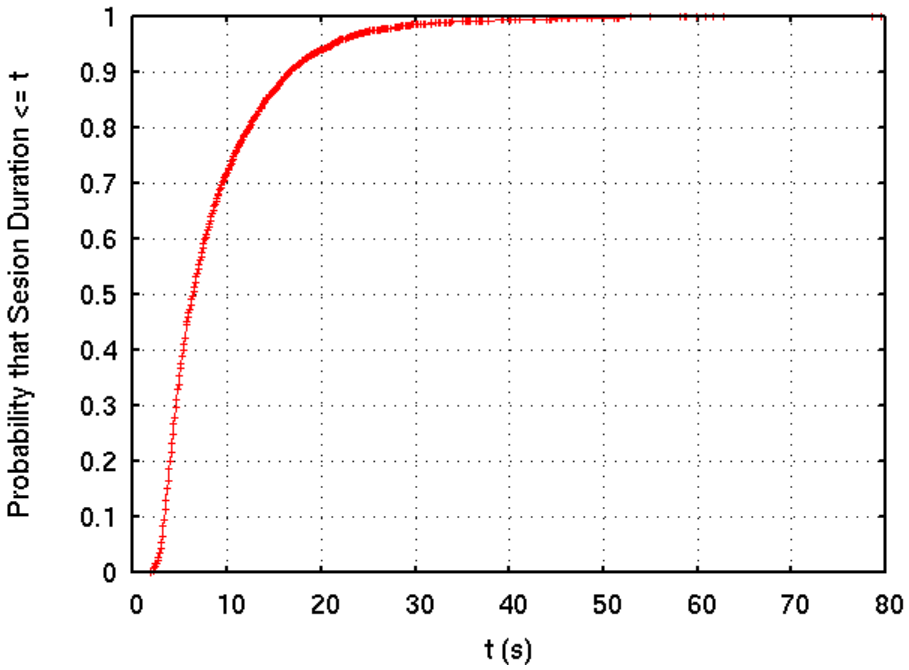


Fig. 2: Session Duration CDF

For `ns3::psc::McpttPusherOrchestratorSessionCdf` the session duration is based on the CDF in figure *Session Duration CDF*, while the mean value given to the exponential random variable responsible for generating session IAT is computed as:

Average Session IAT = (Average Session Duration) * ((1 / Session Activity Factor) - 1)

Here, "Average Session IAT" is the mean value given to the exponential random variable, "Average Session Duration" is 8.58 (which is the average session duration computed from the CDF data), and "Session Activity Factory" is a value between (0, 1] that is set by the user.

Note that the `ns3::psc::McpttPusherOrchestratorContention` class can affect the overall activity factor since it can create additional PTT requests, but how it will affect the activity factor depends on the protocol configuration (e.g., queuing, preemption, etc.). However, based on a configurable Contention Probability (CP) threshold, the probability that a single pusher's request will occur at the same time as another pusher's request is:

Probability of Experiencing Contention = 2 * CP / (1 + CP)

The following code snippet gives an example of how the pusher model is configured and attached to a set of applications. In this case the voice activity factor during a session is 0.5, which means that there will be a PTT event about 50 % of the time in each session, while the session activity factor is 1.0, which means that there will be active sessions during the entire simulation. The timeline in figure *Pusher Model Example* captures the activity of both pushers and sessions for this configuration.

```
Ptr<McpttPusherOrchestratorSpurtCdf> spurtOrchestrator =
    CreateObject<McpttPusherOrchestratorSpurtCdf>();
spurtOrchestrator->SetAttribute("ActivityFactor", DoubleValue(1.0));

Ptr<McpttPusherOrchestratorSessionCdf> sessionOrchestrator =
    CreateObject<McpttPusherOrchestratorSessionCdf>();
sessionOrchestrator->SetAttribute("ActivityFactor", DoubleValue(0.5));
sessionOrchestrator->SetAttribute("Orchestrator", PointerValue(spurtOrchestrator));
```

(continues on next page)

(continued from previous page)

```
sessionOrchestrator->StartAt(Seconds(2.0));
sessionOrchestrator->StopAt(Seconds(82.0));

mcpttHelper.Associate(sessionOrchestrator, clientApps);
```

It is also worth mentioning that if the Voice Activity Factor is less than 1.0, then depending on the combination of Voice Activity Factor and session duration, it is possible to have a session that does not have any PTT events. This is due to the fact that an active session only determines when PTT events are possible but it does not guarantee that a PTT event will occur.

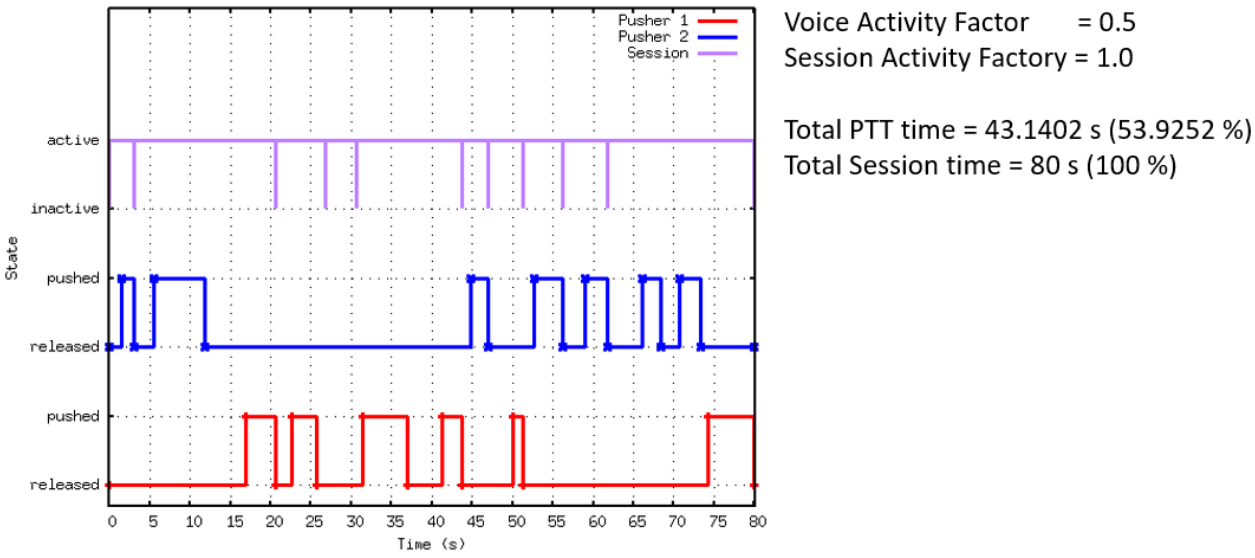


Fig. 3: Pusher Model Example

Helpers

There are five helpers:

- `ns3::psc::McpttHelper` for deploying MCPTT applications,
- `ns3::psc::McpttMsgStats` for tracing transmitted MCPTT application messages,
- `ns3::psc::McpttProseCollisionDetector` for examining ProSe operation, and
- `ns3::psc::McpttStateMachineStats` for tracing state machine state transitions.
- **`ns3::psc::ImsHelper` for adding an optional IMS;** the IMS is modeled as a single node connected to the PGW.

As stated previously, the `ns3::psc::McpttHelper` is used to configure and deploy MCPTT applications. This is the class that a user should use to configure settings that will be common across the applications and use it to create several copies of the same configuration.

The `ns3::psc::McpttMsgStats` class has the necessary sink functions to connect to and trace messages sent from app to app. One can set the name of the output file by setting the `ns3::psc::McpttMsgStats::OutputFileName` attribute. The user can also specify which types of messages should be captured. So if a user wanted to capture all message types the user should then set, `ns3::psc::McpttMsgStats::CallControl`, `ns3::psc::McpttMsgStats::FloorControl`,

and `ns3::psc::McpttMsgStats::Media` attributes to "true". If the user also wishes to include not just the type of message in the trace but also the contents of the message the user can set `ns3::psc::McpttMsgStats::IncludeMessageContent` to "true" as well.

The `ns3::psc::McpttStateMachineStats` class has the necessary functions to act as a sink to trace state transitions of the various state machines throughout a simulation. One can set the name of the output file by setting the `ns3::psc::McpttStateMachineStats::OutputFileName` attribute.

Scope and Limitations

Only the logic of the protocols described for call control and floor control were meant to be captured by the current MCPTT model. There is a feature described throughout the call control document called PPPP or Prose Per-Packet Priority that is taken into consideration at the application but is not propagated to the lower layers as specified because currently this feature is not supported by the ProSe model. On-network currently only supports pre-arranged group call, using an on-demand session, with automatic commencement. Also, elevated call types, such as 'Emergency,' and the use of Guaranteed Bit-Rate (GBR) bearers' are not modeled in on-network as this would require additional features in the LTE model.

1.2.3 Usage

Helpers

Adding IP Multimedia Subsystem (IMS)

An IMS node can be added to a simulation in a manner similar to the optional configuration of an EPC network. In the below snippet, the ImsHelper object is LteHelper, and the ImsHelper also is connected to the PGW node:

```
Ptr<ImsHelper> imsHelper = CreateObject<ImsHelper>();
imsHelper->ConnectPgw(epcHelper->GetPgwNode());
```

The IMS node itself can be fetched as follows:

```
Ptr<Node> ims = imsHelper->GetImsNode();
```

The ConnectPgw() method creates the IMS node, adds an IP (internet) stack to it, and then adds a point-to-point link between it and the PGW node. The SGi interface is assigned the 15.0.0.0/8 network by default, although this can be changed to another network. Finally, a static route towards the UE subnetwork (7.0.0.0/8) is inserted on the IMS.

Examples

There are two off-network MCPTT examples in the 'psc/examples' folder:

- `example-mcptt.cc` is a basic scenario with two users deployed randomly using the *ns-3* WiFi module in Adhoc mode
- `mcptt-lte-sl-out-of-covrg-comm.cc` is an adaptation of the LTE Sidelink example `lte-sl-out-of-covrg-comm` Mode 2 ProSe example

There are several on-network MCTT examples in the 'psc/examples' folder:

- `example-mcptt-on-network-floor-control-csma.cc` demonstrates the operation of the on-network floor control with a single call on a CSMA network.
- `example-mcptt-on-network-floor-control-lte.cc` demonstrates the operation of the on-network floor control with a single call on an LTE network.

- `example-mcptt-on-network-two-calls.cc` demonstrates two sequential calls executing sequentially in the same LTE scenario.
- `example-mcptt-on-network-two-simultaneous-calls.cc` demonstrates two concurrent calls in the same LTE network, each with different MCPTT server nodes, and an MCPTT user participating in both calls
- `mcptt-operational-modes-static.cc` demonstrates how different modes of MCPTT (on-network, off-network, and on-network using UE-to-network relay) operate in parallel, with some support for plotting access time and mouth-to-ear latency KPIs.
- `mcptt-operational-modes-mobility.cc` is similar in initial configuration to the static program above, but demonstrates mobility of three teams of four UEs into a building, and the relative performance impact of the three different modes of operation (on-network, off-network, and on-network with UE-to-network relay).

mcptt-lte-sl-out-of-covrg-comm

The program `mcptt-lte-sl-out-of-covrg-comm.cc` is an adaptation of the LTE Sidelink example `lte-sl-out-of-covrg-comm.cc` documented in the LTE module. The example was adapted to replace the simple UDP application with MCPTT. The following code excerpts highlight the main aspects of configuring MCPTT in a program.

The first block of code below highlights the use of the class `ns3::psc::McpttHelper` to encapsulate configuration statements on the key objects involved in the MCPTT service. The helper exposes some methods that allow for custom configuration of the PttApp class and attributes, MediaSrc class and attributes, and Pusher class and attributes. In the below, the configuration of `ns3::psc::McpttPusher` to Automatic operation means that the pushing and releasing times will be driven by random variables, and additional methods to easily configure these random variables are provided. Following the configuration of these objects, the usual pattern in *ns-3* for using the application helper to install onto the set of ueNodes, configure a start time, and configure a stop time are followed.

```
ApplicationContainer clientApps;
McpttHelper mcpttHelper;
if (enableNsLogs)
{
    mcpttHelper.EnableLogComponents();
}
mcpttHelper.SetPttApp("ns3::psc::McpttPttApp",
                      "PeerAddress",
                      Ipv4AddressValue(peerAddress),
                      "PushOnStart",
                      BooleanValue(true));
mcpttHelper.SetMediaSrc("ns3::psc::McpttMediaSrc",
                        "Bytes",
                        UintegerValue(msgSize),
                        "DataRate",
                        DataRateValue(dataRate));
mcpttHelper.SetPusher("ns3::psc::McpttPusher", "Automatic", BooleanValue(true));
mcpttHelper.SetPusherPttInterarrivalTimeVariable("ns3::NormalRandomVariable",
                                                 "Mean",
                                                 DoubleValue(pushTimeMean),
                                                 "Variance",
                                                 DoubleValue(pushTimeVariance));
mcpttHelper.SetPusherPttDurationVariable("ns3::NormalRandomVariable",
                                         "Mean",
                                         DoubleValue(releaseTimeMean),
```

(continues on next page)

(continued from previous page)

```
                                          "Variance",
                                          DoubleValue(releaseTimeVariance));

clientApps.Add(mcpttHelper.Install(ueNodes));
clientApps.Start(startTime);
clientApps.Stop(stopTime);
```

The above will prepare each UE for the service, but call configuration remains to be configured. For a basic group call type, using the basic floor machine, the helper provides a single statement to configure the call, as follows.

One could also use the following snippet to configure an `ns3::psc::McpttPusherOrchestrator` to control when push and release events occur for a set of `ns3::psc::McpttPttApp` applications.

```
ApplicationContainer clientApps;
McpttHelper mcpttHelper;
if (enableNsLogs)
{
    mcpttHelper.EnableLogComponents();
}
mcpttHelper.SetPttApp("ns3::psc::McpttPttApp",
                      "PeerAddress",
                      Ipv4AddressValue(peerAddress),
                      "PushOnStart",
                      BooleanValue(true));
mcpttHelper.SetMediaSrc("ns3::psc::McpttMediaSrc",
                        "Bytes",
                        UintegerValue(msgSize),
                        "DataRate",
                        DataRateValue(dataRate));
mcpttHelper.SetPusher("ns3::psc::McpttPusher", "Automatic", BooleanValue(false));

clientApps.Add(mcpttHelper.Install(ueNodes));
clientApps.Start(startTime);
clientApps.Stop(stopTime);

Ptr<McpttPusherOrchestratorSpurtCdf> orchestrator =
    CreateObject<McpttPusherOrchestratorSpurtCdf>();
orchestrator->SetAttribute("ActivityFactor", DoubleValue(0.5));
orchestrator->StartAt(startTime);

mcpttHelper.Associate(orchestrator, clientApps);
```

The following method:

```
mcpttHelper.ConfigureBasicGrpCall(clientApps, usersPerGroup);
```

encapsulates the following operations:

- sets the call control state machine type to `ns3::psc::McpttCallMachineGrpBasic`
- sets the floor control state machine type to `ns3::psc::McpttOffNetworkFloorParticipant`
- iterates across the clientApps in the provided application container. If the provided `usersPerGroup` value is equal to or greater than the size of the `clientApps` container, all instances of `McpttPttApp` will be included in a call with the same GroupId. If `usersPerGroup` is less than the size of `clientApps`, the first `usersPerGroup` will be placed into a call with the first GroupId, the second `usersPerGroup` will be placed into a call with the second GroupId, and so on until the `clientApps` have all been handled. The base GroupId is an optional argument to this method, but defaults to the value of 1 if not provided.

- creates the call instance on each `McpttPttApp`

The various configuration variables used in the above are set near the top of the main program, as follows.

```
// MCPTT configuration
uint32_t usersPerGroup = 2;
DataRate dataRate = DataRate("24kb/s");
uint32_t msgSize = 60;            // 60 + RTP header = 60 + 12 = 72
double pushTimeMean = 5.0;        // seconds
double pushTimeVariance = 2.0;    // seconds
double releaseTimeMean = 5.0;     // seconds
double releaseTimeVariance = 2.0; // seconds
Ipv4Address peerAddress = Ipv4Address("225.0.0.0");
Time startTime = Seconds(2);
Time stopTime = simTime;
```

Here, it is worth noting that the configuration sets two users per group, and the example only has two UEs, so both UEs will belong to the same GroupId. Also, the peerAddress value is set to an IPv4 multicast address. This value should not be changed without similarly changing the IPv4 group address of the TFT configuration further down in the LTE configuration part of the program. The TFT controls the mapping of IP packets to sidelink bearers. If the packets are sent to a peer address for which a suitable TFT is not configured, they will be dropped in the sending UE's stack.

Another difference with respect to the D2D example is that, presently, only IPv4 addressing is supported, so there is no command-line option to use IPv6.

A further difference with respect to the D2D example is that the command-line option to enable *ns-3* logging, `--enableNsLogs`, will not enable LTE logging as in the LTE D2D example, but will instead enable all MCPTT logs, as shown above.

Finally, we note four tracing statements inserted near the bottom of the program:

```
NS_LOG_INFO ("Enabling MCPTT traces...");
mcpttHelper.EnableMsgTraces();
mcpttHelper.EnableStateMachineTraces();
mcpttHelper.EnableMouthToEarLatencyTrace("mcptt-m2e-latency.txt");
mcpttHelper.EnableAccessTimeTrace("mcptt-access-time.txt");
```

These statements are explained in the next section. Some other aspects of LTE tracing are omitted in this modified example, in order to focus on the MCPTT configuration.

mcptt-on-network-two-calls

The program `mcptt-on-network-two-calls.cc` is an adaptation of the LTE example `lena-simple-epc.cc` to experiment with two MCPTT on-network calls involving two different pairs of UEs to a single MCPTT server managing both calls. The first group call (between the first pair of UEs) starts when the McpttPttApps start at time 2 seconds and runs until simulation time 16 seconds. The second call starts at time 18 seconds and runs until time 34 seconds.

This example provides examples of typical statements necessary to configure on-network calls. The first necessary configuration is to add the MCPTT server to the EPC (via the notional IMS) with an ImsHelper.

```
Ptr<ImsHelper> imsHelper = CreateObject<ImsHelper>();
imsHelper->ConnectPgw(epcHelper->GetPgwNode());
```

MCPTT clients are added in a manner similar to off-network configuration; e.g.:

```
McpttHelper mcpttClientHelper;
mcpttClientHelper.SetPttApp("ns3::psc::McpttPttApp");
...
clientAppContainer1.Add(mcpttClientHelper.Install(ueNodePair1));
clientAppContainer1.Start(start);
clientAppContainer1.Stop(stop);
```

However, in addition, an MCPTT server app must be configured:

```
Ptr<McpttServerApp> serverApp = DynamicCast<McpttServerApp>(serverAppContainer.
↪Get(0));
Ipv4Address serverAddress = Ipv4Address::ConvertFrom(imsHelper->GetImsGmAddress());
serverApp->SetLocalAddress(serverAddress);
```

and in a similar manner, the local IP address of the UEs is configured:

```
for (uint32_t index = 0; index < ueIpIface.GetN(); index++)
{
    Ptr<McpttPttApp> pttApp = clientAppContainer.Get(index)->GetObject<McpttPttApp>();
    Ipv4Address clientAddress = ueIpIface.GetAddress(index);
    pttApp->SetLocalAddress(clientAddress);
    NS_LOG_INFO("client " << index << " ip address = " << clientAddress);
}
```

The next few statements show how to use a McpttCallHelper to configure the on-network components of floor arbitrator, the 'towards participant' state machine (the component of the floor control at the server that maintains state for each participant), the participant state machine, and properties of the call configurations on the server.

```
McpttCallHelper callHelper;
// Optional statements to tailor the configurable attributes
callHelper.SetArbitrator("ns3::psc::McpttOnNetworkFloorArbitrator",
                         "AckRequired", BooleanValue(false),
                         "AudioCutIn", BooleanValue(false),
                         "DualFloorSupported", BooleanValue(false),
                         "TxSsrc", UintegerValue(100),
                         "QueueingSupported", BooleanValue(true));
callHelper.SetTowardsParticipant("ns3::psc::McpttOnNetworkFloorTowardsParticipant",
                                 "ReceiveOnly", BooleanValue(false));
callHelper.SetParticipant("ns3::psc::McpttOnNetworkFloorParticipant",
                          "AckRequired", BooleanValue(false),
                          "GenMedia", BooleanValue(true));
callHelper.SetServerCall("ns3::psc::McpttServerCall",
                         "AmbientListening", BooleanValue(false),
                         "TemporaryGroup", BooleanValue(false));
```

The next few statements configure the calls themselves, using the call helper. The user must specify the call type and group ID, the client (McpttPttAtt) applications to configure, the server application, and then the start and stop times of the call.

```
McpttCallMsgFieldCallType callType = McpttCallMsgFieldCallType::BASIC_GROUP;
// Add first call, to start at time 2 and stop at time 10
// Call will involve two nodes (7 and 8) and the MCPTT server (node 3)
uint32_t groupId = 1;
callHelper.AddCall(clientAppContainer1, serverApp, groupId, callType, Seconds(2),␣
↪Seconds(16));

// Add second call, on new groupId, to start at time 8 and stop at time 15
```

(continues on next page)

(continued from previous page)

```
// Call will involve two nodes (9 and 10) and the MCPTT server (node 3)
groupId = 2;
callHelper.AddCall(clientAppContainer2, serverApp, groupId, callType, Seconds(18),␣
↪Seconds(34));
```

Finally, the MCPTT tracing can be enabled to trace messages, state machine transitions, and statistics such as mouth-to-ear latency and access time.

```
NS_LOG_INFO("Enabling MCPTT traces...");
McpttTraceHelper traceHelper;
traceHelper.EnableMsgTraces();
traceHelper.EnableStateMachineTraces();
traceHelper.EnableMouthToEarLatencyTrace("mcptt-m2e-latency.txt");
traceHelper.EnableAccessTimeTrace("mcptt-access-time.txt");
```

mcptt-on-network-two-simultaneous-calls

The program `mcptt-on-network-two-simultaneous-calls.cc` differs from the previously introduced `mcptt-on-network-two-calls.cc` example as follows. First, two calls between three UEs are configured to run concurrently, rather than sequentially, for a duration of nearly 60 seconds. Second, each call has a different MCPTT server. Third, one of the MCPTT users is configured to participate in both calls, and to switch between the two calls every second.

The main change in configuration is to configure a second IMS network (essentially, a second MCPTT server) and to configure its network address to another subnet instead of the default 15.0.0.0:

```
Ptr<ImsHelper> imsHelper2 = CreateObject<ImsHelper>();
imsHelper2->SetImsIpv4Network(Ipv4Address("16.0.0.0"), Ipv4Mask("255.0.0.0"));
imsHelper2->ConnectPgw(epcHelper->GetPgwNode());
```

The first user is added to both the first and second call groups:

```
 ApplicationContainer callContainer1;
 callContainer1.Add(clientAppContainer1.Get(0));
 ...
 ApplicationContainer callContainer2;
...
// Add the first user to the second call also
callContainer2.Add(clientAppContainer1.Get(0));
```

and the two calls configured in the call helper have different server apps:

```
McpttCallMsgFieldCallType callType = McpttCallMsgFieldCallType::BASIC_GROUP;
uint32_t groupId = 1;
callHelper.AddCall(callContainer1, serverApp, groupId, callType, callStartTime,␣
↪callStopTime);

groupId = 2;
callHelper.AddCall(callContainer2, serverApp2, groupId, callType, callStartTime,␣
↪callStopTime);
```

Finally, some events must be added to cause the first MCPTT user to switch calls at specified times. This is done with the `McpttPttApp::SelectCall` method, such as follows:

```
// schedule events to cause the user (McpttPusher) to switch calls at the
// configured times in the simulation
Simulator::Schedule(callStartTime, &McpttPttApp::SelectCall, pttApp, 1, true);
Simulator::Schedule(firstSwitchTime, &McpttPttApp::SelectCall, pttApp, 2, true);
Simulator::Schedule(secondSwitchTime, &McpttPttApp::SelectCall, pttApp, 1, true);
```

The program output of this program shows application and floor control events traced on userId 1, illustrating that as one call is selected (sent to the foreground), the other call machine still evolves in the background.

mcptt-operational-modes-static

The program `mcptt-operational-modes-static.cc` demonstrates how different modes of MCPTT (on-network, off-network, and on-network using UE-to-network relay) operate in parallel, with some support for plotting access time and mouth-to-ear latency KPIs from the resulting traces.

The program creates three teams of UEs (default of 4 UEs per team) with one team in each of the three modes. The off-network and on-network using relay teams are positioned within a building, preventing full on-network operation for those teams. Each team creates an MCPTT call and uses the automatic pusher model to drive an MCPTT call for the duration of the call (default of 1000 seconds duration).

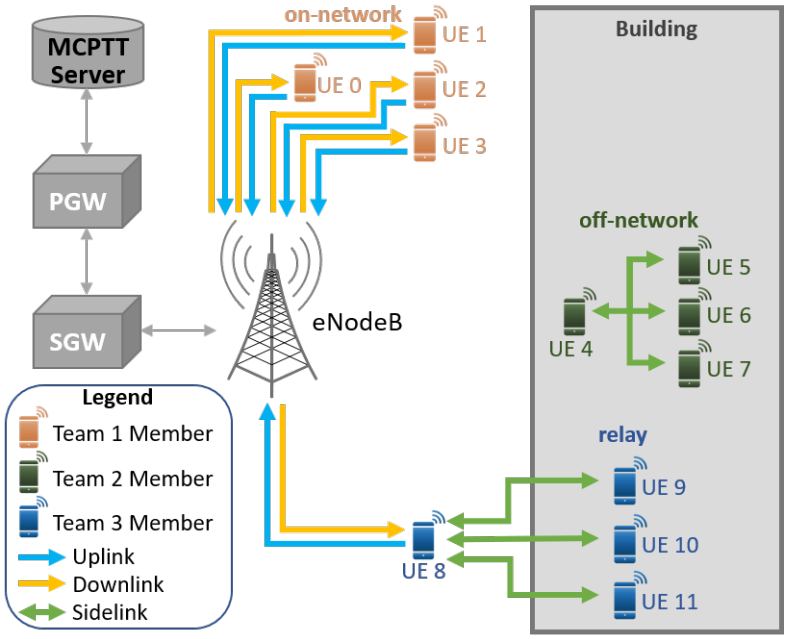


Fig. 4: MCPTT operational modes scenario

A number of traces are generated:

- `mcptt-operational-modes-static-access-time.dat`: Floor access time trace
- `mcptt-operational-modes-static-m2e-latency.dat`: Talk-spurt mouth-to-ear latency tracing
- `mcptt-operational-modes-static-msg-stats.dat`: MCPTT message trace
- `mcptt-operational-modes-static-pc5-signaling.dat`: Relay PC5 interface trace
- `mcptt-operational-modes-static-positions.dat`: UE position trace
- `mcptt-operational-modes-static-rsrp-measurement.dat`: LTE RSRP measurement trace

- `mcptt-operational-modes-static-state-machine-stats.dat`: MCPTT state machine trace

At the end of the program execution, some output is printed on the terminal regarding the number of packets sent and received by each node.

The call duration, whether floor control queueing is enabled or not (disabled by default), and the number of UEs per team are configurable as command-line options. A simulation progress display is also optionally enabled by the `--showProgress` option.

Finally, some plotting programs (requiring Python Matplotlib) are available to create CDF plots of the access time and mouth-to-ear latencies of the three teams. The programs are named `mcptt-operational-modes-plot-access-time.py` and `mcptt-operational-modes-plot-m2e-latency.py`. The programs output, by default, an EPS file showing the CDF of access time and mouth-to-ear latency, respectively. They should be run from the same directory as the traces (or the optional program argument can be used to point to the applicable trace file). The following figures show the output produced using the default configuration of `mcptt-operational-modes-static.cc`.

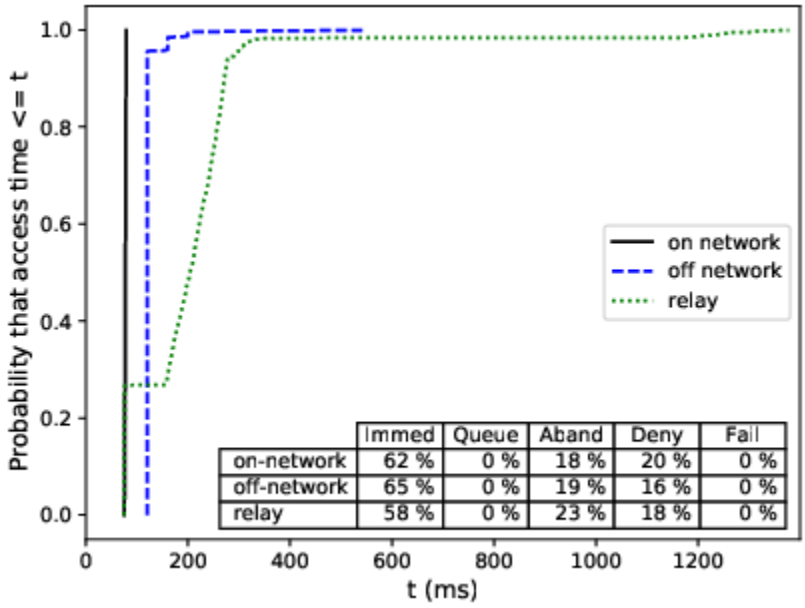


Fig. 5: MCPTT operational modes static access time

mcptt-operational-modes-mobility

- `mcptt-operational-modes-mobility.cc` is similar in initial configuration to the static program above, but demonstrates mobility of three teams of four UEs into a building, and the relative performance impact of the three different modes of operation (on-network, off-network, and on-network with UE-to-network relay).

Traces

There are currently four traces that can be activated by using the `ns3::psc::McpttHelper`, and this can be done by following the example given below, after all the applications have been created.

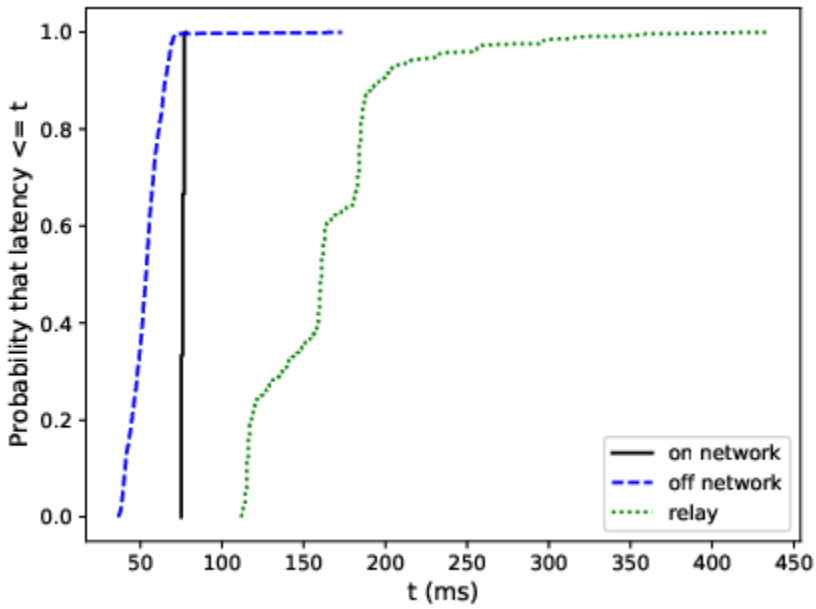


Fig. 6: MCPTT operational modes static mouth-to-ear latency

```
NS_LOG_INFO ("Enabling MCPTT traces...");
mcpttHelper.EnableMsgTraces (); // Enable message trace
mcpttHelper.EnableStateMachineTraces (); // Enable state trace
mcpttHelper.EnableMouthToEarLatencyTrace ("mcptt-m2e-latency.txt");
mcpttHelper.EnableAccessTimeTrace ("mcptt-access-time.txt");
```

The `ns3::psc::McpttMsgStats` class is used for tracing MCPTT messages at the application layer and produces a file with the default name "mcptt-msg-stats.txt" with the following file format.

```
time(s) nodeid callid ssrc selected  rx/tx bytes  message
```

There may also be an additional field at the end of the row called "message" that will be included if the `ns3::psc::McpttMsgStats::IncludeMessageContent` attribute is set to "true". The "time(s)" column describes the time (in seconds) at which a message was sent/received. The "nodeid" column contains the ns3::Node ID value of the sender or receiver, and the "callid" column contains the call ID for this message. The "ssrc" field prints the RTP synchronization source (SSRC) field if available. The "selected" column indicates whether the message was sent or received for the currently selected call on the client, or "N/A" for server messages. The "rx/tx" column indicates if the message was sent or received, i.e., if "rx" is the value in the column then that means the message was received, while a value of "tx" indicates that the message was sent. The "bytes" columns indicates the size (in bytes) of the message that was generated at the application layer. The "message" column, if present, includes the string representation of the message that was sent and includes message field names and values.

The `ns3::psc::McpttStateMachineStats` is used for tracing state machine state transitions and produces a file with the default name, "mcptt-state-machine-stats.txt", with the following format.

```
time(s) userid callid selected     typeid     oldstate        newstate
```

The "time(s)" column describes the time (in seconds) at which the state transition took place. The "userid" column contains the user ID of the MCPTT user that the state machine is associated with. The "callid" column contains the ID of the call the state machine is associated with. The "selected" column indicates whether the message was sent or received for the currently selected call on the client, or "N/A" for server messages. The "typeid" column

contains the string representation of the state machine's `ns3::TypeId`. The "oldstate" column contains the string representation of the `ns3::psc::McpttEntityId` which gives the name of the state that the state machine was in BEFORE the transition took place. And finally, the "newstate" column contains the string representation of the `ns3::psc::McpttEntityId` which gives the name of the state that the state machine was in AFTER the transition took place.

One of the key performance indicators (KPI) defined for MCPTT is the mouth-to-ear latency. More information about this statistic can be found in NIST technical report NISTIR 8206 [NIST.IR.8206]. When the *ns-3* model generates the first RTP packet of a talk spurt, the timestamp is encoded into the payload of that RTP packet and every subsequent RTP packet of the talk spurt. Upon receipt of an `ns3::psc::McpttMediaMsg`, the receiving application will check whether the packet contains a newer 'start-of-talkspurt' timestamp. If so, this indicates the reception of the first RTP packet of a new talk spurt from a sender. The latency of all such talk spurts, traced from the perspective of each receiving application, can be traced using the helper method `ns3::psc::McpttHelper::EnableMouthToEarLatencyTrace()`. This method can be called with no arguments so that mouth-to-ear latency samples can be captured in the simulation via the `ns3::psc::McpttHelper::MouthToEarLatencyTrace` traced callback, or a single argument, the trace filename, can be provided so that the collected samples are also written to a file. The output file is formatted as follows.

```
  time(s) ssrc nodeid callid latency(s)
3.727000    3      8      1   0.024010
3.727000    3      9      1   0.024010
```

In the example, node ID 8 started to receive a talk spurt at time 3.727 s, on call ID 1, with a mouth-to-ear latency of 24 ms (see Figure 1 of [NIST.IR.8206]).

The MCPTT Access time is defined as KPI1 in [TS22179], and measures the time between a request to speak (normally by pushing to talk) and when the user receives an indication from floor control to start speaking. In the ns-3 model, this is measured by state transitions of the floor machinei; the initial request to speak is marked by a change to a Pending Request state in the MCPTT user's floor control machine, and permission is marked by the transition to the Has Permission state. There can be a few possible state transition outcomes. The first is that the floor control server immediately grants permission. The second is that the request is initially queued but later granted. The third is that the request is denied (perhaps because queuing is disabled or the queue is full). the fourth is that the request ultimately fails, either due to the user (pusher) abandoning the request, or the end of the call, or some other reason.

The latency and outcomes of access requests can be traced using the helper method `ns3::psc::McpttHelper::EnableAccessTimeTrace()`. If no arguments are provided to this method, then samples can be captured in the simulation via the `ns3::psc::McpttHelper::AccessTimeTrace` traced callback, or if a single argument, the trace filename, is provided then the collected samples will also be written to a file. The output file is formatted as follows.

```
  time(s) userid callid result latency(s)
 7.246000      1      0      I   0.024115
11.251000      2      0      Q   1.302355
22.689000      4      1      I   0.024172
25.907000      3      1      Q   2.652518
28.952000      4      1      I   0.024378
34.017000      3      1      F   0.651460
```

The above example illustrates that at time 7.246 s, user ID 1 from call ID 0 was able to immediately request the floor in a transaction that took a bit more than 24 ms. The timestamp in the first column indicates the time at which the outcome is decided, not when the floor request was initiated. The five possible results described above can be filtered based on the fourth column above, with "I" denoting "immediate", "Q" denoting "queued", "D" denoting "denied", "F" denoting "failed" (such as due to message loss), and "A" denoting "abandoned" (when the PTT button is released while the request is pending).

When measuring access time in ns-3, we recommend to count the "I" and "Q" outcomes and filter out the "D", "F",

and "A" outcomes. The [TS22179] standard suggests that access time should be less than 300 ms for 99 % of all MCPTT requests, but suggests that the system should have negligible backhaul delay and not be overloaded (less than 70 % load per node) when comparing against this threshold. In simulations, the access time may rise above 300 ms due to non-negligible backhaul delay, the pusher model (if too many MCPTT users are contending for the floor), or congestion or transmission losses in the LTE network.

1.2.4 Testing and Validation

NIST used the 3GPP standards to create over 50 test cases for off-network MCPTT operation. More information about those test cases can be found in NIST technical report NISTIR 8236 [NIST.IR.8236].

On-network MCPTT call control has been tested against the protocol conformance specification 3GPP TS 36.579-2 [TS365792]. The current test suite checks steps 1-11 in Table 6.1.1.1.3.2-1 for the on-network, on-demand pre-arranged group call with automatic commencement mode. The underlying SIP module supporting the on-network MCPTT call control has also been tested against the SIP RFCs.

On-network MCPTT floor control has been tested for several floor control scenarios, including:

- UE floor release after initiating the call
- UE floor grant after another UE initiates the call and then releases the floor
- UE floor revoked after higher priority UE requests floor
- UE floor denied due to another UE requesting the floor
- UE floor queued and later cancelled by the requesting UE
- UE floor queued and later granted after another UE releases the floor
- Floor control with dual floor control enabled (two UEs holding the floor at the same time

1.3 UDP Group Echo Server

The class `ns3::psc::UdpGroupEchoServer` implements a group echo server that echoes received UDP datagrams to a set of clients. The policy of generating replies can be tailored through the use of attributes.

The relevance of this model to public safety communications is that such scenarios often require a many-to-many group communications application, and existing *ns-3* applications are not suitable to generate such traffic.

The UDP group echo implementation is authored by Fernando J. Cintron (fernando.cintron@nist.gov) and is derived from the UdpEchoServer found in the *ns-3* applications module.

1.3.1 Model Description

The implementation is provided in the following files:

- `src/psc/model/udp-group-echo-server.{h,cc}` The model itself
- `src/psc/helper/udp-group-echo-helper.{h,cc}` Helper code for configuration

Additionally, a simple example is provided at `src/psc/examples/example-udp-group-echo.cc`.

The model is based on the `ns3::psc::UdpEchoServer`, but differs in that the existing server only handles one client, while the group echo server handles one or more clients. The behavior of the `ns3::psc::UdpEchoServer` can be reproduced (i.e., it is a special case of this object).

The model works as follows. The UdpGroupEchoServer is an *ns-3* application, listening for UDP datagrams on a configured UDP port. Upon receipt of a datagram for the first time from a client, the server records the client and

a timestamp for when the packet was received. The server then decides whether to forward the packet back to one or more clients, on a pre-configured 'EchoPort'. The set of possible clients is built dynamically based on received packets.

There are a few configurable policies:

1. The server may be configured to echo only to the client that originated the packet (similar to the `UdpEchoServer`).
2. The server may be configured to echo to a group of clients including the sender. Furthermore, the sending client may be excluded from the response.
3. The server may be configured to echo to a group of clients including the sender, so long as the server has heard from each client within a configurable timeout period. Furthermore, the sending client may be excluded from the response.

Attributes

The following is the list of attributes:

- `Port`: Port on which the server listens for incoming packets, default value of 9.
- `EchoPort`: Port on which the server echoes packets to client, default value of 0.
- `Mode`: Mode of operation, either no group session (reply to sender only), timeout limited session (replicate to all clients for which a packet has been received from them within the configured timeout period), and infinite session (reply to all known clients). The default is no group session.
- `Timeout`: Inactive client session expiration time, default of zero seconds.
- `EchoClient`: Whether the server echoes back to the sending client, default value of true.,

Trace sources

The model also provides a `Rx` trace source for all received datagrams.

1.3.2 Usage

A simple example based on CSMA links is provided in the file `src/psc/examples/example-udp-group-echo.cc`.

```
//  *    *
// n0   n1   ... n(nExtra)  n(1+nExtra)
//  |    |    |    |          |
//  ==========================
//     LAN 10.1.2.0
```

By default, node n0 is the client and node n1 is the server, although additional nodes can be added with the `--nExtra` argument. If there are more nodes, the highest numbered node is the server.

Each client is configured with an on-off traffic generator that sends traffic at random times to the server. The following program options exercise some of the configuration of the server:

```
$ ./ns3 run 'example-udp-group-echo --PrintHelp'

Program Options:
  --nExtra:      Number of "extra" CSMA nodes/devices [0]
```

(continues on next page)

(continued from previous page)

```
  --echoClient:  Set EchoClient attribute [true]
  --mode:        Set Mode attribute (InfSession|NoGroupSession|TimeoutLimited)␣
↪[InfSession]
  --timeout:     Set Timeout attribute [+0.0ns]
  --verbose:     Tell echo applications to log if true [true]
  --enablePcap:  Enable PCAP file output [false]
  --time:        Simulation time [10]
```

By default, the number of extra CSMA nodes/devices is zero, so there will be only one client and one server. By running with the existing defaults, the program will configure the server to echo back to the client, use the 'InfSession' mode of operation (to echo without considering timeout value) and a timeout value of 0 (which is not used in this mode). Logging is also enabled, and running the program with defaults yields output such as:

```
$ ./ns3 run 'example-udp-group-echo'
...

8.7006 server received 41 bytes from 10.1.2.1 port 49153
Client found; old timestamp: 8.6806
New timestamp: 8.7006
8.7006 number of clients: 1
------------------------------------------------------------
               Client     Session
------------------------------------------------------------
         10.1.2.1:49153          0
============================================================
8.7006 server sent 41 bytes to 10.1.2.1 port 49153
```

This shows that node 10.1.2.1 sent data at time 8.7006 that was echoed back to the client. Running with the option `--nExtra=2` shows:

```
$ ./ns3 run 'example-udp-group-echo --nExtra=2'

...

9.99949 server received 41 bytes from 10.1.2.1 port 49153
Client found; old timestamp: 9.97949
New timestamp: 9.99949
9.99949 number of clients: 3
------------------------------------------------------------
               Client     Session
------------------------------------------------------------
         10.1.2.1:49153          0
         10.1.2.2:49153 0.00647334
         10.1.2.3:49153 0.00426833
============================================================
9.99949 server sent 41 bytes to 10.1.2.1 port 49153
9.99949 server sent 41 bytes to 10.1.2.2 port 49153
9.99949 server sent 41 bytes to 10.1.2.3 port 49153
```

Here, one packet sent leads to three packets echoed, and the data under the 'Session' column shows the time in seconds since the last packet was received from each client. In mode 'InfSession', there is no explicit timeout, but if we set a timeout to be something small, such as 5ms, and set the mode to 'TimeoutLimited', we can suppress the response to node 10.1.2.2 because it has last polled the server over 6ms ago.

```
$ ./ns3 run 'example-udp-group-echo --nExtra=2 --timeout=5ms --mode=TimeoutLimited'
```

(continues on next page)

(continued from previous page)

```
...

9.99949 server received 41 bytes from 10.1.2.1 port 49153
9.99949 number of clients: 3
----------------------------------------------------------
               Client     Session
----------------------------------------------------------
         10.1.2.1:49153          0
         10.1.2.2:49153 0.00647334 **Session Expired!**
         10.1.2.3:49153 0.00426833
==========================================================
9.99949 server sent 41 bytes to 10.1.2.1 port 49153
9.99949 server sent 41 bytes to 10.1.2.3 port 49153
```

As seen above, the session has expired to 10.1.2.2 with this timeout and mode setting, so the echo response is suppressed.

The C++ code for setting the server is fairly standard *ns-3* syntax and container/helper-based code, as exemplified below:

```
uint16_t serverPort = 9;
// Attributes 'timeout', 'mode', 'echoClient' may be set above
UdpGroupEchoServerHelper echoServer (serverPort);
echoServer.SetAttribute ("Timeout", TimeValue (timeout));
echoServer.SetAttribute ("Mode", StringValue (mode));
echoServer.SetAttribute ("EchoClient", BooleanValue (echoClient));
ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma - 1));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (simTime));
```

1.4 UAV Mobility Energy Model

The class `ns3::psc::UavMobilityEnergyModel` implements an Energy Model that bases the current on how a Node is moving.

This model is roughly based on the energy model released under the GNU GPL v2 found here: ns3-urbanuavmobility

The model is intended to provide simple energy costs for flying a (single/multi)rotor unmanned aerial vehicle (UAV). The default costs are not intended to perfectly model reality, but rather to consider flight time in cases where otherwise it may have been neglected.

1.4.1 Model Description

The implementation is provided in the following files:

- `src/psc/model/uav-mobility-energy-model.{h,cc}` The model
- `src/psc/model/uav-mobility-energy-model-helper.{h,cc}` A helper

A low-level example may be found in:

`src/psc/examples/uav-mobility-energy-example.cc`.

An example using the helper may be found in:

`src/psc/examples/uav-mobility-energy-helper-example.cc`.

The model sets the current based on five basic modes for movement:

- **STOP:** Not Moving
- **HOVER:** The Node has a velocity of 0 m/s, but the Node is off of the ground (z > 0)
- **ASCEND/DESCEND:** The Node is moving only on the z-axis
- **MOVE:** The Node is moving on the x-axis or y-axis. Any z movement is added to the calculated current.

The current set when the Node is stopped (not above the ground [z = 0]) is zero amps, and is not configurable.

The current set when the Node is hovering is fixed and may be set through the `HoverCurrent` attribute.

The model connects to the `CourseChange` trace in the MobilityModel aggregated onto the Node passed to `ConnectMobility ()` or `Init ()`. When the attached MobilityModel notifies this model of a course change this model queries the current velocity and position of the Node. The model will use the velocity and position to determine the state this model should be in (e.g. Ascending, Hovering, Stopped). Each state will incur a different current draw from the attached EnergySource. This model then notifies the attached EnergySource of this new current via `UpdateEnergySource ()`.

When the attached EnergySource depletes, this model will be notified and the `EnergyDepleted` trace will be called. However, there is no default behavior beyond this. So, if you want your Node to do something when the attached EnergySource depletes (i.e. Crash to the ground) then your program must connect to the `EnergyDepleted` trace and implement the desired behavior. See the example for how to go about this.

The current for ascending, descending, and moving is calculated based on the speed at which the Node is moving (except for descending which is based on the inverted speed [1/speed]) multiplied by a conversion factor. Each mode has its own conversion factor and is configurable through the `AscendEnergyConversionFactor`, `DescendEnergyConversionFactor`, and `MoveEnergyConversionFactor` attributes respectively.

Attributes

The model exposes the following attributes:

- **AscendEnergyConversionFactor:** The conversion factor applied to the speed of the Node when it is ascending. In $\frac{A}{m/s}$.
- **DescendEnergyConversionFactor:** The conversion factor applied to the inverted speed of the Node when it is descending. In $\frac{A}{(m/s)^{-1}}$.
- **MoveEnergyConversionFactor:** The conversion factor applied to the speed of the Node when it is moving. In $\frac{A}{m/s}$.
- **HoverCurrent:** The fixed amperage set when the Node is hovering. In amperes.
- **EnergySource:** The energy source this model draws from.

Trace sources

The model exposes the following trace sources:

- **EnergyDepleted:** Called when the attached Energy Source has indicated that it is depleted.
- **EnergyRecharged:** Called when the attached Energy Source has indicated that it has been recharged.
- **TotalEnergyConsumption:** The total energy consumed by the model. In Joules.
- **State:** The current mobility state detected by the model along with the velocity. The possible states may be found in `ns3::psc::UavMobilityEnergyModel::State`
- **Current:** The current draw from the attached energy source, along with the velocity.

1.4.2 Tests

The following tests have been written for this model and may be found in `src/psc/test/uav-mobility-energy-model-test.cc`:

- **`UavMobilityEnergyModelTestCaseEnergyConsumption`:** Tests if the model correctly sets the current on the EnergySource as well as if the correct amount of energy is deducted from the EnergySource.
- **`UavMobilityEnergyModelTestCaseZeroEnergyCost`:** The model supports making any of the possible states consume no energy. Configure each state as such and make sure no energy is consumed.
- **`UavMobilityEnergyModelTestCaseMoveEnergy`:** Move combines the cost of the 2D movement (x, y) and the ascend/descend cost (z). Tests that when a Node has both 2D movement and is ascending that the ascend/descend cost is accounted for and is correct, along with the 2D movement only.
- **`UavMobilityEnergyModelTestCaseLowEnergyTrace`:** Tests if the `EnergyDepleted` trace is actually called.
- **`UavMobilityEnergyModelTestCaseLiIonEnergySource`:** Make sure the model actually draws energy from another EnergySource that is not the BasicEnergySource.
- **`UavMobilityEnergyModelTestCaseMobilityModel`:** Tests that the model correctly integrates with a MobilityModel. Tests that the MOVE state is correctly determined from the attached MobilityModel
- **`UavMobilityEnergyModelTestCaseChangeCostsRuntime`:** The model supports changing the cost of any of the states (except for STOP) while the simulation is running. Tests if this feature actually works.
- **`UavMobilityEnergyModelTestCaseTotalEnergyConsumption`:** The model tracks the total Joules consumed over the lifetime of the model. This total is available through both the `GetTotalEnergyConsumption ()` and through the `TotalEnergyConsumption` trace source. Validate this total is correct from both sources.
- **`UavMobilityEnergyModelTestCaseInitialVelocity`:** Tests if a velocity set before the simulation starts works correctly.
- **`UavMobilityEnergyModelTestCaseState`:** Tests if the model correctly sets the state and that the `State` trace has the correct state.
- **`UavMobilityEnergyModelTestCaseTraceCurrent`;** Tests if the model correctly sets the current and that the `Current` trace has the correct current.
- **`UavMobilityEnergyModelTestCaseAttributeEnergySource`:** Tests if setting the EnergySource through the `EnergySource` attribute works properly.
- **`UavMobilityEnergyModelTestCaseFixedStateChangeSpeed`:** Assures that if the speed/velocity at which the Node is moving is changed, but the state does not (i.e. moving, then moving faster in the same direction) that the current is updated (and the `Current` trace notified) and the state remains the same (and the `State` trace is not notified)
- **`UavMobilityEnergyModelTestCaseVelocityTraces`:** Several of the traces have the velocity from the MobilityModel included. Validate that the included velocities are correct.

1.4.3 Usage

The model may be installed on any Node with a Mobility Model installed. The helper may also be used for a simpler installation. The helper will create any of the necessary components on the Node and install and init the model.

Each Node requires exactly one Energy Source passed to `SetEnergySource()` and an aggregated MobilityModel when it is passed to `ConnectMobility ()` when using the low-level API. Alternatively, both the EnergySource

and Node may be passed to `Init ()` for convenience. If the helper is used, it allows for a passed Energy Source, or an Energy Source aggregated onto the Node.

1.5 Intel HTTP Model

The classes `ns3::psc::IntelHttpClient`, `ns3::psc::IntelHttpHeader`, and `ns3::psc::IntelHttpServer` implement a configurable HTTP model.

This model is an implementation of the model defined in this Intel paper: A NEW TRAFFIC MODEL FOR CURRENT USER WEB BROWSING BEHAVIOR

1.5.1 Model Description

The implementation is provided in the following files:

- `src/psc/model/intel-http-client.{h,cc}` The client application
- `src/psc/model/intel-http-header.{h,cc}` The header used in packets
- `src/psc/model/intel-http-server.{h,cc}` The server application
- `src/psc/helper/intel-http-helper.{h,cc}` The helper

An example may be found in: `src/psc/examples/example-intel-http.cc`.

This model represents a configurable HTTP client and corresponding HTTP server. The model is based on analysis of logs from ten HTTP Squid proxy caches in the 2007 timeframe (HTTP/1.0 and HTTP/1.1 clients). The log data included requests from 60,000 clients.

In the following description, names containing an 'Rvs' suffix denote names of an `ns3::RandomVariableStream` object. Unless otherwise specified, sizes are in Bytes.

The HTTP client generates a request size from `RequestSizeRvs` and sends a request to the server defined by `RemoteAddress` and `RemotePort`.

The server receives the request and generates a size for the response page from `HtmlSizeRvs` as well as the number of embedded objects referenced on generated page. The 'page' is then transferred back to the client.

Once the client receives the page, it will wait a time defined by `ParseTimeRvs` to 'parse' the received page. Once parsed, the model goes over each embedded object and checks against the probability that it is cached defined by `CacheThreshold`.

If the embedded object is not cached the client will generate another request size (from `RequestSizeRvs`). The server generates Inter-Arrival Time for the embedded object from `EmbeddedObjectIatRvs` and waits that time, generates a size for the object from `EmbeddedObjectSizeRvs` and sends that response. The client then parses that response.

Once all embedded objects have been received, the client will read the completed page for an amount of time defined by `ReadTimeRvs`.

All packets carry an application header, the `IntelHttpHeader`. This header has 3 fields. First, a `Type` field describing whether the packet refers to an HTML object (the main object) or an embedded object. Next, a numeric field (`NumberOfObjects`) with variable meaning, depending on the type of the packet:

- In an HTML Object Request, the field is meaningless.
- In an HTML Object Response, the field details how many embedded objects are associated with the HTML Object served.

- In an Embedded Object Request, the field indicates the sequence number of the embedded object requested within the list of embedded objects for the last HTML object requested.
- In an Embedded Object Response, the field indicates the sequence number of the embedded object served within the list of embedded objects for the last HTML object requested.

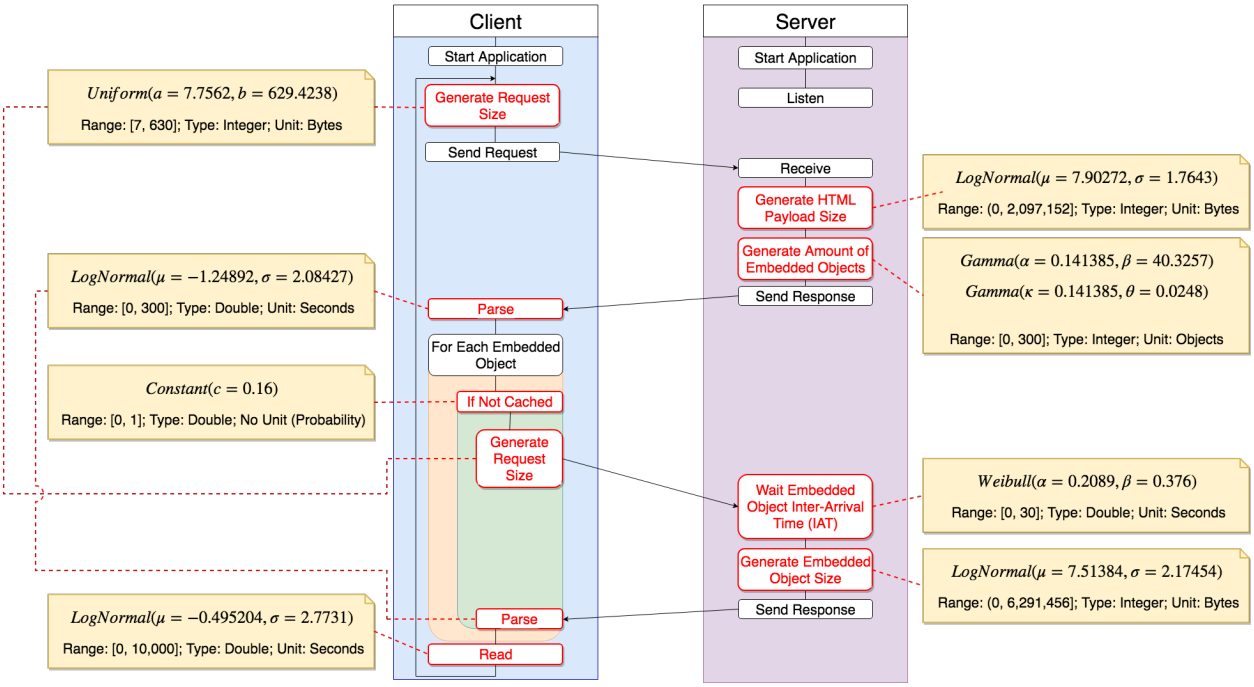
Finally, a `Payload Size` field, with the size of the Request message (for Requests) or the size of the object served (for Responses).

Limitations

The `ns3::psc::IntelHttpServer` may only serve one `ns3::psc::IntelHttpClient` at a time.

This restriction may be circumvented by installing several `ns3::psc::IntelHttpServer` applications on one Node on different ports. See the `MultipleServerClients` example in `src/psc/examples/example-intel-http.cc` for a demonstration of one server Node serving many client nodes.

Model Flow Diagram



Attributes

IntelHttpClient

- **`RemoteAddress:`** The address of the server to communicate with.
- **`RemotePort:`** The port on the server to send to.
- **`CacheThreshold:`** Threshold to decide if an embedded object is cached or not. If the value generated by `CacheRvs` is less than or equal to this value, then the object was cached and will not be requested.
- **`CacheRvs:`** Stream that generates the value compared with the `CacheThreshold` to decide if an embedded object is cached or not. Should generate a value between 0 and 1.

- **RequestSizeRvs:** Random Variable Stream to generate the size of the requests.
- **ParseTimeRvs:** Random Variable Stream to generate the time required to parse a downloaded file. Bound by `ParseTimeLowBound` and `ParseTimeHighBound`.
- **ReadTimeRvs:** Random Variable Stream to generate the time spent reading a page after it has been downloaded. Bound by `ReadTimeLowBound` and `ReadTimeHighBound`.
- **SocketBufferSize:** Size in Bytes of the socket send and receive buffers.

IntelHttpHeader

`IntelHttpHeader` exposes no attributes.

IntelHttpServer

- **Port:** Port on which we listen for incoming packets.
- **HtmlSizeRvs:** Random Variable Stream to generate the size of the HTML pages. Bound by `HtmlSizeLowBound` and `HtmlSizeHighBound`.
- **EmbeddedObjectAmountRvs:** Random Variable Stream to generate the amount of embedded objects. Bound by `EmbeddedObjectAmountLowBound` and `EmbeddedObjectAmountHighBound`.
- **EmbeddedObjectIatRvs:** Random Variable Stream to generate the IAT of embedded objects. Bound by `EmbeddedObjectIatLowBound` and `EmbeddedObjectIatHighBound`.
- **EmbeddedObjectSizeRvs:** Random Variable Stream to generate the size of the embedded objects. `EmbeddedObjectSizeLowBound` and `EmbeddedObjectSizeHighBound`.

Trace sources

IntelHttpClient

- **Rx:** General trace for receiving an application packet of any kind. If the application packet was fragmented, only one line will be traced when the last fragment is received. This trace source is invoked for all application packets received (i.e., all the Responses). This source makes available the received application packet (with the `IntelHttpHeader`) and the address of the node that sent the packet. As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding transmission trace.
- **RxMainObject:** Trace for when an application-level packet containing a main object is fully received. This source is invoked once for each Response with an HTML object (main object). If the Response message was fragmented, the trace will be invoked only when the final fragment has been received. This source makes available the received application packet (with the `IntelHttpHeader`) and the address of the node that sent the packet. As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding transmission trace.
- **RxEmbeddedObject:** Trace for when an application-level packet containing an embedded object is fully received. This source is invoked once for each Response for an embedded object. If the Response message was fragmented, the trace will be invoked only when the final fragment has been received. This source makes available the received application packet (with the `IntelHttpHeader`) and the address of the node that sent the packet. As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding transmission trace.

- **Tx:** General trace for sending an application packet of any kind. This trace source is invoked for all application packets sent (i.e., all the Requests). This source makes available the application packet sent (with the `IntelHttpHeader`). As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding reception trace.
- **TxRequestMainObject:** Trace for when an application-level packet containing a Request for a main object is sent. This source is invoked once for each Request with an HTML object (main object). This source makes available the transmitted application packet (with the `IntelHttpHeader`). As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding reception trace.
- **TxRequestEmbeddedObject:** Trace for when an application-level packet containing a Request for an embedded object is sent. This source is invoked once for each Request with an embedded object. This source makes available the received application packet (with the `IntelHttpHeader`). As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding reception trace.
- **CacheHit:** Trace called when object on a page was cached and does not need to be transmitted.
- **CacheMiss:** Trace called when object on a page was not cached and needs to be requested.

IntelHttpServer

- **Rx:** General trace for receiving an application packet of any kind. If the application packet was fragmented, only one line will be traced when the last fragment is received. This trace source is invoked for all application packets received (i.e., all the Requests). This source makes available the received application packet (with the `IntelHttpHeader`) and the address of the node that sent the packet. As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding transmission trace.
- **RxRequestMainObject:** Trace for when an application-level packet containing a Request for a main object is fully received. This source is invoked once for each Request with an HTML object (main object). If the request message was fragmented, the trace will be invoked only when the final fragment has been received. This source makes available the received application packet (with the `IntelHttpHeader`) and the address of the node that sent the packet. As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding transmission trace.
- **RxRequestEmbeddedObject:** Trace for when an application-level packet containing a Request for an embedded object is fully received. This source is invoked once for each Request for an embedded object. If the Request message was fragmented, the trace will be invoked only when the final fragment has been received. This source makes available the received application packet (with the `IntelHttpHeader`) and the address of the node that sent the packet. As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding transmission trace.
- **Tx:** General trace for sending an application packet of any kind. This trace source is invoked for all application packets sent (i.e., all the Responses). This source makes available the application packet sent (with the `IntelHttpHeader`). As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding reception trace.
- **TxMainObject:** Trace for when an application-level packet containing a main object is sent. This source is invoked once for each Response with an HTML object (main object). This source makes available the transmitted application packet (with the `IntelHttpHeader`). As the `IntelHttpHeader` is included,

trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding reception trace.

- **`TxEmbeddedObject`:** Trace for when an application-level packet containing an embedded object is sent. This source is invoked once for each Response with an embedded object. This source makes available the received application packet (with the `IntelHttpHeader`). As the `IntelHttpHeader` is included, trace sinks can make use of the `RequestType` and `NumberOfObjects` fields to match packet entries with the corresponding reception trace.

1.5.2 Tests

The following tests have been written for this model and may be found in `src/psc/test/intel-http-model-test.cc`:

- **`IntelHttpTestCaseAllCacheHits`:** Test with a perfect *CacheThreshold* that no embedded objects are requested/transmitted.
- **`IntelHttpTestCaseAllCacheMisses`:** Test with a *CacheThreshold* of 0 that all embedded are requested/transmitted.
- **`IntelHttpTestCaseCacheTotal`:** Verify the total number of cache hits/misses matches the total embedded objects. Also verifies the 'objectsLeft' parameter passed to the *CacheHit* and *CacheMiss* traces.
- **`IntelHttpTestCaseEmbeddedObjectSequence`:** Tests the, given a constant *EmbeddedObjectIatRvs*, embedded objects are both requested and transmitted in order.
- **`IntelHttpTestCaseEventOrder`:** Test that each event defined by the *Model Flow Diagram* happens in the correct order.
- **`IntelHttpTestCaseHeaderContent`:** Verify the request types in each possible packet are of the correct type. As well as the embedded object count.
- **`IntelHttpTestCaseIpv6`:** Verify the model still works when providing IPv6 addresses instead of IPv4 ones.
- **`IntelHttpTestCaseObjectInterArrival`:** Verify that, after receiving a request for an embedded object, the server will wait the time from the *EmbeddedObjectIatRvs* before responding.
- **`IntelHttpTestCaseParseTime`:** Test the the client waits the time from the *RequestSizeRvs* before continuing.
- **`IntelHttpTestCaseReadTime`:** Test that the client waits the time from *ReadTimeRvs* before requesting another main object.
- **`IntelHttpTestCaseTxRxCountSizes`:** Verify the specific transmit (i.e., *TxMainObject*) and receive (i.e., *RxEmbeddedObject*) traces are called the same number of times as the general ones (*Tx*/*Rx*). Also verifies the packets of each type are of the correct size.

1.5.3 Usage

A simple usage example is provided in: `src/psc/examples/example-intel-http.cc`.

The following attributes are exposed for users to change but are set to defaults that match the Intel cache logs, so in practice, most users will not change these values:

- `HtmlSizeRvs`, `HtmlSizeLowBound`, and `HtmlSizeHighBound`.
- **`EmbeddedObjectAmountRvs`, `EmbeddedObjectAmountLowBound`,** and `EmbeddedObjectAmountHighBound`.

- **EmbeddedObjectIatRvs, EmbeddedObjectIatLowBound,** and EmbeddedObjectIatHighBound.
- **EmbeddedObjectSizeRvs, EmbeddedObjectSizeLowBound,** and EmbeddedObjectSizeHighBound.
- RequestSizeRvs.
- ParseTimeRvs, ParseTimeLowBound, and ParseTimeHighBound.
- ReadTimeRvs, ReadTimeLowBound, and ReadTimeHighBound.

On the other hand the following attributes are more likely to be changed by users:

- **RemoteAddress, RemotePort, Port:** Besides the basic configuration match between client and server regarding the connection address and port, as the server model only supports a single client, when configuring scenarios with multiple clients special attention must be paid to ensure there are enough servers for all the clients, and that each of them use a unique connection string (address and port).
- **CacheThreshold, and CacheRvs:** Intel's whitepaper defines the cache mechanism but it does not specify the parameters (threshold or distribution to use). Based on this, the model provides default values for both parameters (CacheThreshold and CacheRvs) that are functional (a 16 % cache hit ratio on the client fits a small cache size with diverse (i.e., multiple different sites) browsing). However, these values may not fit all use cases (e.g., local caches in corporate environments where browsing is restricted to intranet pages that share the same structure, images, etc. can easily reach 90 % hit ratios).
- **SocketBufferSize:** The models configure a socket buffer size set to the maximum size by default (2,147,483,647 Bytes), to avoid the case where the server generates embedded objects with sizes larger than the available socket size. The maximum buffer size allows us to avoid dropping these packets and distorting the distribution configured by the user (or proposed by Intel). However, depending on the area of study and the interests of the users running the simulations, this may lead to unrealistic results, so the parameter is available for configuration as needed (with the caveat that the user will need to ensure that the distribution chosen for the Embedded Object sizes does not generate values that may overflow the buffer).

1.6 Video Streaming Model

The class ns3::psc::PscVideoStreaming implements the model of a live video streaming application over UDP. The class ns3::psc::PscVideoStreamingDistributions provides the data distributions for some examples of video streaming with specific parameters.

1.6.1 Model Description

The implementation is provided in the following files:

- src/psc/model/psc-video-streaming.{h,cc} The video streaming application
- src/psc/model/psc-video-streaming-distributions.{h,cc} Data distributions for sample video streams

An example may be found in: src/psc/examples/example-video-streaming.cc.

The video streaming application is modeled using statistical approach, in which the traffic generated by the streaming application is characterized by two independent variables that represent the size of the generated packets, and the interval between said packets. These functions are characterized by the Cumulative Distribution Function (CDF) that captures the distribution of values for a given type of video stream. Therefore, a video stream can be defined by just the two CDFs representing the packet size and inter-packet interval. This simple representation makes it easy to extend if needed the data distributions provided as examples.

Different types of video streams will be modeled through different CDFs, so it is important to understand the characteristics of the video stream being modeled in order to assess if any of the provided sample CDFs are a good match for

a specific scenario. Some of the aspects that may affect the data distribution are, for example, the video codec used, video resolution, frame rate, brightness of the scene, and amount of movement in the scene.

If new data distributions are needed, the model implementation makes it easy to load the data files from text files, enabling the simulation of videos with characteristics different of those provided by the default distributions.

Additionally, while the model is designed to only work with data distributions of videos with the same characteristics, it is easy to model complex streams with multiple scenes or changing streaming parameters by updating the data distributions used by the model dynamically while a simulation is running. By doing this, it is possible to model complex behaviors by capturing the data distributions of simple video streams, and then combining these simple pieces to build complicated and dynamic behaviors.

The model loads the raw data of the CDFs into two maps: `m_sizeDistribution` for the distribution of the packet size, and `m_intervalDistribution` for the inter-packet interval. If we use one of the provided sample distributions in `src/psc/model/psc-video-streaming-distributions.{h,cc}` we can load the values by using the attribute `Distribution` with one of the following labels:

- "1080p-bright" for H.264 encoded videos at 1920x1080, 60 frames per second, Constant Rate Factor 15, and bright scenes
- "1080p-dark" for H.264 encoded videos at 1920x1080, 60 frames per second, Constant Rate Factor 15, and dark scenes
- "720p-bright" for H.264 encoded videos at 1280x720, 30 frames per second, Constant Rate Factor 25, and bright scenes
- "720p-dark" for H.264 encoded videos at 1280x720, 30 frames per second, Constant Rate Factor 25, and dark scenes

If we want to load our own data distribution, we can do so by calling `ReadCustomDistribution` with the paths to the text files with the CDF for packet size and inter-packet interval as arguments. These files expected to have the following specific format:

- The first line will specify the fixed interval between the probabilities in the CDF.
- The next lines will only contain the values for the probabilities in ascending order, starting at 0 and increasing by the increment read in the first line, up to 1.

The fixed probability increment is a way of ensuring that the CDF size is kept to a reasonable size, as these distributions tend to have long tails that would increase the amount of entries (and therefore, memory and processor cycles used during the simulation) without actual significance for the size or interval.

An example of a custom CDF file for packet size is as follows:

```
0.05
83719
83725
177583
190703
196828
199767
201894
203689
205419
207177
208978
210626
212399
214640
219810
229150
```

(continues on next page)

(continued from previous page)

```
237063
240963
244952
253338
417404
```

This data file corresponds the distribution shown in Figure *Plot of the sample CDF*:

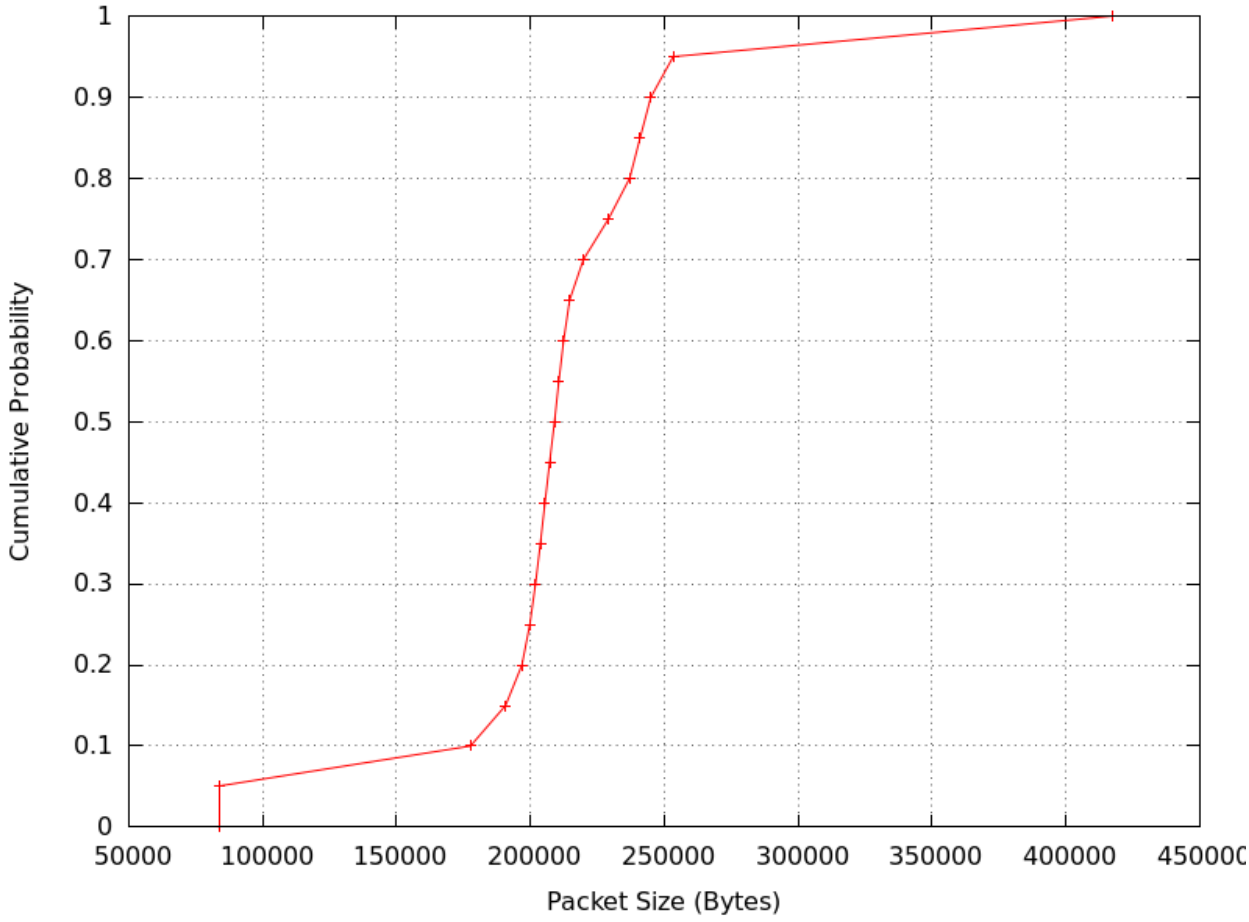


Fig. 7: Plot of the sample CDF

Once the data distributions have been loaded into the maps, the method `LoadCdfs` creates internal random variables of type `ns3::EmpiricalRandomVariable` for each of the distributions. This method is called by the `StartApplication` method, but it can be invoked by other classes to reload the data distributions.

Once the simulation starts, the model will generate a random packet size from the `EmpiricalRandomVariable` (which internally uses a uniform random number to generate a probability, and linear interpolation between the two closest provided values to compute the value associated with the probability generated). If the packet exceeds the size indicated in the attribute `MaxUdpPayloadSize` the packet will be fragmented. Once all the fragments have been sent to the UDP socket, the application waits an interval generated by the inter-packet interval `EmpiricalRandomVariable` before sending another packet.

One final feature of the models is that it is possible to configure a period of time at the beginning of the application run (called "Boost Time" in the code) in which the demand of the model is increased. This behavior mimics some streaming applications sending high amounts of data in the first seconds of the stream in order to build a buffer in the

client that will provide some protection against unexpected variation of the delay. In the model this is done by only choosing packet sizes of a selected top percentile, and inter-packet intervals of the same bottom percentile (therefore, choosing only between the largest packet sizes with the smallest inter-packet intervals, thus boosting the demand). Once a specified number of packets has been sent with this boosted demand, the complete distributions are used for both packet size and inter-packet interval. To configure this "Boost" period the model provides two attributes:

- `BoostLengthPacketCount` Length of the boosted demand, in packets.
- `BoostPercentile` Top (or bottom) percentile (from 0 to 100) to choose packet sizes and intervals from during the "Boost Time".

Limitations

Currently, custom CDFs must be defined with fixed probability steps. The model can be easily updated to overcome this limitation by interested users.

Changing the data distributions takes effect immediately, which may cause an abrupt change in the application demand, instead of an expected smooth transition.

Attributes

PscVideoStreaming

- **`Distribution:`** Label in `ns3::psc::PscVideoStreamingDistributions` that selects the data distribution to be used by the model
- **`ReceiverAddress:`** Address of the node receiving the data packets
- **`ReceiverPort:`** UDP port of the application receiving the data packets
- **`BoostLengthPacketCount:`** Length of the Boost Time expressed as number of packets sent. Set to 0 to indicate no Boost Time is desired.
- **`BoostPercentile:`** CDF Percentile, between 0 and 100, from which to draw packet sizes and inter-packet intervals during Boost Time. A BoostPercentile of 90 means that for the duration of Boost Time, packet sizes will be selected from the 10 % largest values, while inter-packet intervals will be selected from the 10 % smallest values.
- **`MaxUdpPayloadSize:`** Maximum payload size for the UDP packets transmitted. If a packet larger than this attribute is generated, it will be fragmented in pieces with, at most, this payload.

Trace Sources

PscVideoStreaming

- **`Tx:`** General trace for sending an application packet of any kind. This trace source is invoked for all packets passed to the Socket. Therefore, if a packet is fragmented, this trace source will be invoked once for each fragment.

1.6.2 Usage

A simple usage example is provided in `src/psc/examples/example-video-streaming.cc`.

The data distribution to be used can be loaded from one of the provided distributions throught the `Distribution` Attribute:

```
Ptr <PscVideoStreaming> streamingServer = CreateObject<PscVideoStreaming>();
streamingServer->SetAttribute("Distribution", StringValue("1080p-bright"));
```

Alternatively, the distribution data can be loaded from text files (one for the size distribution, one for the inter-packet interval distribution):

```
Ptr <PscVideoStreaming> streamingServer = CreateObject<PscVideoStreaming>();
streamingServer->ReadCustomDistribution(sizeFilePath, intervalFilePath);
```

Once the application starts, the data that was loaded will be used to create some `EmpiricalRandomVariables` for use in the `Send` method. To change the data distribution used (for example, to model the change from a bright scene into a dark scene), we will update the data distribution using one of the methods described above, and then we will invoke the `LoadCdfs` method to signal that the new data should be used:

```
Ptr <PscVideoStreaming> streamingServer = CreateObject<PscVideoStreaming>();
...
streamingServer->ReadCustomDistribution(sizeFilePath, intervalFilePath);
streamingServer->LoadCdfs();
```

1.7 PSC Application Model

The PSC application model provides a simple and flexible way to generate traffic for various use cases using a client/server concept. Specifically, it uses a Request-Response design where the client sends request messages to the server that, if enabled, replies with response messages.

1.7.1 Model Description

The client generates requests following an on-off model, where "on" periods are called sessions. Random variables are associated with the inter-packet interval, number of packets in a session, and the inter-session interval.

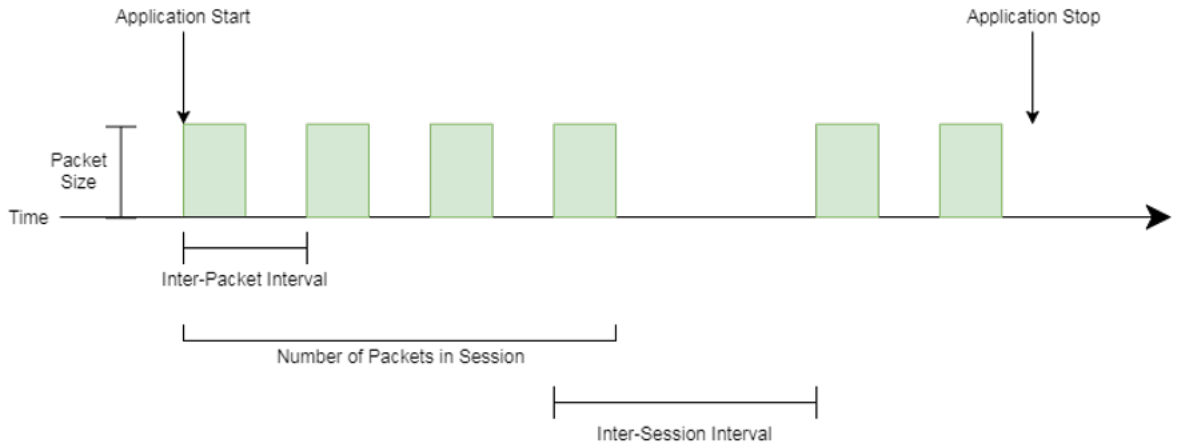


Fig. 8: Client application model

On the server side, a response is generated as soon as a request arrives unless the previous response was sent within the minimum inter-response time, in which case it is delayed. The server application is thus configured using random variables for the packet size and minimum inter-response time.

The implementation is provided in the following files:

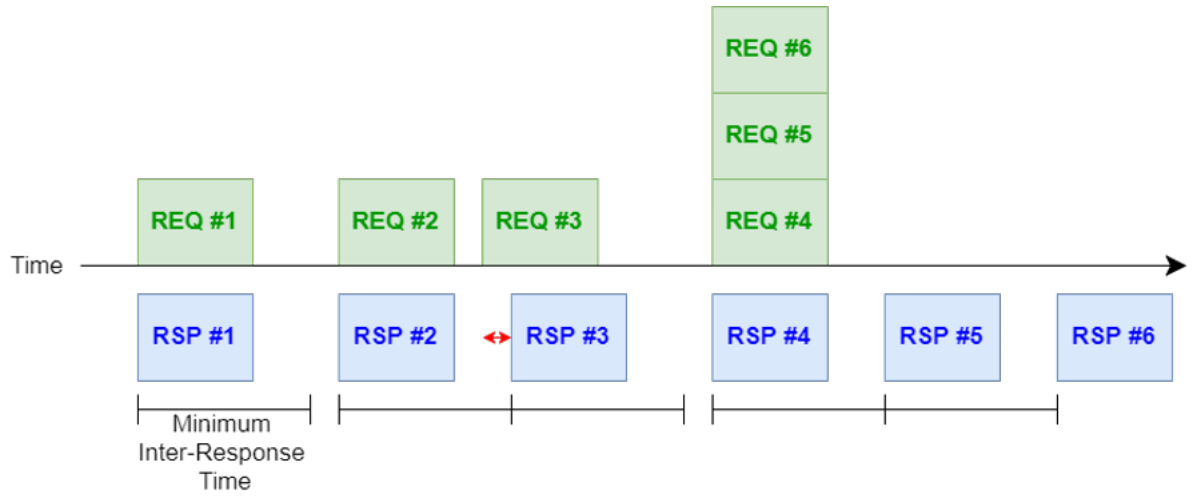


Fig. 9: Server application model

- `src/psc/model/psc-application-configuration.{h.cc}` The parameters used by the helper to configure the client and server applications.
- `src/psc/model/psc-application.{h.cc}` Super class for the client and server implementation.
- `src/psc/model/psc-application-client.{h.cc}` The client application sending requests.
- `src/psc/model/psc-application-server.{h.cc}` The server application receiving requests and sending responses.

Note: The application is able to run over any transport protocol, e.g., UDP or TCP.

Additionally, a simple example is provided at `src/psc/examples/example-psc-application.cc`.

Helpers

One helper, `ns3::psc::PscApplicationHelper`, has been created to facilitate the configuration of the applications in both the client and the server. Currently, the API includes a single function Install that takes as parameters the application configure (i.e., name and traffic parameters), the client and server nodes, server address, whether the server should send a response, and the start and stop times of the application. During the install procedure, the helper will configure the name of the application for each node by appending the instance number and the extension "_Client" or "_Server" depending on the role of the node. The helper uses an internal variable to the application configuration object to determine the number of times the application has been installed. This is convenient when the same application is deployed for several pairs of nodes. The helper will also configure the random variables in each node associated with the packet sizes, packet intervals, session duration, and time between sessions. The install function will also set the start and stop times of both the client and server, with the server starting 1 s before the client and stopping 1 s after the client in order to limit situations where packets are lost because the server is not ready. The helper returns an ApplicationContainer containing first the client then the server.

Note: The helper currently only works with IPv6 networks.

Attributes

PscApplicationConfiguration

PscApplicationConfiguration exposes no attribute.

PscApplication

- `AppName`: string representing the name of the application.
- `Protocol`: TypeId of the protocol to be used (e.g., UDP).

PscApplicationClient

- `PacketSize`: Packet size (in bytes) for the requests sent by the client to the server.
- `RemoteAddress`: Address of the server.
- `RemotePort`: Port number of the server application.
- `PacketsPerSession`: Random variable used to select the number of packets to transmit within a session.
- `PacketInterval`: Random variable used to select the interval of time between two packets.
- `SessionInterval`: Random variable used to generate the interval between sessions.

PscApplicationServer

- `Port`: Listening port to receive packets.
- `PacketSize`: Size of the packets (in bytes) for the responses sent to the client.
- `MinResponseInterval`: Minimum interval between consecutive response packets.
- `IsSink`: Flag to indicate where the server is a sink (i.e., does not send responses to the client) or not.

Trace sources

PscApplication

- `Tx`: General trace for packets transmitted by the application. Each trace includes the name of the application and the SeqTsSizeHeader, allowing tracking of the sequence number, time stamp, and packet size.
- `Rx`: General trace for packets received by the application. Each trace includes the name of the application and the SeqTsSizeHeader, allowing tracking of the sequence number, time stamp, and packet size.
- `Times`: General trace to provide the start and stop times of the application. Each trace includes the name of the application, the start time, and the stop time. The trace is called every time the application is started or stopped.

1.7.2 Usage

There is one example in the `psc/example/example-psc-application.cc`.

In that example, two nodes, n0 and n1 are connected via a Carrier Sense Multiple Access (CSMA) link with a 100 Mb/s data rate and 1 ms delay.

```
//
// n0 ==================  n1
//    LAN 6001:db80::/64
```

By running with the existing defaults, the program will configure n0 as the client and n1 as the server. The client is configured with "on" session durations of 10 s with 5 s between sessions. In each session, the client will send 10 requests of 50 Bytes at 1 s interval. For each request, the server will send a response of 20 Bytes.

The configuration and deployment of the PscApplication instances is done in two steps:

- Creation and configuration of a PscApplicationConfiguration object as shown below:

```
Ptr<PscApplicationConfiguration> appConfig =
    CreateObject<PscApplicationConfiguration>("PscApplicationExample",
        UdpSocketFactory::GetTypeId(), // Socket type
        5000                           // port number
    );

appConfig->SetApplicationPattern(
    CreateObjectWithAttributes<ConstantRandomVariable>(
        "Constant", DoubleValue(10)), // Number of packets to send per session
    CreateObjectWithAttributes<ConstantRandomVariable>(
        "Constant", DoubleValue(1)), // Packet interval (in s)
    CreateObjectWithAttributes<ConstantRandomVariable>("Constant",
        DoubleValue(5)), // Time between sessions
    50,  // Client packet size (bytes)
    20); // Server packet size (bytes)
```

- Deployment of the application in the client and server nodes, which can easily be done using an instance of PscApplicationHelper as show below:

```
Ptr<PscApplicationHelper> appHelper = CreateObject<PscApplicationHelper>();

ApplicationContainer apps = appHelper->Install(
    appConfig,
    csmaNodes.Get(0),
    csmaNodes.Get(1),
    csmaNodes.Get(1)->GetObject<Ipv6L3Protocol>()->GetAddress(1, 1).GetAddress(),
    echoClient,
    Seconds(startTime),
    Seconds(simTime));
```

The example program supports a few options to change the application behavior and the level of output. The list of arguments is listed by using the '—PrintHelp' option shown below:

```
$ ./ns3 --run 'example-psc-application --PrintHelp'

Program Options:
  --echoClient:    Set EchoClient attribute [true]
  --verbose:       Tell echo applications to log if true [true]
  --enableTraces:  Enable trace file output [false]
  --time:          Simulation time [30]
```

The "echoClient" argument controls whether the server will send responses to the client. The "verbose" argument enables debug information and requires the program to be compile with 'debug' option. The "enableTraces" lets the program connect to the application trace sources and store the packets transmitted and received by both the client and the server applications. Finally, the "time" argument controls the duration of the simulation.

By enabling the traces, the program will generate an output file called PscApplicationExample_trace.txt with the packet information. If the "echoClient" argument is set to "true" (default), the output will look like:

```
Name    Time    Action  Payload SeqNum
PscApplicationExample_0_Client 2 TX 50 1
PscApplicationExample_0_Server 2.00503 RX 50 1
PscApplicationExample_0_Server 2.00503 TX 20 2
PscApplicationExample_0_Client 2.00604 RX 20 2
PscApplicationExample_0_Client 3 TX 50 3
PscApplicationExample_0_Server 3.00101 RX 50 3
PscApplicationExample_0_Server 3.00101 TX 20 4
PscApplicationExample_0_Client 3.00202 RX 20 4
PscApplicationExample_0_Client 4 TX 50 5
PscApplicationExample_0_Server 4.00101 RX 50 5
...
```

If the "echoClient" argument is set to "false", the output will look like:

```
Name    Time    Action  Payload SeqNum
PscApplicationExample_0_Client 2 TX 50 1
PscApplicationExample_0_Server 2.00503 RX 50 1
PscApplicationExample_0_Client 3 TX 50 2
PscApplicationExample_0_Server 3.00101 RX 50 2
PscApplicationExample_0_Client 4 TX 50 3
PscApplicationExample_0_Server 4.00101 RX 50 3
PscApplicationExample_0_Client 5 TX 50 4
PscApplicationExample_0_Server 5.00101 RX 50 4
PscApplicationExample_0_Client 6 TX 50 5
PscApplicationExample_0_Server 6.00101 RX 50 5
...
```

1.8 Incident Scenario Framework

In [NIST2019b], guidelines were provided to help authors of scenarios document incidents with enough detail to be reproducible by different researchers using different platforms. In addition, the guidelines separated the description of the incident and the communication needs (who, what, where) from the technology (how), thus making it easier to investigate different network capabilities for the same scenario.

Similarly, the objective of the incident scenario framework presented in this section is to provide a set of classes and a methodology for developing potentially complex scenarios involving hundreds of users and a multitude of applications. The modular approach helps the users evaluate different technologies and configurations without having to start from scratch.

Note: The full scenario is large and runs for over four hours of simulation time. The scenario will trigger an assert or exception if the LTE traces feature is enabled or if asserts are enabled in the build. These are apparently due to upstream LTE issues that have not been resolved as of release v7.0. In addition, LTE traces are verbose and will consume tens of GB of file system if enabled, and will cause slower execution time. Therefore, LTE traces are disabled by default, and users are encouraged to build ns-3 in optimized mode without asserts enabled (the default setting). For example:

```
$ ./ns3 configure -d optimized --enable-examples --enable-tests
```

1.8.1 Design Overview

The methodology for implementing scenarios involves the following steps:

1. Describe the scenario: define the area where the incident occurs, including the buildings if present, all the users and their mobility, and the timing of the main events.
2. Configure the network: configure technology specific devices in the nodes. For example, deploy an LTE network and attach NetDevices to the nodes.
3. Configure the applications: deploy applications in the nodes with appropriate start and stop time based on the scenario events.

This methodology was used to implement the fictional school shooting scenario described in [NIST2021c] and whose implementation is located in `psc/example/schoolshooting`.

To assist with the description of scenarios and provide consistent information, a new class `ns3::psc::PscScenarioDefinition` was created to define and store scenario information composed of:

- Areas: an area is a zone of interest in the scenario. When an incident occurs, first responders typically define perimeters and areas where specific activities will take place (e.g., triage). Those areas are defined by a name and a rectangle.
- Structures: a structure represents a building at the location of the incident. A structure is defined by a name and a BuildingContainer to store one or more ns-3 buildings.
- Node groups: a group of nodes associated with a given role. Typical response teams are organized by roles, e.g., SWAT teams or firefighters. A group of nodes is defined by the name of the group and a NodeContainer with one or more ns-3 nodes.
- Key events: an event is used to identify the time at which a significant event occurred during the scenario, e.g., an explosion or an arrest. Those events are usually followed by actions from the first responders. An event is defined by a name and the time at which it occurs.
- Applications: a list of applications used during the incident.

The `ns3::psc::PscScenarioDefinition` stores the information using different maps with a user-provided unique ID as the key.

The implementation is provided in the following file:

- `src/psc/model/psc-scenario-definition.{h.cc}`.

Helpers

The `ns3::psc::PscScenarioTraceHelper` class defined by the files `src/psc/helper/psc-scenario-definition.{h.cc}` can be used to provide consistent output traces for any incident scenario. The traces are generated at the time they are enabled, therefore the user must define the scenario completely before attaching the helper otherwise some information may not be included in the trace files.

The name of the trace files is based on the scenario name and append a suffix based on the type of trace and the file extension ".txt".

The trace file for the areas has a name that ends with the suffix "-areas.txt" and has the following format:

```
ID  Name  xMin  xMax  yMin  yMax
```

where xMin, xMax, yMin, and yMax define the boundaries of the area.

The trace file for the structures has a name that ends with the suffix "-structures.txt" and has the following format:

```
ID  Name  xMin  xMax  yMin  yMax  zMin  zMax
```

where xMin, xMax, yMin, yMax, zMin, and zMax define the 3D boundaries of the structure (i.e., ns-3 building).

The trace file for the nodes has a name that ends with the suffix "-nodes.txt" and has the following format:

```
GroupID GroupName UID X Y Z
```

where UID is ns-3 generated node ID, and X, Y, and Z are the initial coordinates for the node.

The trace file for the events has a name that ends with the suffix "-events.txt" and has the following format:

```
ID  Name  Time
```

where Time is the time of the event.

Additionally, the helper can be used to trace the progress of the simulation by calling the 'EnableTimeTrace' method and providing an `ns3::Time` argument that specifies the time interval between updates. The simulation time is then printed into a file with the suffix "-time.txt".

Attributes

The class `PscScenarioDefinition` does not expose any attribute.

Trace sources

The class `PscScenarioDefinition` does not expose any trace source.

1.8.2 Example: School shooting

The example available in the `psc/example/schoolshooting` folder is a large scale school shooting incident involving over 100 first responders and vehicles, more than a dozen different types of applications, and that is over four hours in simulation time.

The implementation of the example is provided in the following files:

- `src/psc/examples/schoolshooting/psc-schoolshooting.cc` The main program to launch the example.
- `src/psc/examples/schoolshooting/schoolshooting-definition-helper.{h.cc}` Helper class to define the scenario.
- `src/psc/examples/schoolshooting/schoolshooting-network-technology-helper.{h.cc}` Abstract helper class that defines the API for the underlying technology helper.
- `src/psc/examples/schoolshooting/schoolshooting-lte-helper.{h.cc}` Helper class for deploying the LTE network supporting the incident that implements the underlying technology API.
- `src/psc/examples/schoolshooting/schoolshooting-application-helper.{h.cc}` Helper class to define and deploy the applications.
- `src/psc/examples/schoolshooting/schoolshooting-application-trace-helper.{h.cc}` Helper class to activate application traces.
- `src/psc/examples/schoolshooting/schoolshooting-lte-vis-helper.{h.cc}` Helper class to visualize the incident using NetSimulyzer [NIST2021b].

Helpers

SchoolShootingDefinitionHelper

The `SchoolShootingDefinitionHelper` helper is responsible for defining the school shooting scenario description. All the areas, buildings, nodes, and events are defined in this class. The helper also defines the mobility models of each node throughout the duration of the incident.

SchoolShootingNetworkTechnologyHelper

The `SchoolShootingNetworkTechnologyHelper` helper is meant to provide an API for technology specific deployment helpers. It provides virtual functions to access nodes specified in the scenario that may be implemented differently based on the technology used.

SchoolShootingLteHelper

The `SchoolShootingLteHelper` helper is a subclass of `SchoolShootingNetworkTechnologyHelper` and is responsible for deploying an LTE network and associated devices in order to provide first responders with a means of communication.

SchoolShootingApplicationHelper

The `SchoolShootingApplicationHelper` helper is the interface towards the applications. It is responsible for deploying all the applications inside the nodes. It manages the ports to be used by each application. Getter functions are also available to access the installed applications during simulation. Finally, the helper API also allows the user to enable traces for the various applications.

SchoolShootingApplicationTraceHelper

The `SchoolShootingApplicationHelper` helper is responsible for creating trace files associated with the various applications. When an application trace is enabled, the helper will connect to the model's trace sources and register callback functions to write traces to files. Because the scenario uses different types of applications, the helper functions are specific to each application model.

SchoolShootingLteVisHelper

The `SchoolShootingLteVisHelper` helper is an optional helper that generates a JSON file to replay and visualize the incident with NetSimulyzer [NIST2021b]. It connects to the scenario definition to obtain building and node information (including positions throughout the simulation), to the technology helper to obtain network information (e.g., eNodeB height), and to the application helper to connect to the applications' trace sources.

Attributes

SchoolShootingDefinitionHelper

- `CreateClassrooms`: boolean to enable the creation of ns-3 buildings representing the classrooms inside the main buildings. Due to limitations in the ns-3 propagation models with buildings inside other buildings, it

should not be used when a technology is enabled. For example, this feature can be enabled with the scenario is ran with the NetSimulyzer traces enabled and the LTE disabled.

SchoolShootingNetworkTechnologyHelper

The class `SchoolShootingNetworkTechnologyHelper` does not expose any attribute.

SchoolShootingLteHelper

- `PathLossModelType`: the path loss model to use. The default is `ns3::Hybrid3gppPropagationLossModel`.
- `SchedulerType`: the MAC scheduler. The default is `ns3::RrFfMacScheduler`.
- `DlBandwidth`: the number of RBs in the downlink. The default is 50 RBs.
- `UlBandwidth`: the number of RBs in the uplink. The default is 50 RBs.
- `DlEarfcn`: the Downlink E-UTRA Absolute Radio Frequency Channel Number (EARFCN) as per 3GPP 36.101. The default is 5330 (LTE Band 14).
- `UlEarfcn`: the Uplink E-UTRA Absolute Radio Frequency Channel Number (EARFCN) as per 3GPP 36.101. The default is 23 330 (LTE Band 14).
- `UeAntennaHeight`: the height of the UEs. The default is 1.5 m.
- `EnbAntennaHeight`: the height of the eNodeBs. The default is 30 m.
- `UeTransmissionPower`: the UE transmit power. The default is 23 dBm.
- `EnbTransmissionPower`: the eNodeB transmit power. The default is 46 dBm.
- `DefaultTransmissionMode`: the eNodeB transmission mode. The default is 2 (MIMO spatial multiplexing).
- `LinkDataRate`: the rate for the links between network elements. The default is 10 Gb/s.
- `LinkDelay`: the delay for the links between network elements. The default is 10 ms.

SchoolShootingApplicationHelper

- `Testing`: boolean to enable/disable the testing mode of the applications. If enabled, applications are configured with condensed parameters and start/stop times below 100 s.

SchoolShootingApplicationTraceHelper

The class `SchoolShootingApplicationTraceHelper` does not expose any attribute.

SchoolShootingLteVisHelper

- `Interval`: time interval between node mobility updates. Smaller values increase the size of the output file.

Trace sources

None of the helper classes exposes trace sources. However, the helpers `SchoolShootingApplicationHelper` and `SchoolShootingLteVisHelper` connect to many different trace sources exposed by the technology and application models.

Usage

The main program located in `src/psc/examples/schoolshooting/psc-schoolshooting.cc` is responsible for assembling the various components of the simulation. Thanks to the use of helper classes, the code is relatively simple.

The following code block shows the minimum setup to create a simulation based on the school shooting scenario:

```
// 1. Create the scenario
Ptr<SchoolShootingDefinitionHelper> scenarioDefinitionHelper =
    CreateObject<SchoolShootingDefinitionHelper>();

// 2. Create technology helper and initialize the network
Ptr<SchoolShootingLteHelper> lteScenarioHelper = CreateObject<SchoolShootingLteHelper>
↪();
lteScenarioHelper->SetScenarioDefinitionHelper(scenarioDefinitionHelper);
lteScenarioHelper->Initialize();

// 3. Create the application helper and
Ptr<SchoolShootingApplicationHelper> scenarioApplicationHelper =
    CreateObject<SchoolShootingApplicationHelper>();
scenarioApplicationHelper->SetTechnologyHelper(lteScenarioHelper);
scenarioApplicationHelper->SetScenarioDefinitionHelper(scenarioDefinitionHelper);

// 4. Enable traces
Ptr<PscScenarioTraceHelper> scenarioTraceHelper =
    CreateObject<PscScenarioTraceHelper>(scenarioDefinitionHelper->
↪GetScenarioDefinition());
scenarioTraceHelper->EnableScenarioTraces();
scenarioApplicationHelper->EnableApplicationTraces();
lteScenarioHelper->EnableLteTraces();
```

However, for convenience, the example program supports a few options to change the application behavior and the trace selection. The list of arguments is listed by using the'—PrintHelp' option shown below:

```
$ ./ns3 run 'psc-schoolshooting --PrintHelp'

Program Options:
  --duration:         Duration (in Seconds) of the simulation [+4.0833h]
  --enableLte:        Flag to enable LTE deployment [true]
  --enableLteTraces:  Flag to enable LTE traces [true]
  --enableGuiTraces:  Flag to enable the visualization traces [false]
  --guiResolution:    Granularity of the visualization [1000]
  --reportTime:       Indicates whether or not the simulation time is reported␣
↪periodically [+1s]
  --testing:          Indicates if using modified application times for 100 s␣
↪simulation [false]
  --appId:            The ID for the application (see schoolshooting-application-
↪helper.h::SchoolShootingApplicationId [-1]
```

The "duration" argument allows the user to control the duration of the simulation, but there is no traffic after the

incident ends at 14700 s (245 min). The "enableLte" argument determines if the LTE technology is enabled. If disabled, buildings and nodes are still created, but applications cannot run. Disabling it is useful to verify node mobility as the simulation speed drastically improves. If LTE is enabled, application traces are also enabled. The name of the scenario is also based on whether or not LTE is enabled. If LTE is disabled, the scenario name is "SchoolShootingNoLte", and if it is enabled, the name is "SchoolShootingLte", thus avoiding overwritting output files. The "enableLteTraces" controls the generation of the standard LTE traces, some of which are very large due to the simulation length. The "enableGuiTraces" and "guiResolution" enables the generation of a trace file for NetSimulyzer and the frequency of updates, respectively. Note that if NetSimulyzer is not part of the ns-3 installation, enabling the traces will be ignored. The "reportTime" argument sets the periodicity of the time trace that prints the current simulation time. This is useful to monitor the progress of the simulation which can be a couple of days depending on the machine on which the simulation is running. The "testing" argument can be used to toggle a modified version of the scenario in which applications parameters are condensed and the simulation is set to 100 s. As the name indicates, this was used for testing the scenario and detecting potential issues without running the full simulation. Finally, the "appId" argument can be used to run the simulation by enabling a single type of application as defined in the enumeration `SchoolShootingApplicationHelper::SchoolSchootingApplicationId`. This was used to validate the application configuration. Note that the behavior observed with a single application enabled does not necessarily reflect the behavior that will be observed when all the applications are enabled due to the possibility of congestion and packet loss.

In addition to the parameters exposed as command-line arguments, ns-3 allows the configuration of the default value of any class attribute through the command-line interface, and allows available classes and attributes to be printed out. See the ns-3 tutorial for more information about this.

A number of output files are generated based on the traces enabled. In addition to scenario files, application-specific traces are also generated. The number and format of the traces is dependent on the model being used. By default the following files will be created:

- Scenario information:
  - `SchoolShootingLte-structures.txt`: list of buildings and their locations.
  - `SchoolShootingLte-areas.txt`: list of areas and their boundaries.
  - `SchoolShootingLte-nodes.txt`: list of nodes including their group name and initial location.
  - `SchoolShootingLte-events.txt`: list of events and the time they occur.
  - `SchoolShootingLte-time.txt`: file tracking the simulation time.
- Application statistics:
  - `SchoolShootingLte-AutomaticVehicleLocation-appTraffic.txt`: trace of packets sent and received by the instances of the vehicle location tracking application. Generated by the `PscApplication` model.
  - `SchoolShootingLte-AutomaticVehicleLocation-appTimes.txt`: trace of start and stop times of the instances of the vehicle location tracking application. Generated by the `PscApplication` model.
  - `SchoolShootingLte-Biometrics-appTraffic.txt`: trace of packets sent and received by the instances of the biometrics application. Generated by the `PscApplication` model.
  - `SchoolShootingLte-Biometrics-appTimes.txt`: trace of start and stop times of the instances of the biometrics application. Generated by the `PscApplication` model.
  - `SchoolShootingLte-BuildingPlan-appTraffic.txt`: trace of packets sent and received by the instances of the building plans data application. Generated by the `PscApplication` model.
  - `SchoolShootingLte-BuildingPlan-appTimes.txt`: trace of start and stop times of the instances of the building plans data application. Generated by the `PscApplication` model.

- `SchoolShootingLte-DeployableCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the deployable camera. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-EmsVehicleCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the EMS vehicle camera. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-HelicopterCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the helicopter camera. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-IncidentManagement-appTraffic.txt`: trace of packets sent and received by the instances of the incident management application. Generated by the `PscApplication` model.
- `SchoolShootingLte-IncidentManagement-appTimes.txt`: trace of start and stop times of the instances of the incident management application. Generated by the `PscApplication` model.
- `SchoolShootingLte-NG911Video-appTraffic.txt`: trace of packets sent and received by the instances of the NG911 videos. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-RmsBca-appTraffic.txt`: trace of packets sent and received by the instances of the Records Management System (RMS) and Bureau of Criminal Apprehension (BCA) database application. Generated by the `PscApplication` model.
- `SchoolShootingLte-RmsBca-appTimes.txt`: trace of start and stop times of the instances of the RMS and BCA database application. Generated by the `PscApplication` model.
- `SchoolShootingLte-Satellite-appTraffic.txt`: trace of packets sent and received by the instances of the satellite images application. Generated by the `PscApplication` model.
- `SchoolShootingLte-Satellite-appTimes.txt`: trace of start and stop times of the instances of the satellite images application. Generated by the `PscApplication` model.
- `SchoolShootingLte-SchoolCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the school camera. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-SwatHelmetCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the helmet cameras. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-TacticalTelemetry-appTraffic.txt`: trace of packets sent and received by the instances of the tactical telemetry application. Generated by the `PscApplication` model.
- `SchoolShootingLte-TacticalTelemetry-appTimes.txt`: trace of start and stop times of the instances of the tactical telemetry application. Generated by the `PscApplication` model.
- `SchoolShootingLte-ThrowPhone-appTraffic.txt`: trace of packets sent and received by the instances of the throw phone voice application. Generated by the `PscApplication` model.
- `SchoolShootingLte-ThrowPhone-appTimes.txt`: trace of start and stop times of the instances of the throw phone voice application. Generated by the `PscApplication` model.
- `SchoolShootingLte-ThrowPhoneCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the throw phone camera. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-TrafficCameraVideo-appTraffic.txt`: trace of packets sent and received by the instances of the traffic camera. Generated by the `PscVideoStreaming` model.
- `SchoolShootingLte-VideoConference-appTraffic.txt`: trace of packets sent and received by the instances of the audio conference application. Generated by the `PscApplication` model.
- `SchoolShootingLte-VideoConference-appTimes.txt`: trace of start and stop times of the instances of the audio conference application. Generated by the `PscApplication` model.

  - `SchoolShootingLte-VideoConferenceVideo-appTraffic.txt`: trace of packets sent and received by the instances of the video conference application. Generated by the `PscVideoStreaming` model.
  - `SchoolShootingLte-Web-appTraffic.txt`: trace of packets sent and received by the instances of the web application. Generated by the `IntelHttp` model.
  - `mcptt-m2e-latency.txt`: MCPTT related trace containing mouth-to-ear latency. Generated by the `MCPTT` application model.
  - `mcptt-access-time.txt`: MCPTT related trace containing access time performance. Generated by the `MCPTT` application model.

- **LTE traces** Additionally, there are 13 LTE trace files to monitor the uplink and downlink transmissions at the physical layers all the way to the PDCP layers. The format of the files is described in the LTE section of the ns-3 model documentation.

If the NetSimulyzer traces are enabled, the following file is also generated:

- `SchoolShootingLte.json`: file containing visualization information for NetSimulyzer. Once loaded, the user can visualize the topology, replay the simulation, and analyze the network performance. A screenshot of the scenario loaded in NetSimulyzer is shown below.

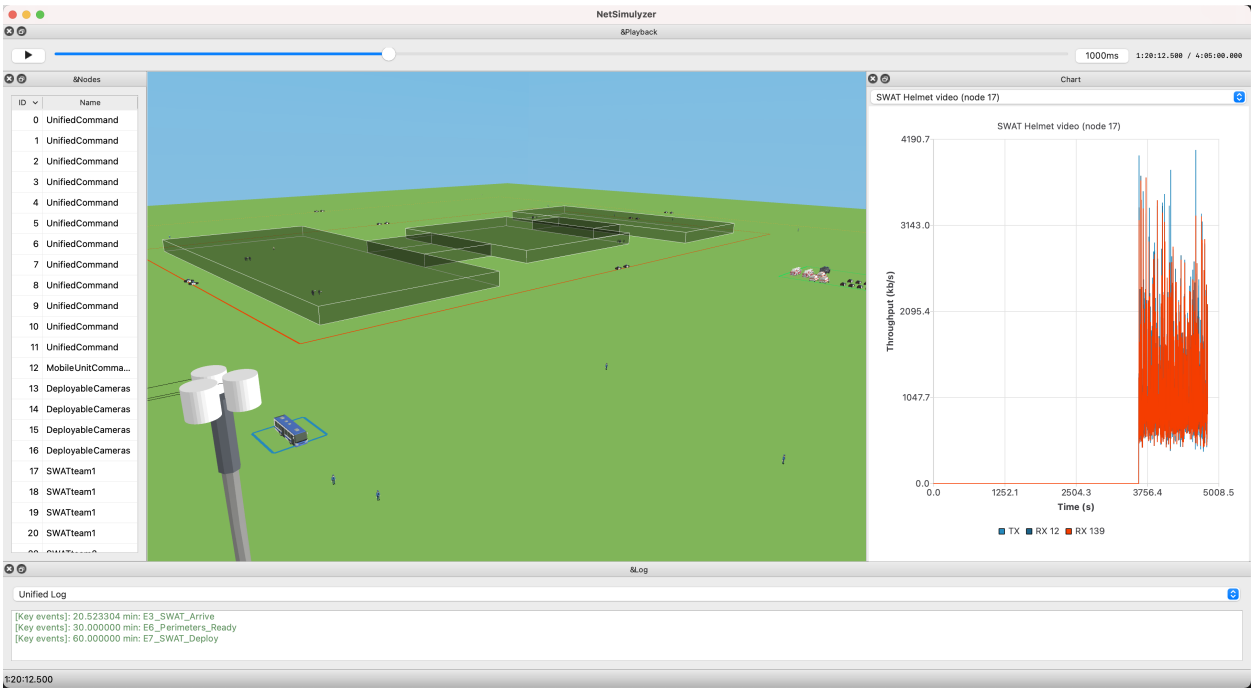


Fig. 10: Screenshot of school shooting scenario in NetSimulyzer

Extending the scenario

Even though several attributes have been exposed to change the network parameters, the physical deployment remains the same. Using the proposed design, researchers interested in evaluating different technologies (e.g., NR) or variations of the sample LTE deployment can simply extend the class `ns3::psc::SchoolShootingNetworkTechnologyHelper`. For visualization, only a small portion of the `SchoolShootingLteVisHelper` is actually technology dependent (e.g., accessing the eNodeBs), therefore the helper can easily be reused.

CHAPTER

TWO

LTE D2D MODELS

This chapter is an excerpt of the LTE module documentation chapter, containing the D2D-related models that have been added to the *ns-3* LTE module.

2.1 Design Documentation

2.2 LTE Sidelink

2.2.1 Overview

This section describes the *ns-3* support for LTE Device to Device (D2D) communication based on the model published in [NIST2016] [NIST2017] and extended to support UE-to-Network Relay operations [NIST2019].

3GPP introduced D2D Proximity Services (ProSe) in release 12 to allow ProSe enabled devices to exchange information directly, i.e., without traversing the eNB. Figure *Non-Roaming Reference Architecture* shows new ProSe entities and the new interfaces added to the network [TS23303].

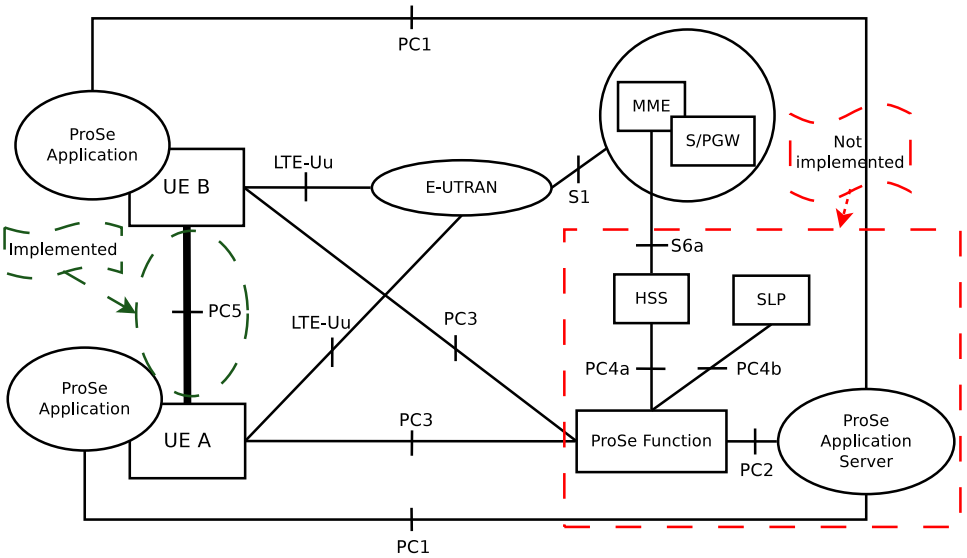


Fig. 1: Non-Roaming Reference Architecture

At the time of writing this documentation only the new radio interface, i.e., PC5 is implemented. This interface is also known as Sidelink at physical layer. The model supports all the three following LTE D2D functionalities defined under ProSe services:

1. Direct communication
2. Direct discovery
3. Synchronization

These LTE D2D functionalities can operate regardless of the network status of the UEs. Thus, four scenarios were identified by 3GPP [TR36843]:

1A. Out-of-Coverage

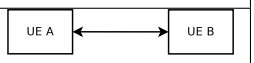
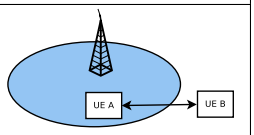
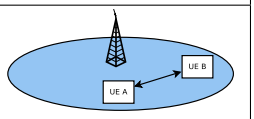
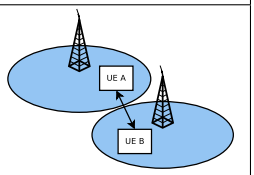
1B. Partial-Coverage

1C. In-Coverage-Single-Cell

1D. In-Coverage-Multi-Cell

At this stage, the model has been tested for the four scenarios. The Table *ns-3 LTE Sidelink supported/tested scenarios* gives more information about the support of these four scenarios for all the three ProSe services.

Table 1: ns-3 LTE Sidelink supported/tested scenarios

#	Description	UE A	UE B	Direct Communication	Direct Discovery	Synchronization	Example
1A	Out-of-Coverage	Out-of-Coverage	Out-of-Coverage	Yes RA = Mode 2	Yes RA = Type 1	Yes Autonomous synchronization	
1B	Partial Coverage Single Cell	In-Coverage	Out-of-Coverage	Yes RA = Mode 2	Yes RA = Type 1	Yes Network and autonomous synchronization	
1C	In-Coverage Single Cell	In-Coverage	In-Coverage	Yes RA = Mode 1 RA = Mode 2	Yes RA = Type 1	Yes Network synchronization	
1D	In-Coverage Multiple Cells	In-Coverage	In-Coverage	Yes RA = Mode 1 RA = Mode 2	Yes RA = Type 1	Yes Network synchronization	
RA = Resource Allocation							

The model is developed in a way that simulating the above ProSe services are not interdependent. In the following, we describe all the changes introduced in the *ns-3* LTE architecture and its protocol stack to realize Sidelink functionality.

2.2.2 Architecture

eNB architecture

There is no change in the eNB data (lte-design:ref:*fig-ca-enb-data-plane*), control (lte-design:ref:*fig-ca-enb-ctrl-plane*) plane and neither in its PHY/channel (lte-design:ref:*fig-lte-enb-phy*) model.

UE architecture

Figure *UE data plane architecture* and Figure *UE control plane architecture* show the current UE model architecture in the data plane and the control plane, respectively.

The class `LteSlUeNetDevice` was created to handle packets in the data plane when the UEs are using the UE-to-Network Relay functionality and have established a one-to-one direct communication link with another UE over the Sidelink. Each UE has an `LteSlUeNetDevice` object per active one-to-one direct communication link and the `EpcUeNas` is now in charge of filtering the data packets towards the appropriate interface.

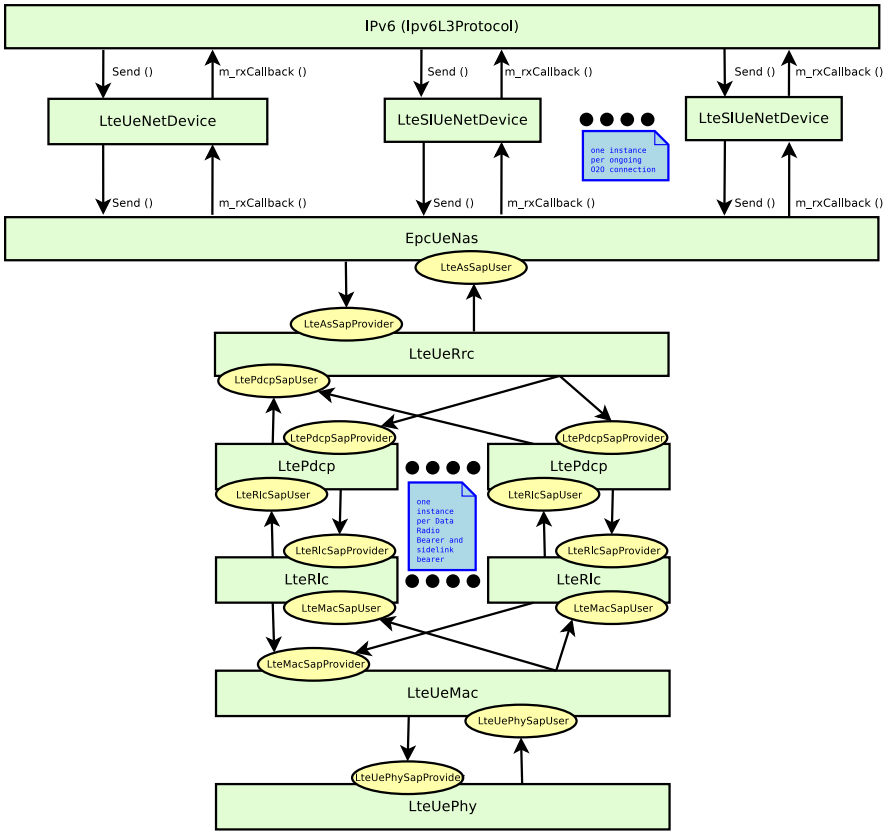


Fig. 2: UE data plane architecture

In the control plane, the class `LteSlUeController` was added to the model to provide a framework for the implementation of custom algorithms controlling the different decision processes involved in the UE-to-Network Relay functionality, e.g., Relay selection, connection setup, IP configuration, etc.. More details are provided below in the Sidelink UE Controller section.

The Sidelink uses the same spectrum than the uplink for transmission and reception. Figure *PHY and channel model architecture for the UE* shows that a new instance of the `SpectrumPhy` class is now present in the PHY/channel architecture of the UE in order to support the reception of packets transmitted by other UEs in the uplink spectrum.

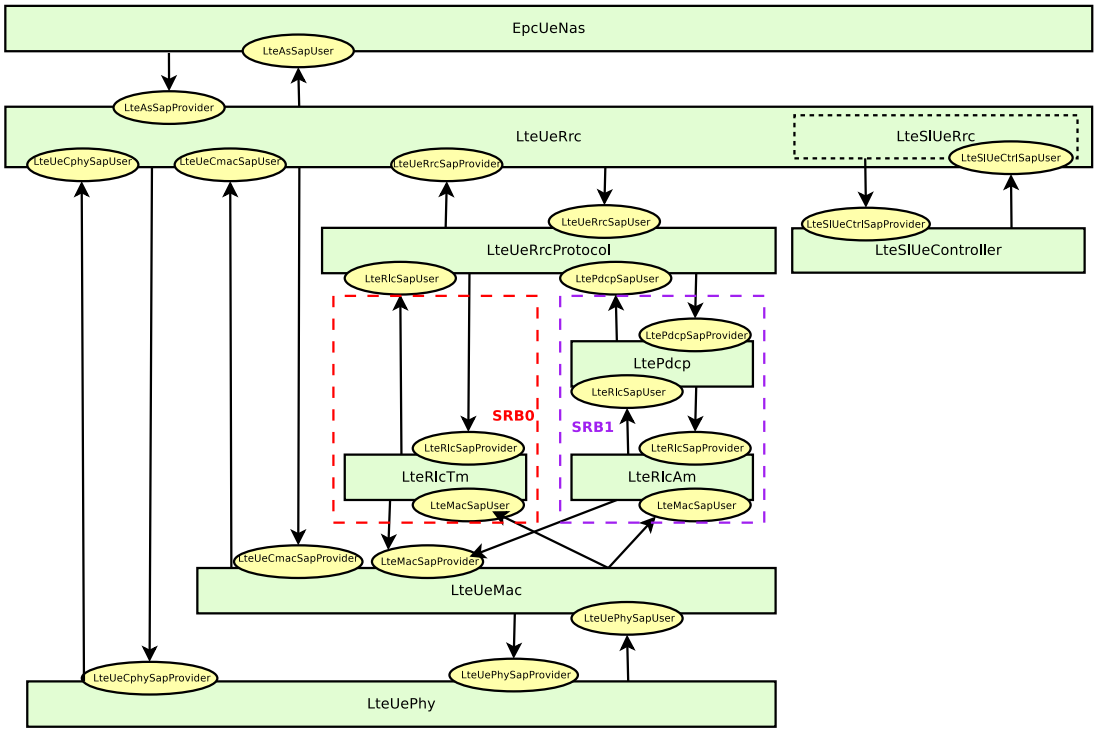


Fig. 3: UE control plane architecture

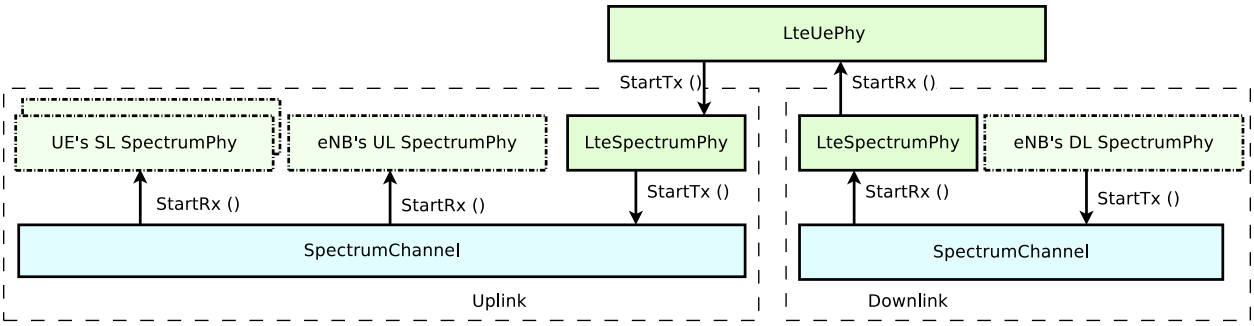


Fig. 4: PHY and channel model architecture for the UE

NAS

In LTE module, the NAS layer functionality is provided by `EpcUeNas` class. In addition to its existing capabilities, this class has been extended to support LTE ProSe services, namely direct communication, direct discovery, and UE-to-Network Relay.

For direct communication, it supports the functions to activate/deactivate Sidelink bearers. Since there is no EPS bearer for Sidelink communication, the existing TFT cannot be used to map IP packets to a Sidelink bearer. Therefore, a new type of TFT, called `LteSlTft` is implemented. It maps IP packets to the Sidelink bearers based only on their destination IP address. Moreover, the `Send` function has been extended for both UE NAS "Active" and "Off" states. In the active state, if there is a Sidelink bearer established between the source and the destination UE, it utilizes Sidelink bearer to send the packet, thus, prioritizing a Sidelink bearer over normal LTE uplink bearer. On the other hand, in off state in which previously UE was unable to send the packets, is now able to use a Sidelink bearer, if established.

The extensions for the direct communication are also useful for UE-to-Network Relay communication by helping in sending and receiving of packets for the communication between Relay UE and Remote UE. Additionally, to support public safety communication scenarios using UE-to-Network Relays, the NAS has also been extended to include the transfer of IPv6 packets. The `EpcUeNas` now uses TFTs for ingress filtering to forward the received packets to the appropriate `NetDevice`. For the Remote UE, an `LteSlTft` is created to receive any packet with the destination address matching the address generated for the Remote UE's `LteSlUeNetDevice`. For a Relay UE, an `LteSlTft` is created to receive any packet with the source address prefix matching the prefix delegated for relay purposes by the core network. Upon receiving a packet in the DoRecvData function of `EpcUeNas`, a check is initially made to verify if the Source IP address and the Destination IP address in the packet header matches the ingress `LteSlTft`. If yes, the packet is forwarded onto the appropriate sidelink callback function. Otherwise, the packet is forwarded onto the uplink callback function.

For the direct discovery, it conveys the information, e.g., list of discovery applications and the nature of the application, i.e., announcing or monitoring to `LteUeRrc` class.

RRC

The RRC layer has been modified to support all the three ProSe services. Among all the new changes, a major modification which is common in both eNB and UE RRC is the addition of two new classes called `LteSlEnbRrc` and `LteSlUeRrc`. Both the classes serve a common purpose of holding Sidelink resource pool (i.e., communication and discovery) configuration done through the functions added to the `LteHelper` class. Figures *Relationship between LteEnbRrc and LteSlEnbRrc* and *Relationship between LteUeRrc and LteSlUeRrc* show the relationship between `LteEnbRrc<-->LteSlEnbRrc` and `LteUeRrc<-->LteSlUeRrc` classes.

Note: Only the key responsibilities of the new classes are shown. For complete view of these classes please refer to their class APIs.

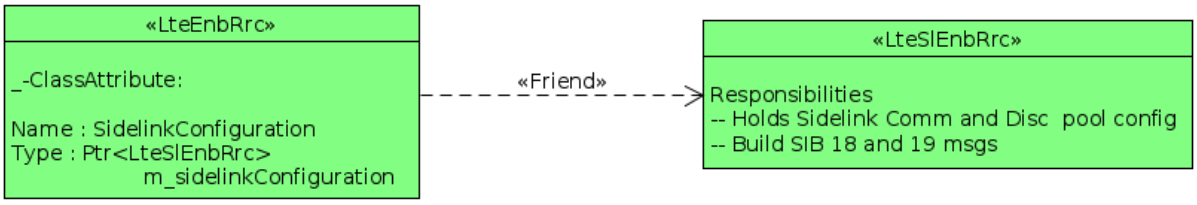


Fig. 5: Relationship between LteEnbRrc and LteSlEnbRrc

The following Figure *ns-3 LTE Sidelink pool configuration flow* shows the interaction among the classes to configure a Sidelink pool.

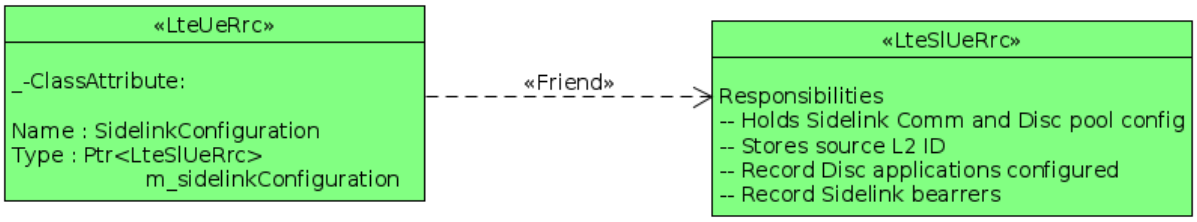


Fig. 6: Relationship between LteUeRrc and LteSlUeRrc

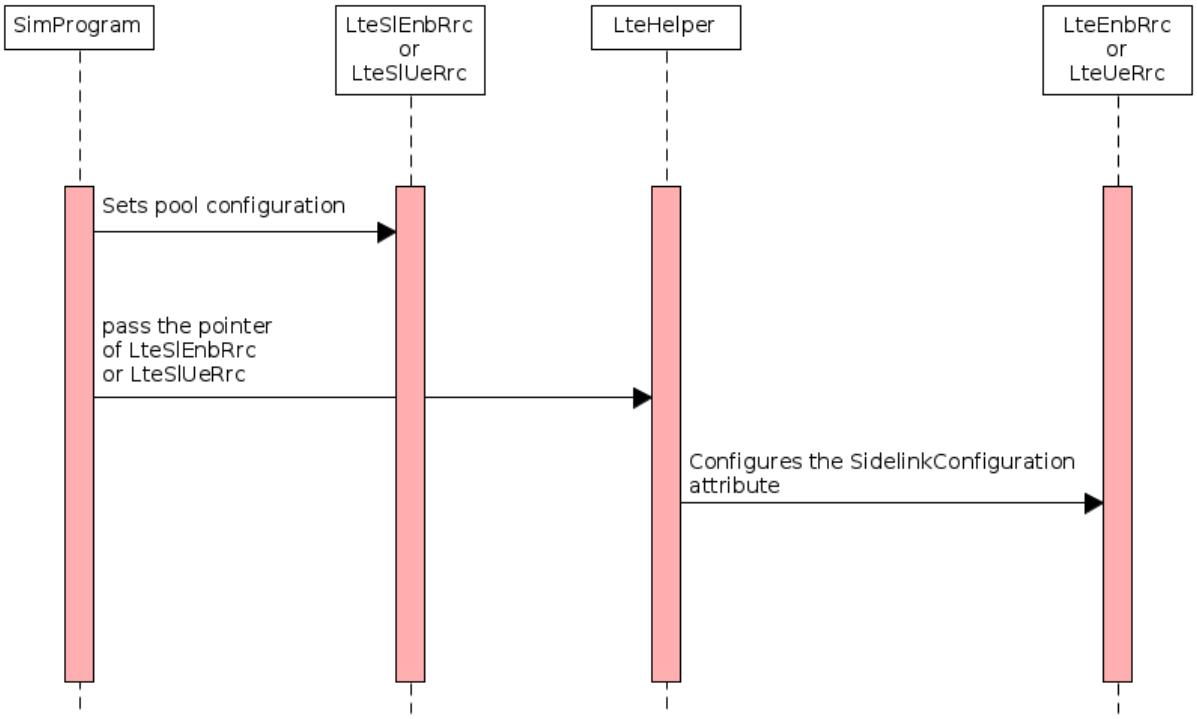


Fig. 7: ns-3 LTE Sidelink pool configuration flow

The Sidelink pools, for both in-coverage and out-of-coverage UEs are configured through the user's simulation script. Moreover, all the RRC SAP classes, i.e., control and data including the `LteRrcSap` class have been extended to support Sidelink functionalities.

In the following we will explain the remaining modifications specifically introduced in the eNB and UE RRC layer.

eNB RRC

To support in-coverage Sidelink scenarios, SIB18 and SIB19 were added to broadcast Sidelink resource pool configuration for communication and discovery respectively. For simplicity, we use the same periodicity (default: 80 ms) of all the SIB messages, which could be configured by changing the attribute `SystemInformationPeriodicity` value. The resource pools are configured through `LteHelper` as discussed earlier. Similar to other SIB messages, SIB18 and SIB19 are defined in `LteRrcSap class`. By receiving these SIB messages, a UE can deduce the type of ProSe service an eNB can support. The eNB is also now capable of processing `SidelinkUeInformation` messages sent by the UEs [TS36331]. In general, this type of message contains information related to the frequency the UE is intended to use for Sidelink, resources required by the UE for Sidelink communication or discovery and a list of destination identities. We note that, **at this moment only one Tx/Rx pool per UE** is supported. In response to the `SidelinkUeInformation`, eNB sends a `RrcConnectionReconfiguration` message containing the resource allocation information as per the supported ProSe services. In case of in-coverage Sidelink communication, resource allocation is performed as per MODE1, which is also referred as `Scheduled` mode [TR36843]. In this mode, the `RrcConnectionReconfiguration`, as per the standard, includes the specifications of the pool to be used and the timing specification for Sidelink Buffer Status Report (SL-BSR) transmission and re-transmission. On the other hand, if the pool is for MODE 2, i.e., `UeSelected` mode, it only includes the dedicated pool specifications.

For the in-coverage discovery, only `Type1`, i.e. UE selected resource allocation is supported.

UE RRC

The `LteUeRrc` and `LteSlUeRrc` classes have been extended to support all the ProSe services for the scenarios in Table *ns-3 LTE Sidelink supported/tested scenarios*. To highlight all the modifications, lets discuss them in context of each ProSe service.

Sidelink direct communication

The UE RRC now supports the creation of Sidelink bearers for both in-coverage and out-of-coverage UEs. A new function `DoActivateSidelinkRadioBearer` is implemented for this purpose. If the configuration of the TFT used indicates the nature of the Sidelink radio bearer as `Bidirectional`, the creation of Sidelink bearers for Tx and Rx, which is to populate the Tx/Rx pool configuration along with the list of destinations to the lower layers, occurs at the same time. Only the creation of PDCP/RLC instances for TX bearer is done in `DoActivateSidelinkRadioBearer` function. The creation of PDCP/RLC instances for Rx occurs upon the reception of first Sidelink packet [TS36300]. For the in-coverage case, the bearer establishment procedure involves the communication with the eNB by sending `SidelinkUeInformation` message for the sake of resource allocation as shown in Figures *ns-3 LTE Sidelink in-coverage radio bearer activation (Tx)* and *ns-3 LTE Sidelink in-coverage radio bearer activation (Rx)*.

For the out-of-coverage UEs, the Sidelink bearer activation and the resource allocation is done by the UE autonomously. Figure *ns-3 LTE Sidelink out-of-coverage radio bearer activation (Tx)* and *ns-3 LTE Sidelink out-of-coverage radio bearer activation (Rx)* show sequence diagrams of the out-of-coverage Sidelink bearer activation for TX and RX respectively.

As mentioned earlier, the UE RRC is now also capable of processing new SIB18 message and the Sidelink direct communication configuration received in `RrcConnectionReconfiguration` message. For in-coverage

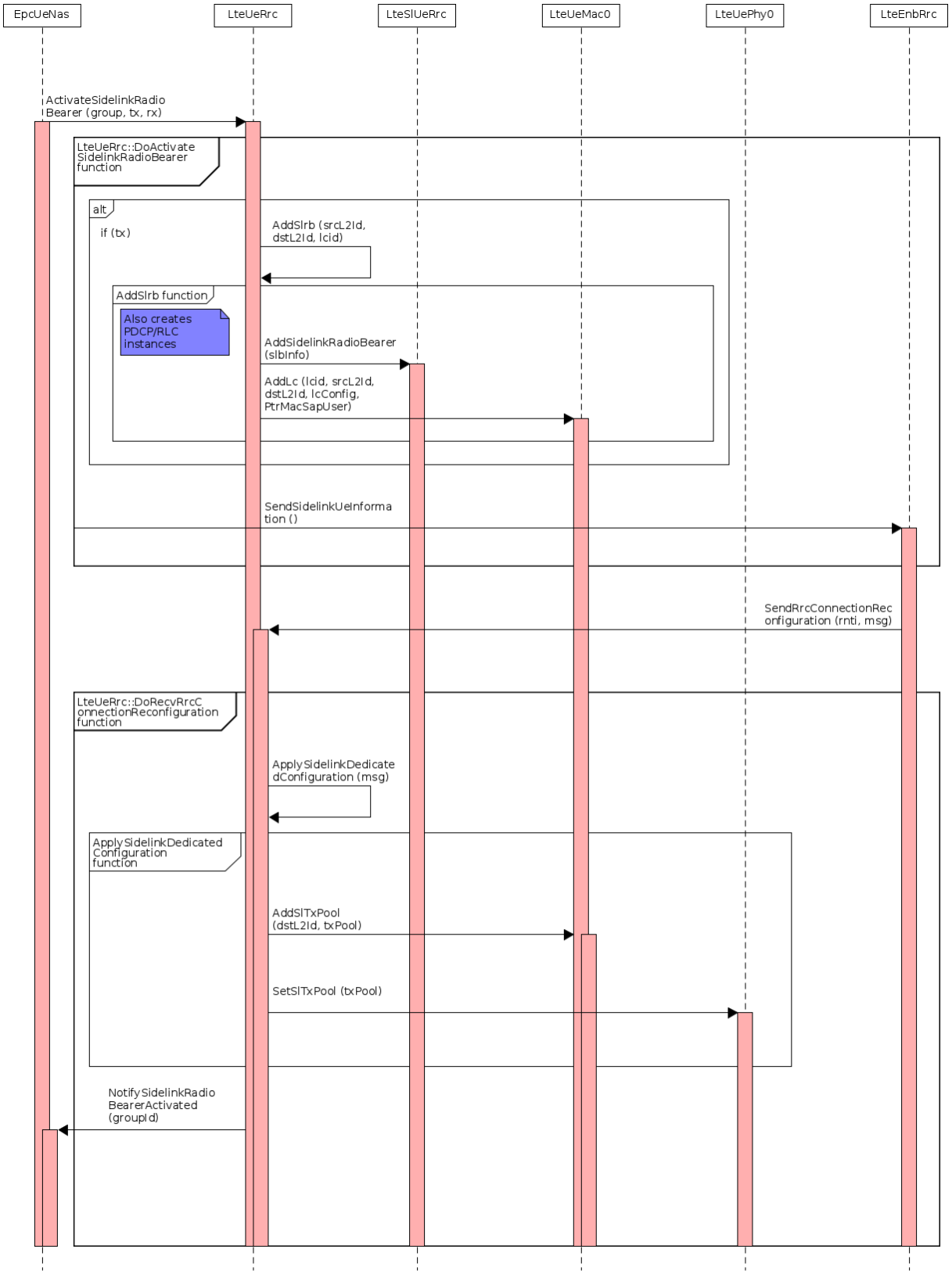


Fig. 8: ns-3 LTE Sidelink in-coverage radio bearer activation (Tx)

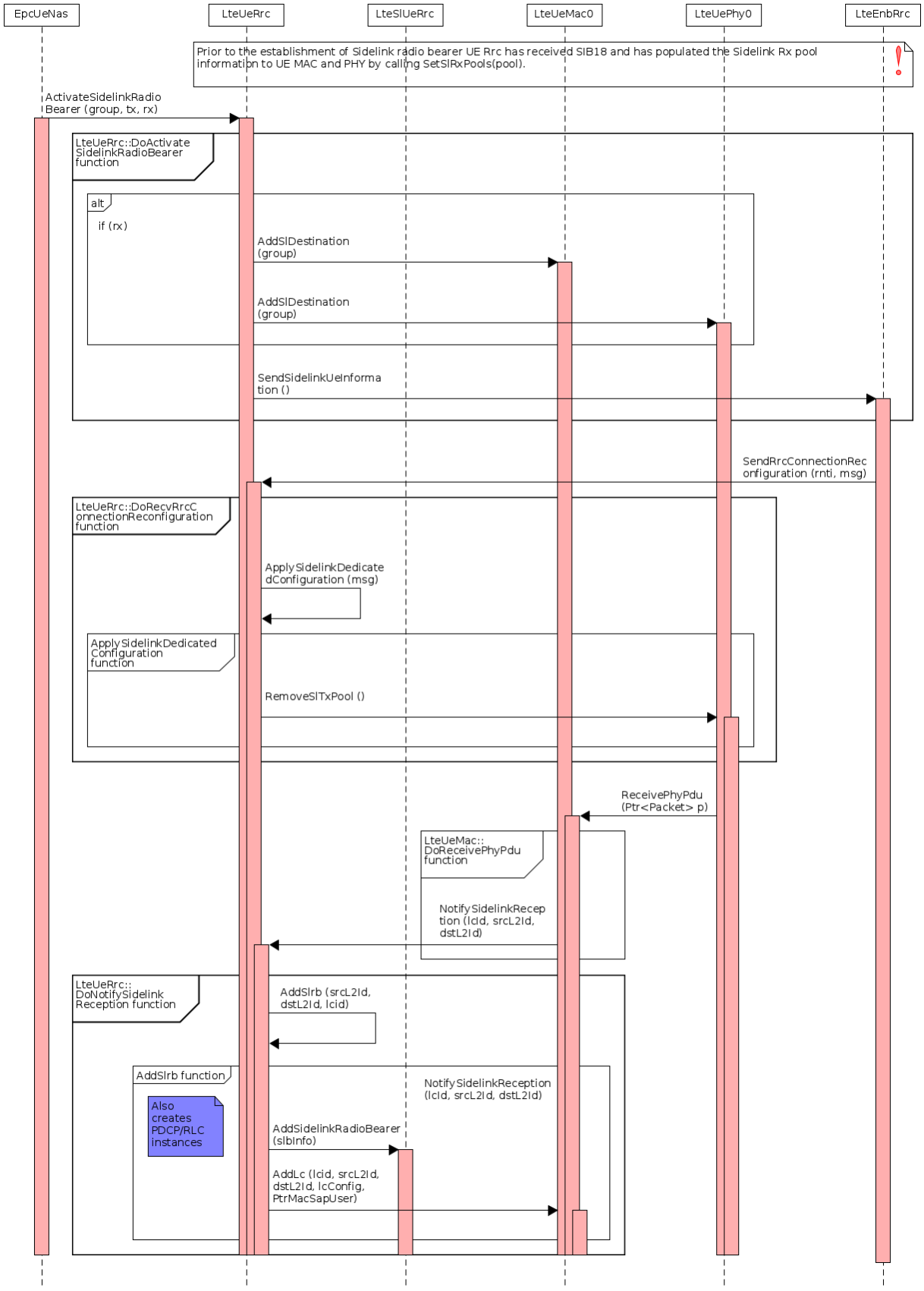


 Fig. 9: ns-3 LTE Sidelink in-coverage radio bearer activation (Rx)

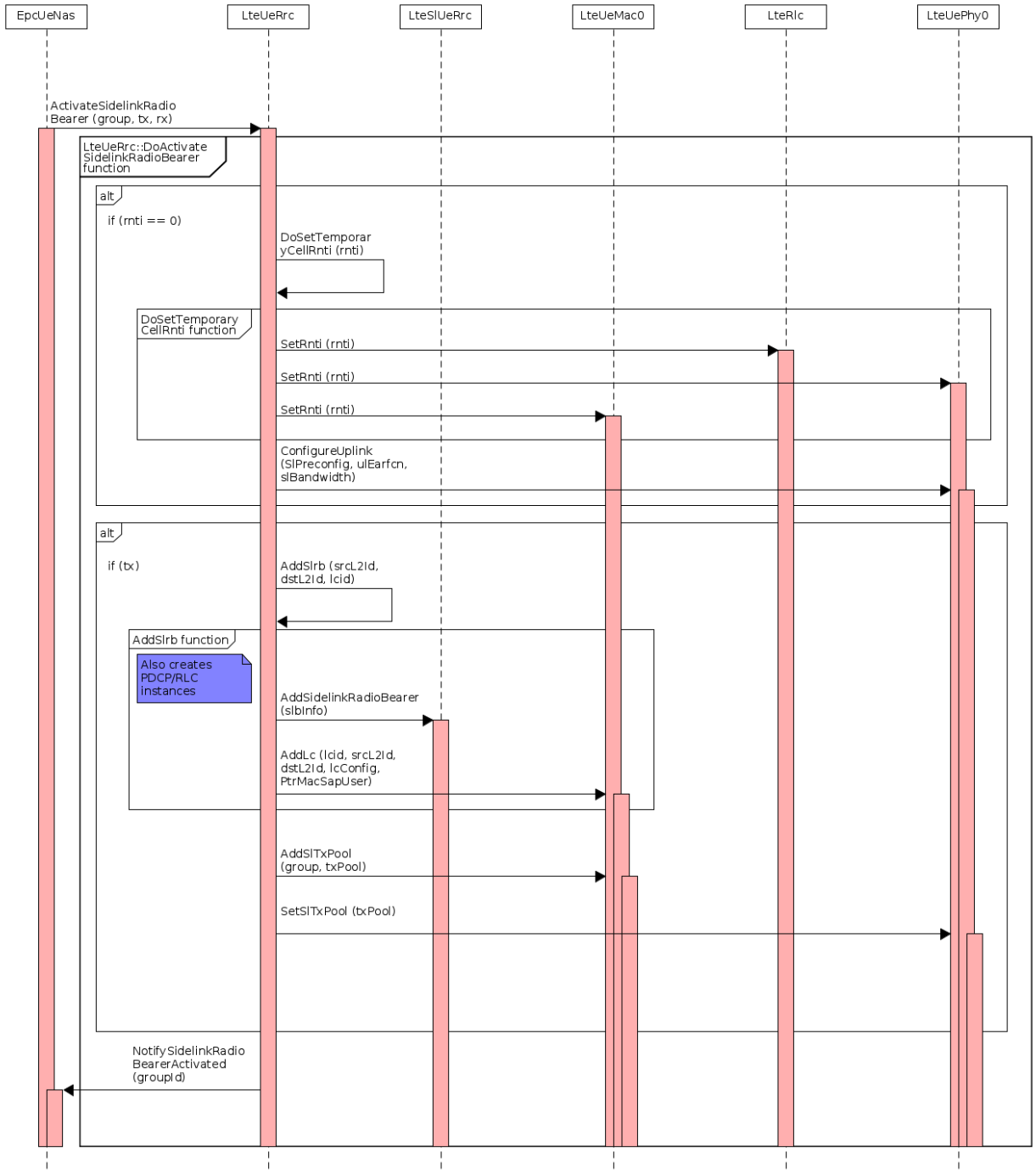


Fig. 10: ns-3 LTE Sidelink out-of-coverage radio bearer activation (Tx)

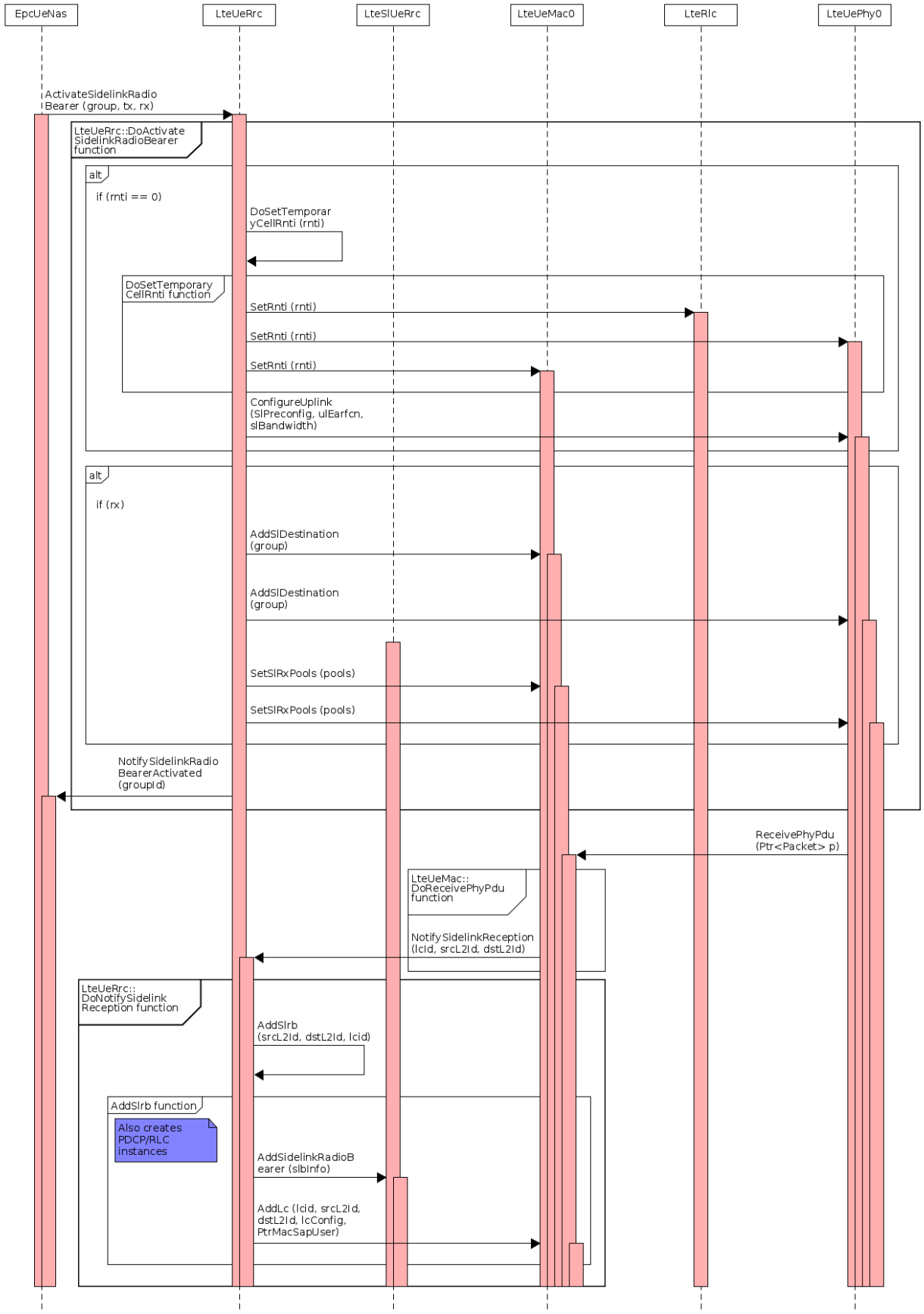


Fig. 11: ns-3 LTE Sidelink out-of-coverage radio bearer activation (Rx)

UEs, resource allocation can be done in MODE 1 and MODE 2. In MODE 1, the resources are set to "Scheduled" and the eNB is responsible of scheduling the exact number of resources (i.e., with the help of the scheduler), while in MODE 2, resources are "UeSelected" and a UE receives only one dedicated resource pool through `RrcConnectionReconfiguration` message to choose the resources from [TS36331]. By standard the SIB18 message is used as an indicator that an eNB supports ProSe communication services, thus, the UE transmits `SidelinkUeInformation` if it has received SIB18 message. A UE in "IDLE_CAMPED_NORMALLY" or in "CONNECTED_NORMALLY" is able to receive SIB18 messages, upon which, Rx pools are updated and also populated to the MAC and PHY layers. On the other hand, if the UE is out-of-coverage it uses a pre-configured pool and MODE 2 for resource allocation.

Sidelink direct discovery

For the Sidelink direct discovery, `LteUeRrc` and `LteSlUeRrc` classes now support creation/removal of discovery applications for both in-coverage and out-of-coverage UEs. The `LteSlUeRrc` class has functions called StartDiscoveryApps and StopDiscoveryApps to accomplish this process. These functions take in as arguments the application codes as payloads and the discovery role of the UE which are populated using the `LteSidelinkHelper`. For the in-coverage UEs, communication occurs with the eNB by transmitting the `SidelinkUeInformation` message for resource allocation, followed by the eNB `RrcConnectionReconfiguration` message. An in-coverage UE only transmits `SidelinkUeInformation` if it has received SIB19 message from its serving eNB. We note that, for in-coverage UEs, Scheduled resource allocation mode is not supported yet. The model supports only Type1, i.e., "UeSelected" resource allocation. An in-coverage UE can receive a list of multiple pools with the criteria of selection either RANDOM or RSRPBASED via the `RrcConnectionReconfiguration` message. A UE in IDLE_CAMPED_NORMALLY or in CONNECTED_NORMALLY can receive SIB19 messages, upon which, Rx pools are updated and populated to the MAC and PHY layers. On the other hand, when out-of-coverage, the UE only uses the pre-configured pool.

The classes `LteUeRrc`, `LteSlUeRrc` and `LteSlDiscHeader` are also extended to incorporate Model B application discovery along with Model A application discovery. In model A, the announcing UE periodically broadcasts PC5_DISCOVERY messages, called announcements, to indicate its presence. In model B, the discoverer UE seeks to discover UEs by sending a request message with the required application code. The discoveree UEs check if the application code is present in a list of monitored codes. If present, the UE then transmits the same application code as the intended response message payload. For both Model A and Model B discovery, the payloads are added to the appropriate message in the `LteSlDiscHeader` class and transmitted using the TransmitApp and TransmitDiscoveryMessage functions in `LteSlUeRrc` and `LteUeRrc` classes respectively. In case multiple requests from separate UEs are received, the model ensures that only one PC5_DISCOVERY response message is sent per discovery period. Multiple responses are not needed since these messages are broadcasted.

Note: At this stage only one pool is supported

Sidelink synchronization

The `LteUeRrc` class holds the main implementation for performing Sidelink synchronization. In particular, it supports

- Activation/Deactivation of Sidelink Synchronization Signal (SLSS)
- The configuration of the SLSS
- Delivering of the configured SLSS to phy layer for transmission
- Reception and L3 filtering of the SLSS from other UEs
- Selection of a Synchronization Reference UE (SyncRef)
- Notification of SyncRef change to the lower layers

By setting the attribute `UeSlssTransmissionEnabled == true` enables the transmission of SLSS. The purpose of synchronization process is to align the frame and subframe number of a UE with a SyncRef. An SLSS from a SyncRef is considered detectable if;

$$SRSRP_{strongestSyncRef}[dBm] - MinSrsrp[dBm] > syncRefMinHyst[dB] \tag{2.1}$$

$MinSrsrp$ is the minimum threshold for detecting SLSS configured using `MinSrsrp` attribute of `LteUeRrc` class and $syncRefMinHyst$ is the pre-configured hysteresis value, which could be configured from the user's simulation script. In case more than one SyncRefs are detected during a scanning period (Refer to the UE phy section below for details), a SyncRef with the highest Sidelink RSRP (S-RSRP) is selected. Upon re-selection of the SyncRef, if a better SyncRef is detected the UE selects this new SyncRef if;

$$SRSRP_{NewStrongestSyncRef}[dBm] - SRSRP_{OldStrongestSyncRef}[dBm] > syncRefDiffHyst[dB]$$

where $syncRefDiffHyst$ is a threshold representing how higher the S-RSRP of the newly detected SyncRef should be than the currently selected SyncRef's S-RSRP to consider a change. On the other hand, if the S-RSRP of the previously selected SyncRef satisfies the (2.1) and its S-RSRP is greater than a pre-configured $syncTxThreshOoC$ (i.e. out-of-coverage synchronization threshold) [TS36331] the SyncRef is considered as valid till the next re-selection.

In case, the previously selected SyncRef is not valid anymore and no other SyncRef has been detected, the UE itself becomes a SyncRef by choosing SLSS-ID and the Sidelink synchronization offset indicator. Different from the standard [TS36331], the current implementation for the sake of simplicity selects a SLSS-ID as `SLSS-ID = IMSI * 10`, while the synchronization offset indicator, as per the standard, is randomly selected between the two pre-configured offsets, i.e., `syncOffsetIndicator1` and `syncOffsetIndicator2`. We note that, when a Synchronization process is followed by ProSe group communication or discovery, a UE could only be able to receive a message from other UE, if they have same SLSS-ID and the group id is known to the receiving UE. However, when simulating only ProSe communication or discovery the SLSS-ID of all the UEs are initialized to 0 and the UE receives the transmission if the group id is known. Moreover, the transmission of the SLSS is subjective to the data transmission by the UE. At the beginning of every Sidelink Control (SC) period, a notification by the UE MAC layer is sent to the UE RRC, indicating if there is data to be transmitted. The function `DoNotifyMacHasSlDataToSend` and `DoNotifyMacHasNoSlDataToSend` of `LteUeRrc` class serve this purpose. If there is no data to be transmitted the UE stops transmitting the SLSS.

UE-to-Network Relay communication

The classes `LteUeRrc` and `LteSlUeRrc` now support the creation and maintenance of one-to-one direct communication links between two UEs using the Sidelink. A new type of signaling protocol, called PC5 signaling, has been defined for this purpose [TS24334]. In the model, the PC5 signaling messages are modeled as ns-3 headers and are listed in Table *PC5 Signaling Messages* together with the acronyms used in the rest of the section to refer to them. One-to-one direct communication links are used for UE-to-Network Relay communication functionality and two roles are defined for the UEs in this context: Relay UE and Remote UE. Figure *One-to-one direct communication typical PC5 messages exchange* shows a typical PC5 messages exchange between a Remote UE and a Relay UE to successfully establish and maintain a one-to-one direct communication link between them.

The Remote UE initiates the procedure by sending a DCRq message to the Relay UE. After the successful exchange of the DSMCm and DSMCp messages for link authentication, the Relay UE sends a DCA message to accept the connection. Once established, the Relay UE can request additional information from the Remote UE, by sending a RUIRq message, for which the Remote UE responds back with a RUIRs message. This procedure is optional and deactivated by the default, but it can be enabled using the attribute "RuirqEnabled" of the class `LteSlUeRrc`.

For both UEs, the reception of PC5 data or PC5 signaling messages indicates that the connection is active. Additionally, probe DCK messages are periodically sent by the Remote UE to maintain the one-to-one direct communication link connection in the event that there is no data to exchange. The Relay UE then responds by sending a DCKA message.

When either the Remote UE or the Relay UE requires to end the one-to-one direct communication link, it sends a DCR message to the peer UE, which then replies with a DCRA message.

During the one-to-one direct communication link setup, the Relay UE sends a DCRj message if it cannot accept the connection. Currently in the model, this only happen when the DSMCp message is not received on time (i.e., T4111 expires). The Remote UE can in turn reject the DSMCm and send a DSMRj. Currently, no logic is implemented for this, i.e., the Remote UE always accept the DSMCm.

Note: Rekeying messages are defined in the model but currently not used in the implementation.

Table 2: PC5 Signaling Messages

Acronym	PC5 Signaling Message	Message ID
DCA	DIRECT_COMMUNICATION_ACCEPT	2
DCK	DIRECT_COMMUNICATION_KEEPALIVE	4
DCKA	DIRECT_COMMUNICATION_KEEPALIVE_ACK	5
DCR	DIRECT_COMMUNICATION_RELEASE	6
DCRA	DIRECT_COMMUNICATION_RELEASE_ACCEPT	7
DCRj	DIRECT_COMMUNICATION_REJECT	3
DCRq	DIRECT_COMMUNICATION_REQUEST	1
DRRq *	DIRECT_REKEYING_REQUEST	15
DRRs *	DIRECT_REKEYING_RESPONSE	16
DRT *	DIRECT_REKEYING_TRIGGER	17
DSMCm	DIRECT_SECURITY_MODE_COMMAND	12
DSMCp	DIRECT_SECURITY_MODE_COMPLETE	13
DSMRj	DIRECT_SECURITY_MODE_REJECT	14
RUIRq	REMOTE_UE_INFO_REQUEST	18
RUIRs	REMOTE_UE_INFO_RESPONSE	19
* Messages defined but currently not used in the implementation		

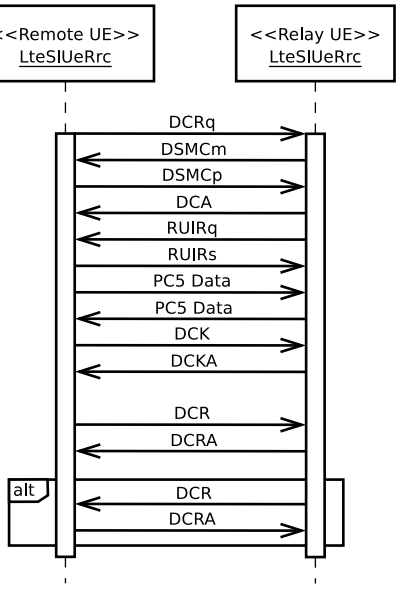


Fig. 12: One-to-one direct communication typical PC5 messages exchange

The class `LteSlUeRrc` uses a map of `LteSlO2oCommParams` objects indexed by an `LteSlPc5ContextId` structure to keep track of each active one-to-one direct communication link. The `LteSlPc5ContextId` is composed of the Layer 2 ID of the peer UE and the context ID. The context ID is a sequence number that each UE

having a Remote UE role increments when it starts a relay direct communication process. This sequence number is then sent in the DCRq to be used by the Relay UE for the `LteSlPc5ContextId` of the corresponding `LteSlO2oCommParams` object on his side.

The class `LteSlO2oCommParams` implements the one-to-one direct communication link state machine, the different one-to-one communication timers and counters, and also keeps the copies of certain PC5 signaling messages used for retransmission purposes. Figure *One-to-one direct communication link state machine for the Relay UE* and Figure *One-to-one direct communication link state machine for the Remote UE* depict the one-to-one direct communication link state machines depending on the UE role.

The timers involved in the one-to-one direct communication link establishment and maintenance are listed in Table *One-to-one direct communication link timers* together with their start, stop, and expiration conditions. Timers T4100, T4101, T4103, and TRUIR control the retransmission of DCRq, DCK, DCR, and RUIRq messages, respectively, bounded by the corresponding maximum number of retransmissions listed in Table *One-to-one direct communication PC5 signaling messages retransmission limits*.

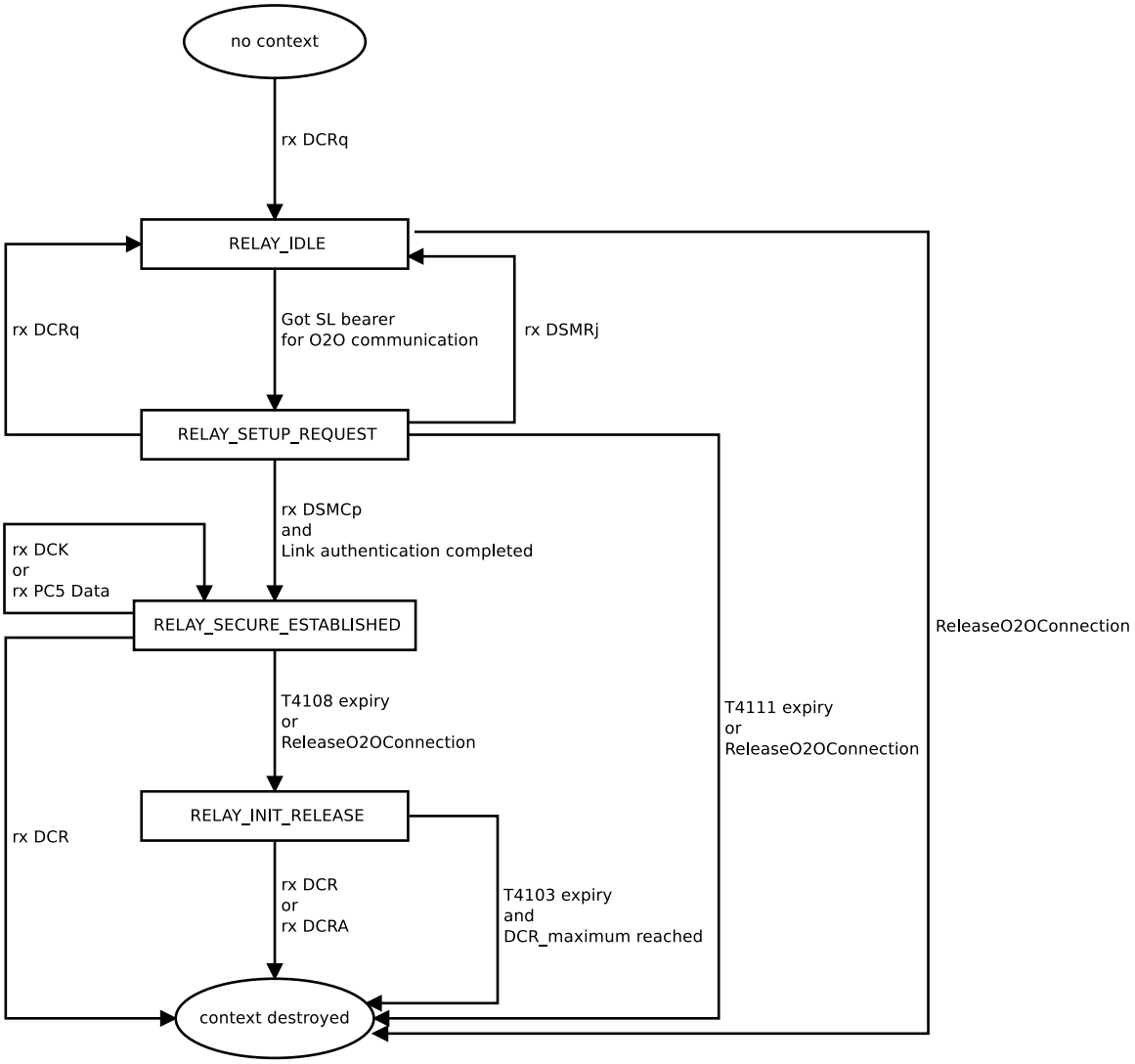


Fig. 13: One-to-one direct communication link state machine for the Relay UE

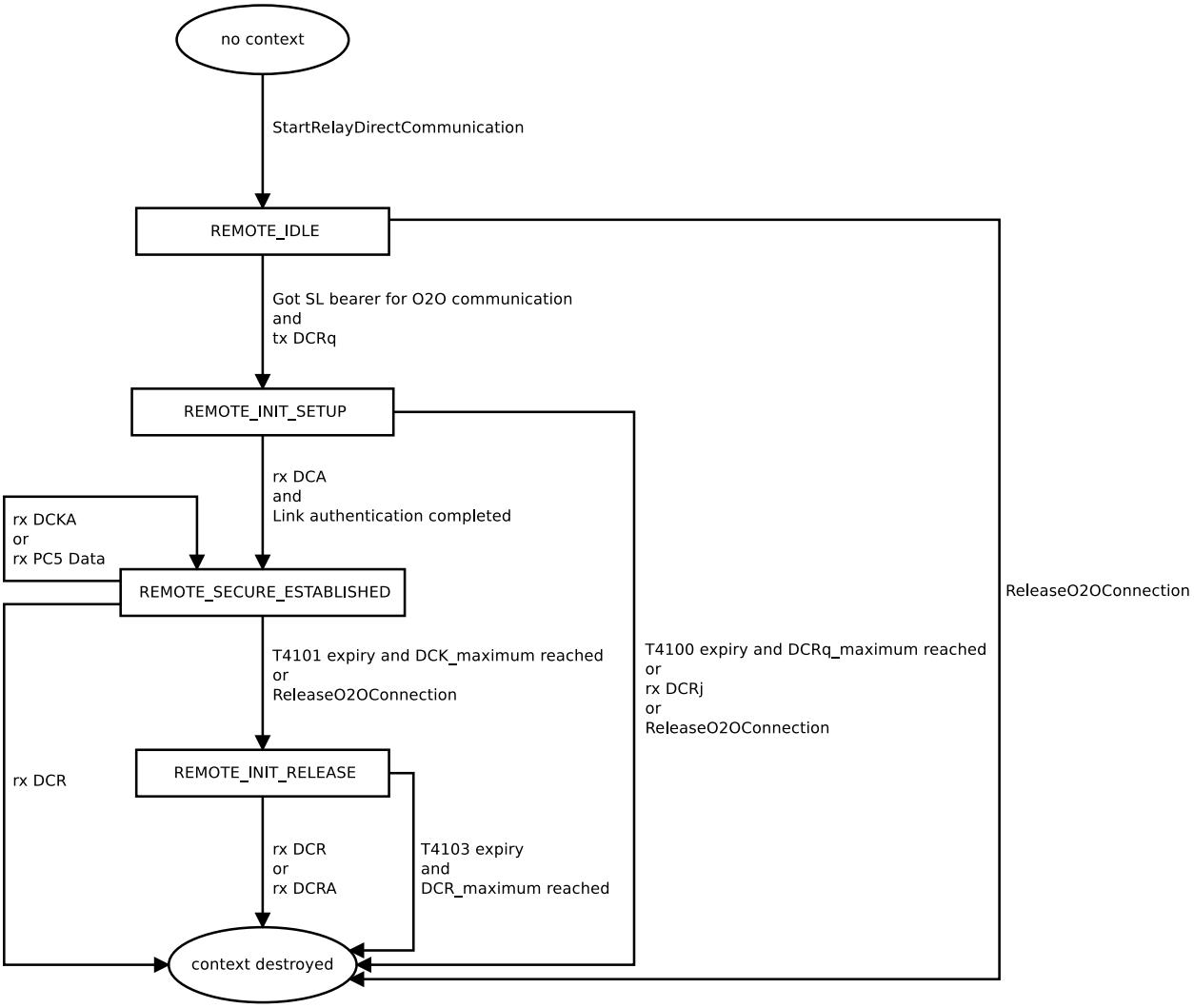


Fig. 14: One-to-one direct communication link state machine for the Remote UE

Table 3: One-to-one direct communication link timers

Timer	UE Role	Start	Stop	On Expiry
T4100	Remote	Upon transmitting a DCRq	Upon receiving a DCA or a DCRj	Retransmit DCRq
T4101	Remote	Upon transmitting a DCK	Upon receiving a DCKA or PC5 data	Retransmit DCK
T4102	Remote	Upon completing connection and upon receiving a DCKA or PC5 data thereafter	Upon receiving a DCKA or PC5 data	Transmit DCK
T4103	Remote Relay	Upon transmitting a DCR	Upon receiving a DCRA	Retransmit DCR
T4108	Relay	Upon completing connection and upon receiving a DCK or PC5 data thereafter	Upon receiving a DCK or PC5 data	Initiate connection release
T4111	Relay	Upon transmitting a DSMCm	Upon receiving a DSMCp or DSMRj	Abort connection setup
TRUIR	Relay	Upon transmitting a RUIRq	Upon receiving a RUIRs	Retransmit RUIRq

Table 4: One-to-one direct communication PC5 signaling messages retransmission limits

Variable	UE Role	Description
DCRq_maximum	Remote	Maximum number of DCRq retransmissions
DCK_maximum	Remote	Maximum number of DCK retransmissions
DCR_maximum	Remote Relay	Maximum number of DCR retransmissions
RUIR_maximum	Relay	Maximum number of RUIR retransmissions

UE-to-Network Relay discovery and selection

The classes `LteUeRrc` and `LteSlUeRrc` were extended to support the creation/removal of UE-to-Network Relay services, which are identified by relay service codes, and are used for the UE-to-Network Relay discovery procedure. Both discovery models (A and B) are supported for UE-to-Network Relay discovery.

With Model A, the Relay UE periodically broadcasts PC5_DISCOVERY messages of type *UE-to-Network Relay Discovery Announcement* to indicate its presence and includes the supported relay service code in the message. Remote UEs monitor those messages and react accordingly when they find the relay service code of interest.

With Model B, the Remote UE sends a PC5_DISCOVERY message of type *UE-to-Network Relay Discovery Solicitation* containing the relay service code it is looking for. Relay UEs monitor those messages and verify if the relay service code in the solicitation message is in their list of provided service codes. In affirmative case, they respond by sending a message of type *UE-to-Network Relay Discovery Response*, which are monitored by the Remote UEs. In case multiple solicitations from separate Remote UEs are received, the model ensures that only one PC5_DISCOVERY response message is sent per discovery period. Multiple responses are not needed since these messages are broadcasted.

For both discovery models, the appropriate parameters depending on the discovery message type are set in an `LteSlDiscHeader` object in the function TransmitRelayMessage of the `LteSlUeRrc`, which is then passed to lower layers for transmission using the TransmitDiscoveryMessage function in the `LteUeRrc`.

As in the Sidelink Direct Discovery, the relay service code and the role of the UE (Remote UE or Relay UE) are populated using the `LteSidelinkHelper`. When using Model A, the Relay UE sends announcements every

discovery period. When using Model B, the frequency of the solicitation messages sent by the Remote UEs can be set with the attribute `LteSlUeRrc::RelaySolFreq`.

When a Remote UE starts the UE-to-Network Relay service, the `LteUeRrc` notifies the `LteUePhy` to enable the Sidelink Discovery Reference Signals Received Power (SD-RSRP) measurements. Afterwards, the SD-RSRP measurement reports are received periodically by the `LteUeRrc` which processes them (e.g., apply L3 filtering if it is enabled) and report the valid Relay UEs together with their relay service code and the (L3 filtered) SD-RSRP to the `LteSlUeRrc`. A Relay UE is considered valid if its (L3 filtered) SD-RSRP satisfies:

$$SD\text{-}RSRP[dBm] - qRxLevMin[dBm] > minHyst[dB]$$

where $qRxLevMin$ and $minHyst$ are provided via the Sidelink preconfiguration if the Remote UE is out-of-coverage, or via the SIB19 broadcasted by the eNB if the Remote UE is in-coverage. In either case, these parameters must be configured in the simulation scenario.

Then, the `LteSlUeRrc` reports the list of valid Relay UEs to the `LteSlUeController` which executes the configured Relay UE selection algorithm and report back the selected Relay UE (if any) to the `LteSlUeRrc`. Depending on the result of the selection algorithm, the `LteSlUeRrc` determines if the Remote UE should:

- disconnect from its current Relay UE if any, e.g., if the current Relay UE is not suitable anymore, or a most suitable Relay UE was selected, and/or
- try to connect (i.e., start the one-to-one direct communication link setup) to the selected Relay UE, or
- perform no action, e.g., no suitable Relay UE is available, or the current Relay UE is still the most suitable.

An example of the overall UE-to-Network Relay selection process and the functions involved is depicted in Figure *UE-to-Network Relay selection mechanism.* and the extensions made to the other components of the model to support them are described on each corresponding section in the rest of the document.

Sidelink UE controller

The class `LteSlUeController` was created to provide a framework for the implementation of custom UE-to-Network Relay configurations and algorithms. The Sidelink UE Controller service interface is provided by two Service Access Points (SAPs):

- The `LteSlUeCtrlSapProvider` part is provided by the `LteSlUeController` and used by the `LteSlUeRrc`
- The `LteSlUeCtrlSapUser` part is provided by the `LteSlUeRrc` and used by the `LteSlUeController`

Currently, no primitives are implemented for the `LteSlUeCtrlSapUser` and the following functions are defined for the `LteSlUeCtrlSapProvider`:

- `LteSlUeCtrlSapProvider::RecvRelayServiceDiscovery` is used to indicate that the Remote UE received a UE-to-Network Relay Discovery Announcement or a UE-to-Network Relay Discovery Response containing a monitored relay service code.
- `LteSlUeCtrlSapProvider::Pc5ConnectionStarted` indicates that the one-to-one direct communication link setup procedure started. This occurs for Remote UE when it starts the relay direct communication service and it sends the first DCRq to a given Relay UE, and for the Relay UE the indication is done when it receives a DCRq from a Remote UE and process it.
- `LteSlUeCtrlSapProvider::Pc5SecuredEstablished` indicates that the one-to-one direct communication link setup was completed successfully by the UE. Currently, this happens when the Relay UE receives DSMCp message from the Remote UE and the link authentication was completed, and when the Remote UE receives the corresponding DCA message from the Relay UE. This function should configure the UE to be able to use the one-to-one direct communication link to send and receive data on it and towards the network, e.g., IP configuration and forwarding, traffic filtering and Sidelink bearers configuration.

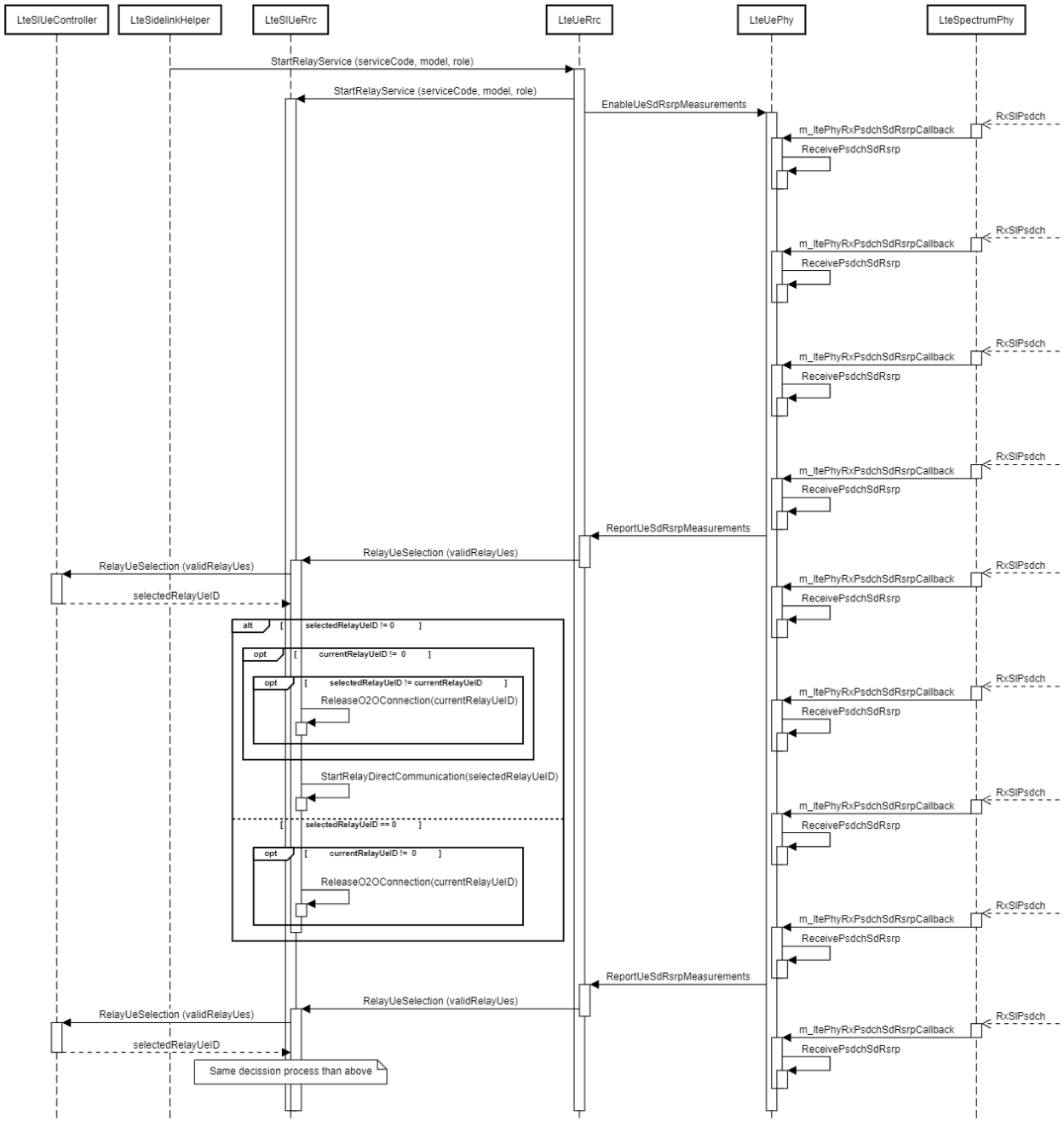


Fig. 15: UE-to-Network Relay selection mechanism.

- `LteSlUeCtrlSapProvider::Pc5ConnectionTerminated` indicates that the direct communication link release procedure is complete for an established one-to-one link. This happen for the UEs in both roles, when they receive a DCR, or when they already started the procedure and receive a DCRA message, or when T4103 expires and DCR_maximum was reached. This function should implement the logic to clean up the configurations in the UE associated to the released link.
- `LteSlUeCtrlSapProvider::Pc5ConnectionAborted` is called by the Remote UE when the direct one-to-one direct communication link setup fails, i.e., when a DCRj message is received or when T4100 expires and DCRq_maximum was reached. This function should implement the logic to clean up the configurations associated to the link being setup.
- `LteSlUeCtrlSapProvider::RecvRemoteUeReport` is used by the Relay UE upon connecting to a Remote UE to indicate that the network was informed about the Remote UE connection.
- `LteSlUeCtrlSapProvider::RelayUeSelection` is called during the Relay selection procedure and it is the primitive that should implement the Relay selection logic of the controller given the list of valid detected Relay UEs and their corresponding Sidelink Discovery Reference Signals Received Power (SD-RSRP). This function should return the ID of the selected Relay UE if any, or zero otherwise to indicate no suitable Relay UE is available.

A Sidelink UE controller is implemented by writing a subclass of the `LteSlUeController` superclass and implementing each of the above mentioned SAP interface functions. Users may develop their own controller this way, or may use the provided example implementation called `LteSlBasicUeController`, which is available in the model to provide the following baseline implementation of the functions mentioned above:

- `LteSlBasicUeController::DoRecvRelayServiceDiscovery` does not implement any logic but can be edited to store the content of the discovery messages (e.g. status indicator) and contribute to future Relay UE selection algorithms.
- `LteSlBasicUeController::DoPc5ConnectionStarted` does not implement any logic and is currently only used for tracing the start of the one-to-one connection setup.
- `LteSlBasicUeController::DoPc5SecuredEstablished` of this controller interacts with the `LteUeNetDevice` of the UE and the `LteSidelinkHelper` to create the `LteSlUeNetDevice` corresponding to the one-to-one direct communication link that was established. It also configures static IPv6 addresses and routing, enables the packet forwarding for the Relay UE, and configures the appropriate packet filters by creating the corresponding `LteSlTft` objects. Please note that this controller sets the Relay UE address as the default route for the Remote UE. Thus, all packets from the Remote UE will be sent to Relay UE instead of to the eNodeB after the connection is established.
- `LteSlBasicUeController::DoPc5ConnectionTerminated` of this controller reverts the configurations made during the `LteSlBasicUeController::DPc5SecuredEstablished` call for the link being released.
- `LteSlBasicUeController::DoPc5ConnectionAborted` resets the link setup process for the Remote UE.
- `LteSlBasicUeController::DoRecvRemoteUeReport` is used by the Relay UE to inform the network about a new Remote UE connection. This is done by calling the helper function `LteSidelinkHelper::RemoteUeContextConnected` with the corresponding Remote UE information.
- `LteSlBasicUeController::DoRelayUeSelection` currently implements the following three Relay UE selection algorithms, and the controller will use the one configured in the `LteSlBasicUeController::RelayUeSelectionAlgorithm` attribute.
  - RANDOM_NO_RESELECTION: (Default) If not connected to any Relay UE, the algorithm randomly selects a Relay UE from the list of valid Relay UEs, and then returns its ID. Otherwise, it simply returns the ID of the Relay UE to which the Remote UE is already connected.

  - MAX_SDRSRP_NO_RESELECTION: If not connected to any Relay UE, the algorithm selects the Relay UE with the strongest SD-RSRP from the list of valid Relay UEs, and then returns its ID. Otherwise, it simply returns the ID of the Relay UE to which the Remote UE is already connected.
  - MAX_SDRSRP: The algorithm selects the Relay UE with the strongest SD-RSRP from the list of valid Relay UEs, and then returns its ID, no matter the Remote UE current connection status.
- Additionally, two traces are provided which can be hooked up to some custom functions:
  - `LteSlBasicUeController::RelayUeSelection` traces each time a Remote UE selects a new Relay UE, and
  - `LteSlBasicUeController::Pc5ConnectionStatus` traces each status change of each one-to-one connection between a Remote UE and a Relay UE.

Note: Only IPv6 is supported when using the UE-to-Network Relay functionality

PDCP

The `LtePdcp` class has been extended to include ProSe source Layer 2 ID and the ProSe Layer-2 Group ID to identify the PDCP/RLC pair to be used in the receiving UE. The reason is that in Sidelink, each radio bearer is associated with one PDCP entity. And, each PDCP entity is associated to one RLC entity, since it uses RLC-UM mode for the transmission / reception [TS36323]. Thus, given the nature of Sidelink communication, i.e., one to many, it may happen that multiple UEs transmit to a single UE, assigning LCIDs independently. It may happen that two UEs select same LCID for the same group. Therefore, for a receiving UE to distinguish between two Sidelink Radio bearers (SLRBs) with the similar LCIDs requires additional identifiers. An enumeration for the Service Data Unit (SDU) type is added in the LtePdcpSapUser class to support the differentiation between distinct types of packets such as IP, ARP, PC5_SIGNALING, and NON_IP at the RRC layer.

RLC

Similar to the PDCP layer, the RLC layer now has ProSe source Layer 2 ID and the ProSe Layer-2 Group ID. This is achieved by extending the structure of `TransmitPduParameters` of `LteMacSapProvider` class. As mentioned earlier, only RLC-UM mode is used for Sidelink. Moreover, `LteRlcUm` class is also extended in accordance to the specification define in section 7.1 of [TS36322]. It says that "For RLC entity configured for Sidelink Traffic Channel (STCH), the state variables `VR(UR)` and `VR(UH)` are initially set to the SN of the first received PDU". A new attribute named "ChannelType" is introduced, which is set to type `STCH` for Sidelink.

MAC

Among the following eNB and UE specific enhancements for ProSe, one of the important changes, irrespective of the eNB or UE implementation, is the limitation to the maximum frame number, which is now set to 1024. This is because all the computations to determine correctly the boundaries of ProSe communication and discovery periods and their resource allocation considers the maximum frame number to be 1024. This limitation also helps us to implement the rollover constraints more accurately, e.g., to have a configurable offset to be applied at the beginning of each communication and discovery periods. This change mainly impacted all the schedulers. In particular, every scheduler maintains a map of uplink resource allocation using the SFnSF (i.e., combination of frame and subframe number) as an unique key to update the PUSCH based CQIs. By limiting the frame number to 1024, means that a combination could repeat after 10 sec of simulation, thus, a c++ map would not allow to insert new allocation information with the same SFnSF combination. To handle this, we appropriately remove any old allocation information before inserting a new one.

eNB MAC

`LteEnbMac` has been extended to support in-coverage ProSe communication, i.e., Mode 1 resource allocation. The SAP between the `LteEnbRrc` and `LteEnbMac` is extended to receive the pool configuration to be conveyed to the scheduler to schedule the resources. If the type of resource allocation for the configured resource pool is `Scheduled` the eNB MAC is now capable of receiving Sidelink Buffer status Report (SLBSR) from the UEs. To achieve this, the `structure MacCeListElement_s` in `ff-mac-common.h` file is extended to support BSR of type SLBSR. The reporting of the BSR in uplink is the same as shown in Figure lte-design:ref:*fig-ca-uplink-bsr*. To handle the scheduling of Sidelink resources for MODE 1, a scheduler `RrSlFfMacScheduler` based on the existing Round Robin implementation is also provided. Thus, making it the only scheduler, for the time being, in LTE module capable of supporting Sidelink and normal LTE resource scheduling. It also makes sure that Sidelink and the uplink resources are not scheduled in the same subframe. For Sidelink discovery, as stated before, the "Scheduled" mode is not implemented yet.

UE MAC

The `LteUeMac` is extended to support all the three ProSe services. The scheduling of Sidelink transmissions is executed in a new function called `DoSlDelayedSubframeIndication` that occurs after the downlink control messages are processed in order to know if there are any uplink transmissions scheduled in the current subframe.

The DoNotifyUlTransmission function in the UE MAC is where the PHY notifies the MAC that it has scheduled a transmission. The UE MAC now processes the discovery message's PHY PDU, the Sidelink Control Information message's PHY PDU and the data message's PHY PDU separately.

In the following we explain these extensions in context of each ProSe service.

Sidelink direct communication

The UE MAC is now capable of receiving the RLC buffer status notifications with extended Sidelink identifiers, i.e., layer 2 source and destination ID. A tuple `SidelinkLcIdentifier` of `LteUeMac` holds the *LCID*, *SrcL2Id* and *DstL2Id*, which is then used as a key for a c++ map storing all the buffer status notifications from RLC layer.

Before a Sidelink D2D communication transmission on the Physical Sidelink Shared Channel (PSSCH) takes place, a Sidelink grant needs to be configured. The grant contains information used to configure how, when, and where in the Sidelink resource pool transmissions will occur. This information is exchanged between communicating parties using the Sidelink Control Information Message (SCI) on the Physical Sidelink Control Channel (PSCCH).

For "Scheduled" resources, i.e. MODE 1, a UE is now capable of processing the Sidelink DCI (i.e., DCI 5 [TS36212]) message containing the resource allocation information for PSCCH and PSSCH transmissions. A UE for transmitting PSCCH uses the PSCCH resource indicated in the SL DCI from the eNB. To ensure the reliability of Sidelink Control information (SCI), each control message is transmitted twice using two different subframe, each utilizing 1 RB belonging to the configured resource block pool [TS36213]. Thus, supporting 2 PSCCH transmissions per grant. While the PSSCH (i.e. Data) transmissions are computed asper the current frame and subframe number and the information including in the SL DCI, i.e., Time Resource Pattern Index (iTrp), Starting RB index and the total number of RBs to be used. Since there are no HARQ feedbacks in Sidelink PSSCH, each transport block is transmitted using 4 HARQ transmissions (i.e. RV = 4). Hence, for each grant every transport block is transmitted using 4 subframes.

On the other hand, for the "UeSelected" resource scheduling, i.e. MODE 2, `LteUeMac` chooses a random PSCCH resource among the available resources and similarly transmits twice using two subframes. While for transmitting PSSCH it uses a pre-configured pool configuration to determine RBs and subframe for 4 PSSCH transmissions. The `LteUeMac` has been enhanced to dynamically configure the Modulation and Coding Scheme (MCS), and the number of Physical Resource Blocks (PRBs) to be used in the PSSCH of each Sidelink period. When there is data to be transmitted over the PSSCH, the transmission buffer size and the available number of transmissions per PSSCH are used to determine the Transport Block Size (TBS) required per Sidelink period. The number of transmission opportunities in

the PSSCH is configured with the KTRP value. Network operators have the option to choose a single KTRP value, or a range from the acceptable values, by setting the trpt- subset accordingly in the Sidelink pool configuration [TS36.213].

A set of configuration metrics are implemented in the LTE UE MAC class; the grant scheduling is performed following the configuration (including optimization goal when applicable) associated with the metrics to meet the required TBS.

1. Fixed (FIXED): Default setting. Uses grant configuration specified by user through the LteUeMac attributes: Ktrp, UseSetTrp, SetTrpIndex, SlGrantMcs, and SlGrantSize. Ktrp and SetTrpIndex are used to specify the number of active subframes and the index of the TRP respectively [TS36213]. The last two attributes are used to specify the MCS and the number of RBs (per subframe) to be used for Sidelink transmissions.

2. Random (RANDOM): Selects a grant configuration (MCS, PRBs, KTRP) at random. For each enabled KTRP value, keep record of all <PRBs, MCS> pairs satisfying the required TBS, choose <PRB, MCS> pair at random.

3. Minimum PRBs Usage (MIN_PRB): Selects a grant configuration that utilizes the least number of PRBs per transmission. For each enabled KTRP value, find the <PRB, MCS> pair with the fewer number of PRBs satisfying the required TBS and choose the <PRB, MCS> pair with fewer PRBs.

4. Maximum Coverage (MAX_COVERAGE): Selects a grant configuration that would maximize the communication range. For each enabled KTRP value, keep record of all <PRBs, MCS> pairs satisfying the required TBS. Evaluate and rank each pair with utility function D = 10 log (PRB) + SNR_threshold (Target BLER at 4th HARQ transmission). Choose the <PRB, MCS> pair having the lowest value D.

Use the SlScheduler attribute from the UE MAC to select the scheduling grant metric (LteUeMac::FIXED is the default).

Moreover, upon the reception of a PHY PDU the `LteUeMac` class is now able to notify UE RRC to establish a Sidelink radio bearer for the reception.

Sidelink direct discovery

The list of pending discovery messages (received from RRC) to be sent are stored in the m_discPendingTxMsgs variable. The Sidelink discovery grant related variables are stored in the `DiscGrant` structure. This contains the discovery message to be sent in a selected resource and a list of PSDCH transmissions within the pool (each discovery message can be retransmitted up to 3 times). The Sidelink discovery pool information is stored in the `DiscPoolInfo` structure. This contains a list of grants for the current discovery period and the next discovery period. These extensions help the model support the transmission of multiple discovery messages within a Sidelink period. Each discovery message occupies 2 contiguous RBs in a subframe belonging to the assigned pool.

In reception, the discovery messages are passed to the RRC layer, where they are appropriately processed.

Sidelink synchronization

For the Sidelink synchronization, `LteUeMac` receives notification from `LteUePhy` upon change of time, i.e., change of frame and subframe number when a new SyncRef is selected.

PHY

eNB PHY

The LTE D2D module of *ns-3* at the time of writing this documentation does not support Radio Link Failure (RLF) functionality. Therefore, to simulate out-of-coverage scenarios a workaround has been implemented to disable eNB PHY layer. An eNB can be disable by configuring the attribute "DisableEnbPhy" of `LteHelper`. Additionally, new conditions have been implemented to disregard control messages such as Sidelink Control Information SCI, Master

Information Block for Sidelink MIB_SL, and Sidelink Discovery Message SL_DISC_MSG as these do not pertain to uplink control messages to the eNB.

UE PHY

As shown in Figure *PHY and channel model architecture for the UE*, to receive the Sidelink transmission in the uplink channel the `LteUePhy` includes an additional `LteSpectrumPhy` instance. In particular, the function `DoConfigureUplink`, which is called by `LteUeRrc` is responsible of adding this new instance to the channel. The `LteUePhy` class has new functions PhyPscchPduReceived, PhyPsdchPduReceived, and PhyPsbchPduReceived to respectively receive control, discovery, and broadcast PHY PDU respectively. Similarly, `LteSpectrumPhy` is extended with new function RxSlPscch, RxSlPsdch, and RxSlPsbch (and corresponding callbacks for interconnecting PHY and MAC) to accomplish reception of control, discovery, and broadcast messages respectively. It also has the function StartRandomInitialSubframeIndication for triggering the SubframeIndication method by randomly choosing the frame and subframe number. Additionally, the function SetDownlinkCqiPeriodicity sets the periodicity for the downlink periodic wideband (P10) and aperiodic subband (A30) CQI reporting. The function InitializeDiscRxPool is used to initialize the discovery reception pool taking in appropriate frame number and subframe number as parameters.

To differentiate between sending uplink and Sidelink MAC PDUs, separate functions DoSendMacPdu and DoSendSlMacPdu respectively have been used. To aid in packet burst transfer, the function GetSlPhyParameters has been created to get the transmission parameters for the given packet burst. Occasionally, the UE PHY informs the MAC through the NotifyUlTransmission function that other transmissions with higher priority needs to occur and that the MAC should not schedule any Sidelink transmissions.

The `LteUePhy` class is extensively edited to support all the three ProSe services.

Sidelink direct communication

The `LteUePhy` is now capable of receiving and transmitting Sidelink control messages. In particular, DCI format 5, and SCI format 0 is now supported [TS36212]. The UE PHY receives the information related to the Sidelink communication pool from the UE RRC in order to compute the boundaries of an SC period to correctly map the frame/subframe and the RBs associated with the PSCCH and PSSCH. At the beginning of each subframe, it checks if the Sidelink transmission pool is added, it make sure to initialize it in order to compute exactly the frame and subframe number of the next SC period. This is needed to remove the Sidelink transmission grant associated to the previous SC period, since a grant is valid only for one SC period.

Sidelink direct discovery

For the discovery, the `LteUePhy` now also supports the transmission and reception of the discovery messages. A UE upon the reception of a discovery message, simply passes it to the MAC layer, which in turn delivers it to the RRC for further filtering.

The `LteUePhy` was also extended to support the measurements storage, layer-1 filtering, and report to upper layers of the Sidelink Discovery Reference Signals Received Power (SD-RSRP), which is used for UE-to-Network Relay (re)selection in the model. When a Remote UE starts the UE-to-Network Relay service, the `LteUeRrc` notifies the `LteUePhy` to enable the SD-RSRP measurements. The `LteUePhy` then calculates the SD-RSRP of each received discovery message and stores them indexed by Relay UE ID and Service Code in the function `ReceivePsdchSdRsrp`. The SD-RSRP measurements are reported to the `LteUeRrc` periodically. A layer-1 filtering measurement period is defined as 4 discovery periods. Given that the length of the discovery period is configurable, the function `GetUeSdRsrpMeasurementFilterPeriod` calculates the layer-1 filtering measurement period, and the reporting function `ReportUeSdRsrpMeasurements` is scheduled to be executed with this periodicity. This function averages the SD-RSRP of each Relay ID and Service Code that were stored during a layer-1 measurement period, and then reports these quantities to the `LteUeRrc`. The trace

`LteUePhy::ReportUeSdRsrpMeasurements` was also added, which can be hooked up with a custom function to monitor the SD-RSRP measurements values.

Sidelink synchronization

The `LteUePhy` is extended to support Sidelink synchronization functionality. In particular, it supports:

- Transmission and reception of SLSS
- Scheduling of SLSS scanning periods
- Measuring S-RSRP and reporting it to the RRC layer
- Synchronizing to the chosen SyncRef
- Performing change of timing
- Scheduling of the SyncRef re-selection process

In the current implementation, all the UEs by default are perfectly synchronized, i.e., all the UEs in a simulation upon being initialized pick the same frame and subframe 1 to start with. Therefore, to simulate synchronization and to make every UE to pick a random frame and subframe number a new attribute "UeRandomInitialSubframeIndication" is introduced. We note that, in our model a UE being synchronized to a SyncRef refers to a state, where both the UEs have similar SLSS-ID and are aligned on frame and subframe level. The SLSS is represented by MIB-SL, which is modeled as a broadcast message. MIB-SL is implemented as per the standard, [TS36331], with additional metadata fields: SLSSID, MIB-SL creation time, MIB-SL reception time and the reception offset. The S-RSRP of the SLSS received by a UE is computed as explained in the section lte-design:ref:*sec-phy-ue-measurements*. Only those SLSS are considered detected, which have the S-RSRP greater than the configured minimum S-RSRP and for which MIB-SL has been decoded successfully. The minimum S-RSRP is configured through the attribute "MinSrsrp". The modeled synchronization process can be categorized by three sequential processes:

1. Scanning
2. Measurement
3. Evaluation

To tailor these processes, the following new attributes have been introduced in the `LteUePhy` class.

- UeSlssScanningPeriod
- UeSlssInterScanningPeriodMin
- UeSlssInterScanningPeriodMax
- UeSlssMeasurementPeriod
- UeSlssEvaluationPeriod
- NSamplesSrsrpMeas

For more information about these attributes the reader is referred to the API of `LteUePhy` class. After the scanning process, only 6 detected SyncRefs with the highest S-RSRP are measured during the measurement period. Moreover, if a UE receives multiple SLSS from different UEs but have the same SLSS-ID and the reception offset, they are considered as different S-RSRP samples of the same SyncRef. The start of the measurement and evaluation processes are subject to the detection of at least 1 SynchRef during the scanning process. At least one SyncRef selection process within 20 seconds is scheduled as per the standard [TS36331]. The MIB-SL is transmitted with a fixed periodicity of 40 ms [TS36331]. The `LteUeRrc` class is responsible of scheduling and delivering the MIB-SL to `LteUePhy` class. Once a suitable SyncRef is selected, the change of timing is performed upon subframe indication, i.e., before the next subframe to avoid any miss alignments. We also note that the transmission of a MIB-SL has priority over the transmission of SCI, if they are scheduled in the same subframe. On the other hand, an MIB-SL cannot be transmitted if a SyncRef scanning or measurement process is in progress. The `LteUeMac` and `LteUeRrc` class are

notified about the successful change of timings using the existing SAP interfaces, i.e., `LteUeCphySapUser` and `LteUePhySapUser` respectively.

LteSpectrumPhy

As mentioned before, the `LteUePhy` class includes an additional `LteSpectrumPhy` instance to receive Sidelink transmissions from other UEs. Therefore, new functions are implemented in `LteSpectrumPhy` class to relay the received Sidelink transmission to the `LteUePhy` class. Those functions are RxSlPscch, RxSlPsdch, and RxSlPsbch (and corresponding callbacks for interconnecting PHY and MAC) to accomplish reception of control, discovery, and broadcast messages, respectively. To model the half duplex for Sidelink, since it is difficult for the UE to receive and transmit at the same time using the same frequency, a new attribute "CtrlFullDuplexEnabled" is introduced. This attribute is "false" by default, thus, all the Sidelink simulations are performed using half duplex mode.

To notify `LteUePhy` about the reception of SLSS a new function `SetLtePhyRxSlssCallback` was implemented, which is hooked to the `ReceiveSlss` function of `LteUePhy` through a callback, while installing UE devices in the `LteHelper` class. Similarly, the function `SetLtePhyRxPsdchSdRsrpCallback` was implemented to set the callback that hooks to the `ReceivePsdchSdRsrp` function of `LteUePhy`, which is used to measure the Sidelink Discovery Reference Signals Received Power (SD-RSRP) of each received discovery message. For other Sidelink signals, i.e., control and data the already implemented callbacks are utilized.

The `LteSpectrumPhy` class is still responsible for evaluating the TB BLER, however, with the introduction of new physical Sidelink channels, i.e. PSCCH, PSSCH, PSDCH, and PSBCH, a new physical error model `LteNistErrorModel` is implemented. This error model uses the BLER vs SNR curves for LSM from [NISTBLERD2D], for all the Sidelink physical channels. In particular, these curves are obtained by extending the LTE toolbox in Matlab and performing Monte Carlo simulations by considering AWGN channel and SISO mode for transmission. In the following, we plot these curves for each Sidelink physical channel.

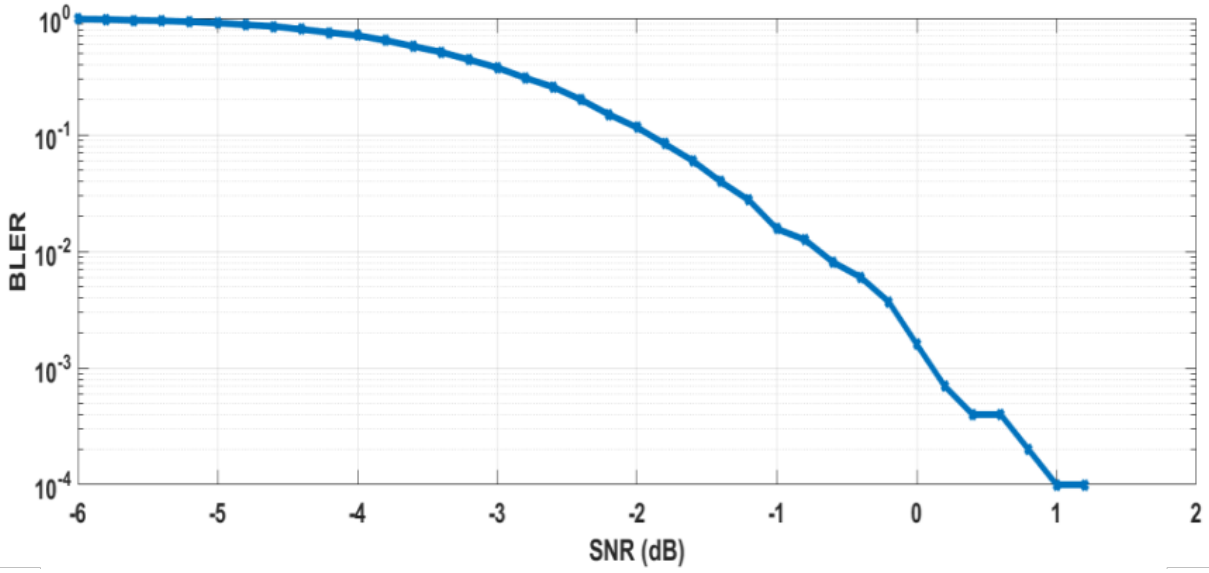


Fig. 16: BLER vs SNR of PSCCH (First transmission)

Different from the already existing error models, e.g., `LteMiErrorModel`, which uses the Mutual Information Based Effective SINR (MIESM) technique for soft combining process lte-design:ref:*sec-data-phy-error-model*, the `LteNistErrorModel` uses a weighted averaging algorithm explained in [NISTBLERD2D] for this purpose. Therefore, a new class `LteSlHarq` is implemented to store the SINR values of each Sidelink transmission that is used for soft combining process of the retransmission. It is to be noted that, as per the standard, no HARQ feedback is available for any Sidelink physical channels. Moreover, to handle interference in D2D scenarios, since all the Sidelink

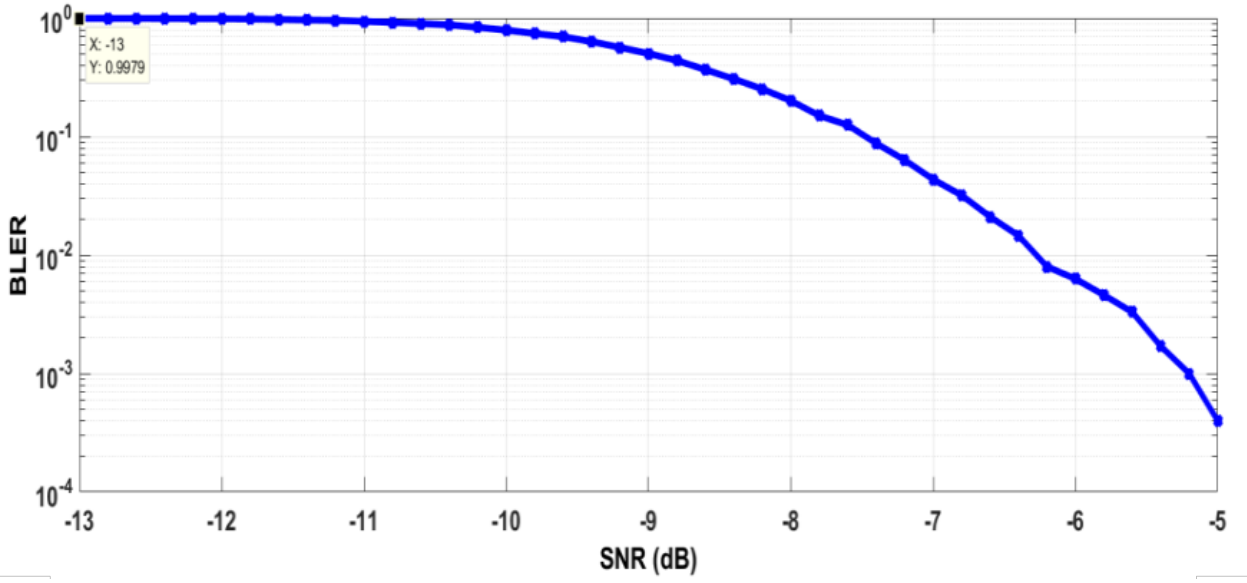


Fig. 17: BLER vs SNR of PSBCH

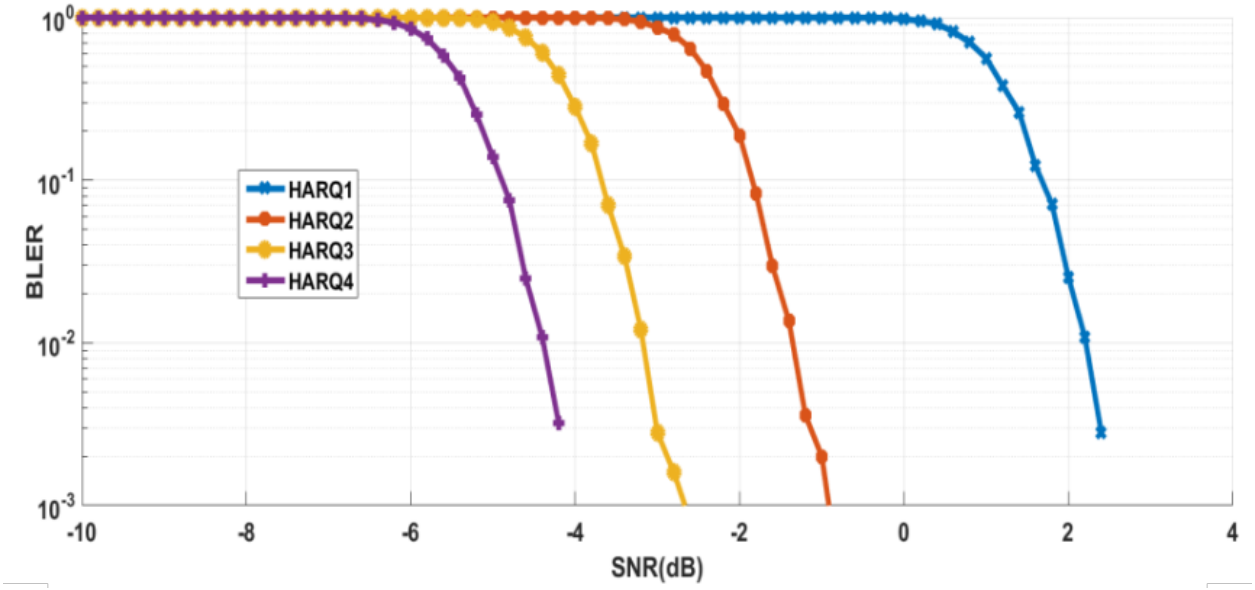


Fig. 18: BLER vs SNR of PSDCH

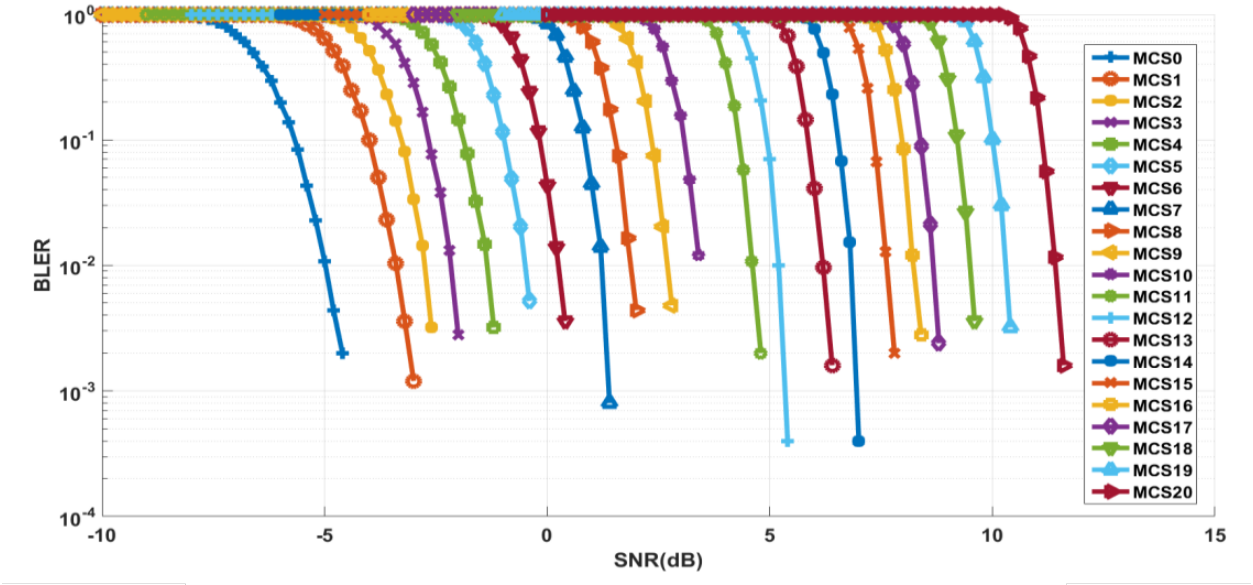


Fig. 19: BLER vs SNR of PSSCH (HARQ 1)

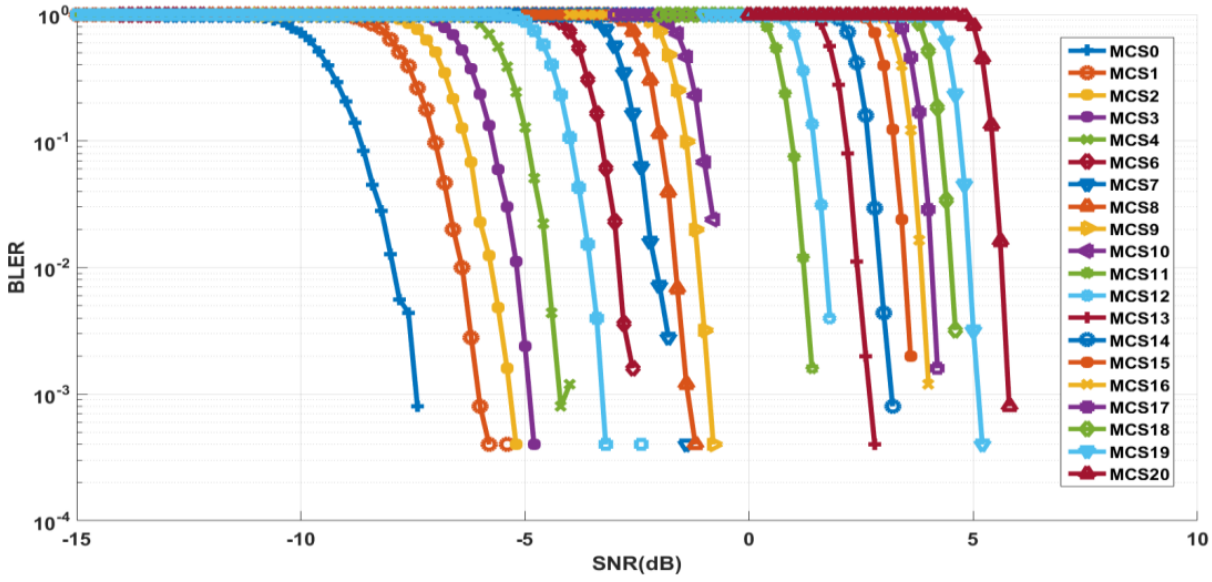


Fig. 20: BLER vs SNR of PSSCH (HARQ 2)

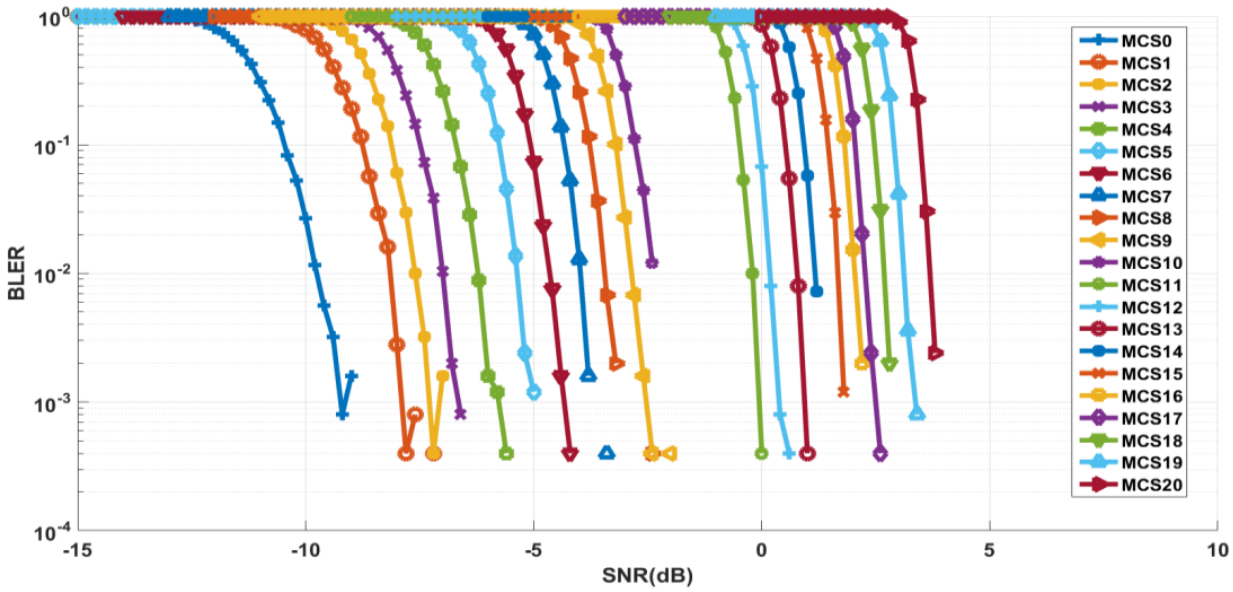


Fig. 21: BLER vs SNR of PSSCH (HARQ 3)

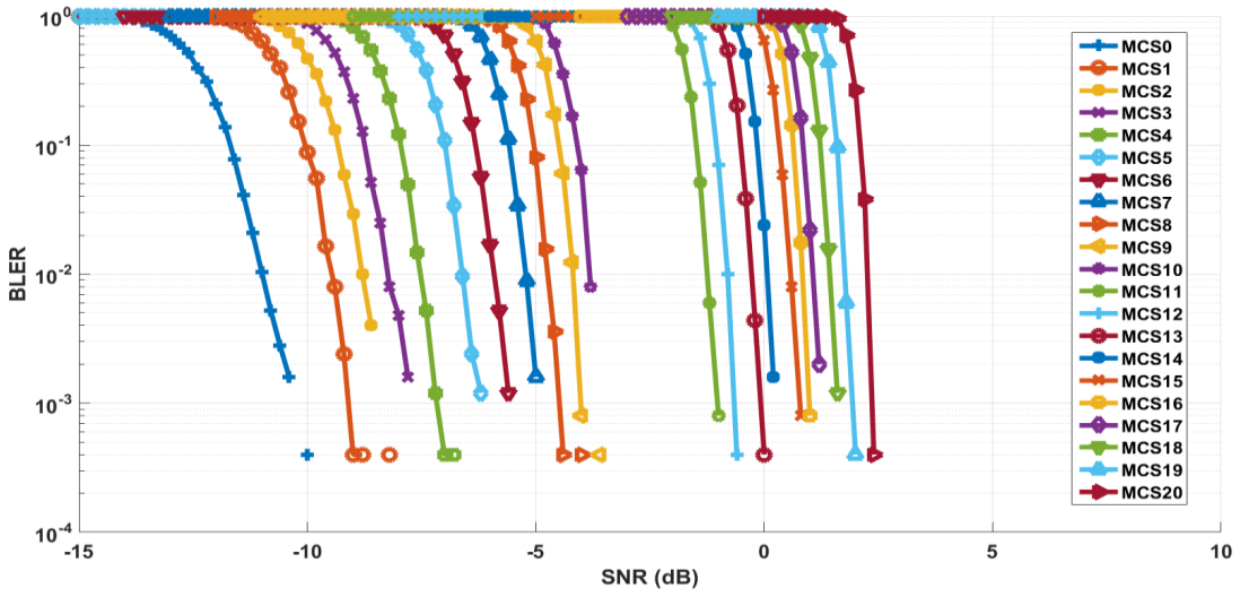


Fig. 22: BLER vs SNR of PSSCH (HARQ 4)

physical channels employ broadcast solution, a new interference model is implemented. This is needed because unlike LTE implementation in which unwanted signals are filtered on the basis of a cell id embedded in transmitted/received `LteSpectrumSignalParameters`, for D2D scenarios two signals received at the same time could be intended for a single UE. This overlapping of signal could occur in out-of-coverage scenarios, where two UEs pick an identical resource to transmit. To tackle such conditions two new classes, `LteSlInterference` and `LteSlChunkProcessor` are implemented.

The `LteSpectrumPhy` class, at the time of reception maintains a vector to store the information about the received signal(s). These signals including the overlapping ones (i.e., they occur at the same time and have equal duration) are passed to `LteSlInterference` class, which maintains the indexing of each signal by storing them into a vector. Once the signal(s) duration is elapsed, a call to `ConditionallyEvaluateChunk` function is invoked. This function iterates over this vector and computes the perceived SINR, interference, and the power of each signal, which are then passed to the respective `LteSlChunkProcessor` instance along with their signal id. Similarly, each `LteSlChunkProcessor` instance maintain its assigned values by storing them in a vector as per the signal id. For example, an `LteSlChunkProcessor` initialized to store the computed SINR values will maintain a vector containing the SINR of each overlapping signal as per their signal id. In the end, each vector is passed to the respective function, which are hooked through a callback in `LteHelper` class at the time of installing the UE device. For instance, the `UpdateSlSinrPerceived` function of `LteSpectrumPhy` receive the vector of perceived SINR values.

Based on the BLER computation in `LteSpectrumPhy`, the messages received on Sidelink physical channels can be divided into three types.

1. The messages whose BLER computation is performed without HARQ, e.g., SCI and MIB-SL
2. The messages whose BLER computation is performed with HARQ and follow the control information, e.g., Sidelink data
3. The messages whose BLER computation is performed with HARQ but do not have the control information, e.g., Discovery messages

For first type of messages, we compute the average SINR per RB for each received TB and sort them in the descending order of the SINR. Then, we try to decode them by computing the BLER with the help of `LteNistErrorModel`. For the second type of signals, we have the transport block information available in `m_expectedSlTbs` given by `LteUePhy` class. With this information, we perform a TB to SINR index mapping and retrieve the SINR of the expected TB from the perceived SINR vector. Then, this SINR and the HARQ info (if it is the retransmission) along with other parameters are used to compute the BLER of this message. Finally, the third type of messages, similar to the first type, are decoded by first storing them in descending order of the SINR, since we do not have the prior information about the TB. The only difference, is that the discovery messages can be retransmitted up to 3 times, therefore, we maintain the HARQ history of each transmission and use it along with other parameters for computing the BLER. We also note that for type 1 and type 3 messages, if the received TBs collide, i.e., they use the same RBs, the UE will try to decode the sorted TBs one at a time, and if any of the TB is decoded the remaining TB(s) are marked corrupted, thus, are not received by the UE. Alternatively, by setting the `DropRbOnCollisionEnabled` attribute all the colliding TBs can be dropped irrespective of their perceived SINR.

2.2.3 Frequency Hopping

The D2D model also supports the frequency hopping on Sidelink PSSCH for the "UeSelected" resource scheduling, i.e., MODE 2. At the time of writing this documentation, only the *inter-subframe* hopping mode, with constant (i.e., Type 1) and pseudo-random (Type 2) is supported.

Note: The documentation for this section will be extended in the later release of the D2D code. Users, interested to gain further information are referred to [NISTFREQHOPP]

2.2.4 Helpers

Four helper objects are used to setup simulations and configure the various components. These objects are:

- `LteHelper`, which takes care of the configuration of the LTE/Sidelink radio access network, as well as of coordinating the setup and release of EPS and Sidelink radio bearers and start/stop ProSe discovery. The `LteHelper` class provides both the API definition and its implementation.
- `EpcHelper`, which takes care of the configuration of the Evolved Packet Core. The `EpcHelper` class is an abstract base class, which only provides the API definition; the implementation is delegated to the child classes in order to allow for different EPC network models.
- `CcHelper`, which takes care of the configuration of the `LteEnbComponentCarrierMap`, basically, it creates a user specified number of `LteEnbComponentCarrier`. `LteUeComponentCarrierMap` is currently created starting from the `LteEnbComponentCarrierMap`. `LteHelper:InstallSingleUeDevice`, in this implementation, is needed to invoke after the `LteHelper:InstallSingleEnbDevice` to ensure that the `LteEnbComponentCarrierMap` is properly initialized.
- `LteSidelinkHelper`, this helper class is provided to ease the burden of a user to configure public safety scenarios involving Sidelink. In particular, it uses other helper classes, e.g., LteHelper to activate/deactivate Sidelink bearers, create groups for Sidelink broadcast or groupcast communication and Lte3gppHexGridEnbTopologyHelper to associate UEs to a Sidelink group or an eNB using wrap around method.

Note: Wrap-around functionality is not fully supported yet.

It is possible to create a simple LTE-only simulation by using the `LteHelper` alone, or to create complete LTE-EPC simulations by using both `LteHelper` and `EpcHelper`. When both helpers are used, they interact in a master-slave fashion, with the `LteHelper` being the Master that interacts directly with the user program, and the `EpcHelper` working "under the hood" to configure the EPC upon explicit methods called by the `LteHelper`. The exact interactions are displayed in the Figure lte-design:ref:*fig-helpers*.

2.3 User Documentation

2.3.1 Sidelink simulation output

Similarly, for the Sidelink the LTE model currently supports the output to file of PHY, MAC and RRC level KPIs. You can enable them in the following way:

```
Ptr<LteHelper> lteHelper = CreateObject<LteHelper>();

// configure all the simulation scenario here...

// Transmission traces
lteHelper->EnableSlPscchMacTraces();
lteHelper->EnableSlPsschMacTraces();
lteHelper->EnableSlPsdchMacTraces();

// Reception traces
lteHelper->EnableSlRxPhyTraces();
lteHelper->EnableSlPscchRxPhyTraces();
lteHelper->EnableDiscoveryMonitoringRrcTraces();

Simulator::Run();
```

A helper method exists to enable all six traces above in one statement:

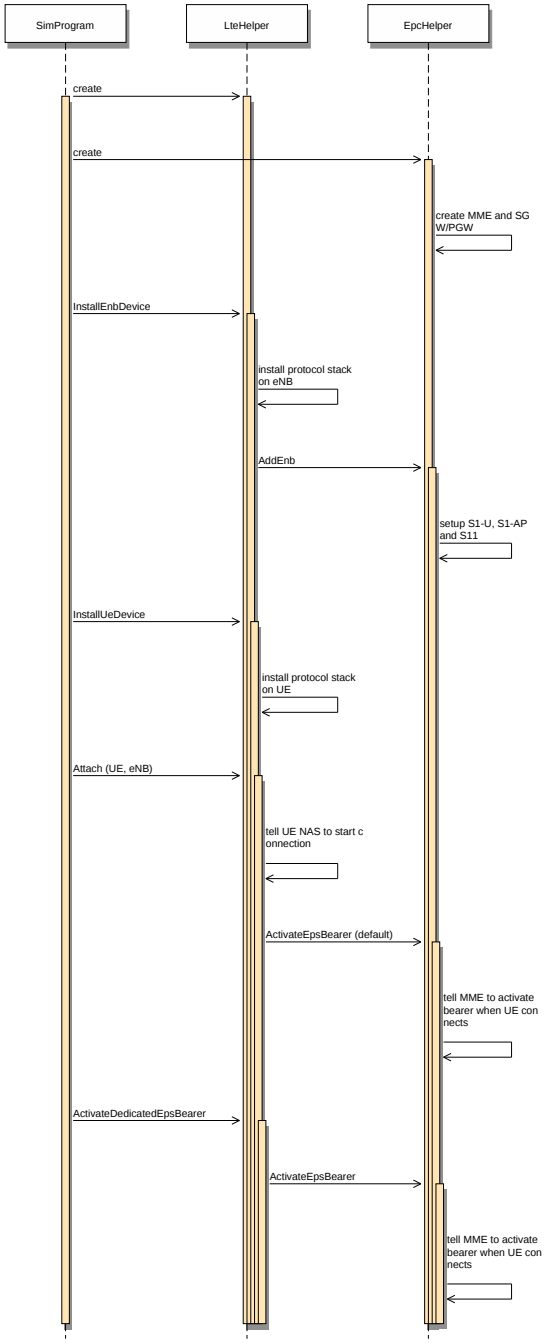


Fig. 23: Sequence diagram of the interaction between `LteHelper` and `EpcHelper`.

```
Ptr<LteHelper> lteHelper = CreateObject<LteHelper>();

// configure all the simulation scenario here...

lteHelper->EnableSidelinkTraces();

Simulator::Run();
```

The Sidelink PSCCH and PSSCH MAC level KPIs are gathered at UE MAC and are written into two files, first one for PSCCH and the second for PSSCH. These KPIs are basically the trace of resource allocation for the Sidelink control and data transmissions for every SC period in MODE 1 and MODE 2. For PSCCH KPIs the format is the following:

1. Simulation time in milliseconds
2. Cell ID
3. unique UE ID (IMSI)
4. Sidelink-specific UE ID (RNTI)
5. Current frame number
6. Current subframe number
7. Sidelink Control (SC) period starting frame number
8. Sidelink Control (SC) period starting subframe number
9. PSCCH resource index
10. Transport block size (PSCCH)
11. Total number of RBs allocated for PSCCH
12. Index of the first RB used for PSCCH
13. Hopping flag
14. Hopping value (value of 255 is assigned when hopping is not enabled)
15. Total number of RBs allocated for PSSCH
16. Index of the first RB used for PSSCH
17. Time Resource Pattern Index (iTRP) used for PSSCH
18. MCS
19. Layer 1 group destination id (8 LSBs of the L2 group destination id at UE MAC)
20. Drop flag (dropped = 1 : not dropped = 0)

while for PSSCH MAC KPIs the format is:

1. Simulation time in milliseconds
2. Cell ID
3. unique UE ID (IMSI)
4. Sidelink-specific UE ID (RNTI)
5. Current frame number
6. Current subframe number
7. Frame number at which the SC period starts

8. Subframe number at which the SC period starts
9. Total number of RBs allocated for PSSCH
10. Index of the first RB used for PSSCH
11. MCS
12. Transport block size
13. Redundancy version
14. Drop flag (dropped = 1 : not dropped = 0)

Similarly, the file containing the PSDCH MAC KPIs have the following content:

1. Simulation time in milliseconds
2. Cell ID
3. unique UE ID (IMSI)
4. Sidelink-specific UE ID (RNTI)
5. Current frame number
6. Current subframe number
7. Frame number at which the discovery period starts
8. Subframe number at which the discovery period starts
9. PSDCH resource index
10. Total number of RBs allocated for PSSCH
11. Index of the first RB used for PSSCH
12. MCS
13. Transport block size
14. Redundancy version
15. Discovery message type
16. Discovery content type
17. Discovery model
18. Content. The information in this field vary depending on the type of UE:
    - `Normal UE:` Application code
    - `Relay UE:` Relay Service Code; Announcer info; Relay UE Id; Status indicator; Spare
19. Drop flag (dropped = 1 : not dropped = 0)

The names of the files used for Sidelink MAC KPI output can be customized via the ns-3 attributes:

- `ns3::MacStatsCalculator::SlCchOutputFilename`
- `ns3::MacStatsCalculator::SlSchOutputFilename`
- `ns3::MacStatsCalculator::SlDchOutputFilename`

Sidelink PHY KPIs are distributed in two different files whose names are configurable through the following attributes:

- `ns3::PhyRxStatsCalculator::SlCchRxOutputFilename`
- `ns3::PhyRxStatsCalculator::SlRxOutputFilename`

In the SlPscchRx file, the following content is available:

1. Simulation time in milliseconds
2. Cell ID (Fixed to 0 for Sidelink)
3. Unique UE ID (IMSI)
4. Sidelink-specific UE ID (RNTI) **Note: It is the RNTI of transmitting UE**
5. PSCCH resource index
6. Transport block size (PSCCH) in bytes
7. Hopping flag
8. Hopping value (value of 255 is assigned when hopping is not enabled)
9. Total number of RBs for PSSCH
10. Index of the first RB used for PSSCH
11. Time Resource Pattern Index (iTRP) used for PSSCH
12. MCS
13. Layer 1 group destination id (8 LSBs of the L2 group destination id at UE MAC)
14. Correctness in the reception of the TB (correct = 1 : error = 0)

The parameters of SlRx file, which stores the PSSCH, PSDCH and PSBCH PHY reception results are:

1. Simulation time in milliseconds
2. Cell ID (Fixed to 0 for Sidelink)
3. Unique UE ID (IMSI)
4. Sidelink-specific UE ID (RNTI) **Note: It is the RNTI of transmitting UE**
5. Layer of transmission
6. MCS
7. Transport block size in bytes
8. Redundancy version
9. New Data Indicator flag
10. Correctness in the reception of the TB (correct = 1 : error = 0)
11. Average perceived SINR per RB in linear units

Note: At the time of writing this documentation, the RNTI for the PSBCH signals is not reported, thus, is set to zero.

And Finally, the name of the file, which stores the RRC level KPIs for PSDCH monitoring is configurable through the attribute `ns3::RrcStatsCalculator::SlDchRxRrcOutputFilename`. The content of this file is the following:

1. Simulation time in milliseconds
2. Cell ID
3. unique UE ID (IMSI)
4. Sidelink-specific UE ID (RNTI)
5. Discovery message type

6. Discovery content type
7. Discovery model
8. Content. The information in this field vary depending on the type of UE:
    - `Normal UE:` Application code
    - `Relay UE:` Relay Service Code; Announcer info; Relay UE Id; Status indicator; Spare

2.3.2 LTE Sidelink PHY Error Model

The Sidelink physical error model consists of the data, control, and the discovery error model. All of them are active by default. It is possible to deactivate them with the ns3 attribute system, in detail:

```
Config::SetDefault("ns3::LteSpectrumPhy::SlCtrlErrorModelEnabled",␣
↪BooleanValue(false));
Config::SetDefault("ns3::LteSpectrumPhy::SlDataErrorModelEnabled",␣
↪BooleanValue(false));
Config::SetDefault("ns3::LteSpectrumPhy::SlDiscoveryErrorModelEnabled",␣
↪BooleanValue(false));
```

Besides the error models, one more attribute, i.e., "DropRbOnCollisionEnabled" is also introduced only for the Sidelink transmissions. This is implemented by keeping in mind the scenarios in which the resources are autonomously scheduled by a UE, which increases the probability of two UEs choosing the same RBs to transmit. Therefore, causing a collision between the TBs. By using this attribute, a user can choose to drop such collided TBs. It can be configured as follows:

```
Config::SetDefault("ns3::LteSpectrumPhy::DropRbOnCollisionEnabled",␣
↪BooleanValue(true));
```

2.3.3 LTE Sidelink

A new directory "d2d-examples" in the LTE module examples directory holds the examples for simulating ProSe services, i.e., direct communication, direct discovery, synchronization, and UE-to-Network Relay. These examples can be divided into the following two categories:

1. Simple examples

These examples cover the different ProSe services using simple scenarios. The examples included in this category are the following:

- *Direct communication:*
    - lte-sl-in-covrg-comm-mode1
    - lte-sl-in-covrg-comm-mode2
    - lte-sl-out-of-covrg-comm
- *Direct discovery:*
    - lte-sl-in-covrg-discovery
    - lte-sl-in-covrg-discovery-multi
    - lte-sl-in-covrg-discovery-collision
    - lte-sl-out-of-covrg-discovery-collision
- *Synchronization:*

  - lte-sl-out-of-covrg-synch
  - lte-sl-out-of-covrg-synch-twotx
- *UE-to-Network Relay:*
  - lte-sl-in-covrg-relay
  - lte-sl-in-covrg-relay-building
  - lte-sl-relay-cluster
  - lte-sl-relay-cluster-selection
- *Multicell scenarios:*
  - lte-sl-in-covrg-comm-mode1-multicell
  - lte-sl-in-covrg-comm-mode2-multicell
  - lte-sl-in-covrg-discovery-multicell
  - lte-sl-in-covrg-relay-mode2-multicell

2. Detailed examples

These examples can be used to simulate out-of-coverage ProSe communication, discovery and synchronization. In particular, these examples covers the simulation scenarios used for the study published in [NIST2017]. The users interested in the extensive simulation campaigns described in this study, are referred to a "README.txt" file for more information. The examples included in this category are:

- wns3-2017-pscch
- wns3-2017-pssch
- wns3-2017-synch
- wns3-2017-discovery

The Sidelink example scripts follow the same rules as writing an LTE simulation script, however, there are additional configurations required to simulate Sidelink. Following are some rules of thumb for writing Sidelink scripts;

- Sidelink simulations require EPC.
- eNB should be disabled to simulate out-of-coverage scenarios.
- Sidelink pools should be configured before the start of the simulation.
- `Lte3gppHexGridEnbTopologyHelper` cannot be use without initializing eNB nodes and appropriate antenna model. * It is possible to avoid the initialization of eNB nodes in out-of-coverage scenarios, if hexagonal ring topology is not used.

In general, all the D2D examples are highly parameterizable and could be divided in the following parts:

1. Configuration of LTE and Sidelink default parameters, e.g., the ones configured by calling `Config::SetDefault` function
2. Topology configuration
3. Sidelink pool configuration
4. IP configuration (if any)
5. Application configuration (if any)
6. Sidelink bearer configuration (if any)
7. Service configuration (if any)

In the following, we will discuss four examples from the "Simple examples" category (i.e., lte-sl-in-covrg-comm-mode1, lte-sl-in-covrg-comm-mode2, lte-sl-out-of-covrg-comm, and lte-sl-in-covrg-relay) and two examples from the "Detailed examples" category, which are wns3-2017-synch and wns3-2017-discovery. We choose wns3-2017-synch, since it also simulates the PSCCH and PSSCH.

lte-sl-in-covrg-comm-mode1

This example simulates an in-coverage MODE 1 ProSe communication by using the following scenario.

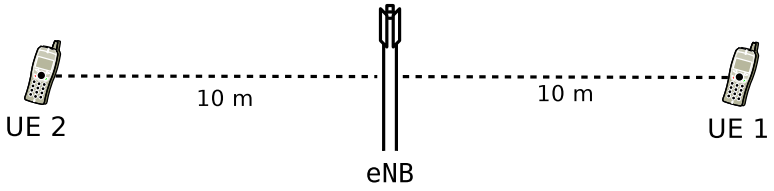


Fig. 24: Sidelink simple in-coverage scenario

Both the UEs are in-coverage of the eNB where UE1 sends the data to UE2 via Sidelink by using the resources assigned by the eNB. A user can configure the simulation time and the output of NS logs of the specified classes by using the corresponding command line variables in the simulation script. For example, a user can run the simulation as follows:

```
./ns3 run "lte-sl-in-covrg-comm-mode1 --simTime=7 --enableNsLogs=false"
```

The simulation time is in seconds. Moreover, this example can support IPv6 instead of the default IPv4 if the `--useIPv6` command-line argument is provided.

Configuration of LTE and Sidelink default parameters

The simulation script begins with the configuration of the parameters of the Sidelink scheduler as follows.

- Configure the Sidelink scheduler:

```
Config::SetDefault("ns3::RrSlFfMacScheduler::Itrp", UintegerValue(0));
Config::SetDefault("ns3::RrSlFfMacScheduler::SlGrantSize", UintegerValue(5));
```

The `ns3::RrSlFfMacScheduler` is a very simple round robin scheduler, which uses a fixed TRP index value and number of RBs to be allocated to a UE.

- Configure the frequency and the bandwidth:

```
Config::SetDefault("ns3::LteEnbNetDevice::DlEarfcn", UintegerValue(100));
Config::SetDefault("ns3::LteUeNetDevice::DlEarfcn", UintegerValue(100));
Config::SetDefault("ns3::LteEnbNetDevice::UlEarfcn", UintegerValue(18100));
Config::SetDefault("ns3::LteEnbNetDevice::DlBandwidth", UintegerValue(50));
Config::SetDefault("ns3::LteEnbNetDevice::UlBandwidth", UintegerValue(50));
```

We use the LTE Band 1 for both LTE and Sidelink communication.

- Configure the error models:

```
// For PSSCH
Config::SetDefault("ns3::LteSpectrumPhy::SlDataErrorModelEnabled",
                                                    BooleanValue(true));
```

(continues on next page)

(continued from previous page)

```
// For PSCCH
Config::SetDefault("ns3::LteSpectrumPhy::SlCtrlErrorModelEnabled",
                                                    BooleanValue(true));

Config::SetDefault("ns3::LteSpectrumPhy::DropRbOnCollisionEnabled",
                                                    BooleanValue(false));
```

- Configure the transmit power of the eNB and the UEs:

```
Config::SetDefault("ns3::LteUePhy::TxPower", DoubleValue(23.0));
Config::SetDefault("ns3::LteEnbPhy::TxPower", DoubleValue(30.0));
```

The powers configured are in dBm.

Topology configuration

- Instantiating LTE, EPC and Sidelink helpers:

```
Ptr<LteHelper> lteHelper = CreateObject<LteHelper>();
Ptr<PointToPointEpcHelper> epcHelper = CreateObject<PointToPointEpcHelper>();
lteHelper->SetEpcHelper(epcHelper);

Ptr<LteSidelinkHelper> proseHelper = CreateObject<LteSidelinkHelper>();
proseHelper->SetLteHelper(lteHelper);
```

- Configuring the pathloss model:

```
lteHelper->SetAttribute("PathlossModel",
                            StringValue("ns3::Cost231PropagationLossModel"));
```

- Enabling the Sidelink:

```
lteHelper->SetAttribute("UseSidelink", BooleanValue(true));
```

Note : Attribute ‘‘UseSidelink‘‘ must be set before installing the UE devices.

- Configuring the scheduler:

```
lteHelper->SetSchedulerType("ns3::RrSlFfMacScheduler");
```

- Creating the eNB and UE nodes and setting their mobility:

```
NodeContainer enbNode;
enbNode.Create(1);
NodeContainer ueNodes;
ueNodes.Create(2);

Ptr<ListPositionAllocator> positionAllocEnb =
                                    CreateObject<ListPositionAllocator>();
positionAllocEnb->Add (Vector(0.0, 0.0, 30.0));
Ptr<ListPositionAllocator> positionAllocUe1 =
                                    CreateObject<ListPositionAllocator>();
positionAllocUe1->Add (Vector(10.0, 0.0, 1.5));
Ptr<ListPositionAllocator> positionAllocUe2 =
```

(continues on next page)

(continued from previous page)

```
                                        CreateObject<ListPositionAllocator>();
positionAllocUe2->Add (Vector(-10.0, 0.0, 1.5));

//Install mobility
MobilityHelper mobilityeNodeB;
mobilityeNodeB.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityeNodeB.SetPositionAllocator(positionAllocEnb);
mobilityeNodeB.Install(enbNode);

MobilityHelper mobilityUe1;
mobilityUe1.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe1.SetPositionAllocator(positionAllocUe1);
mobilityUe1.Install(ueNodes.Get(0));

MobilityHelper mobilityUe2;
mobilityUe2.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe2.SetPositionAllocator(positionAllocUe2);
mobilityUe2.Install(ueNodes.Get(1));
```

- Installing LTE devices to the nodes:

```
NetDeviceContainer enbDevs = lteHelper->InstallEnbDevice(enbNode);
NetDeviceContainer ueDevs = lteHelper->InstallUeDevice(ueNodes);
```

Sidelink pool configuration

This example simulates an in-coverage scenario, therefore, the eNB will be configured with a pre-configured dedicated Sidelink pool. As mentioned in *RRC*, in this scenario the `LteSlEnbRrc` class will be responsible for holding this per-configured pool configuration. The pool configuration starts by setting a flag in `LteSlEnbRrc` as an indication that the Sidelink is enabled. It is configured as follows:

```
Ptr<LteSlEnbRrc> enbSidelinkConfiguration = CreateObject<LteSlEnbRrc> ();
enbSidelinkConfiguration->SetSlEnabled (true);
```

For configuring Sidelink parameters, the "setup" structure of the field "commTxResources-r12" of IE "SL-Preconfiguration" defined in the standard [TS36331] is converted into a C++ structure. This example uses this structure to configure the pool parameters for Sidelink control. The pool configuration is done by using the `LteSlPreconfigPoolFactory` in the following manner,

```
LteRrcSap::SlCommTxResourcesSetup pool;

pool.setup = LteRrcSap::SlCommTxResourcesSetup::SCHEDULED;
//BSR timers
pool.scheduled.macMainConfig.periodicBsrTimer.period =
                                          LteRrcSap::PeriodicBsrTimer::sf16;
pool.scheduled.macMainConfig.retxBsrTimer.period = LteRrcSap::RetxBsrTimer::sf640;
//MCS
pool.scheduled.haveMcs = true;
pool.scheduled.mcs = 16;
//resource pool
LteSlResourcePoolFactory pfactory;
pfactory.SetHaveUeSelectedResourceConfig(false); //since we want eNB to schedule

//Control
```

(continues on next page)

(continued from previous page)

```
pfactory.SetControlPeriod("sf40");
pfactory.SetControlBitmap(0x00000000FF); //8 subframes for PSCCH
pfactory.SetControlOffset(0);
pfactory.SetControlPrbNum(22);
pfactory.SetControlPrbStart(0);
pfactory.SetControlPrbEnd(49);

pool.scheduled.commTxConfig = pfactory.CreatePool();

uint32_t groupL2Address = 255;

enbSidelinkConfiguration->AddPreconfiguredDedicatedPool(groupL2Address, pool);
lteHelper->InstallSidelinkConfiguration(enbDevs, enbSidelinkConfiguration);
```

The resources for data are computed by the scheduler on the basis of the scheduler's attributes configured in the start of this simulation script and the above pool configuration.

Similarly, for the UEs we need to enable the Sidelink in `LteSlUeRrc` by setting a flag and a pre-configuration object, which will be initialized with the pool configurations once the UE receives an `RrcConnectionReconfiguration` message from the eNB, as shown in Figures *ns-3 LTE Sidelink in-coverage radio bearer activation (Tx)* and *ns-3 LTE Sidelink in-coverage radio bearer activation (Rx)*:

```
//pre-configuration for the UEs
Ptr<LteSlUeRrc> ueSidelinkConfiguration = CreateObject<LteSlUeRrc>();
ueSidelinkConfiguration->SetSlEnabled(true);
LteRrcSap::SlPreconfiguration preconfiguration;
ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);
lteHelper->InstallSidelinkConfiguration(ueDevs, ueSidelinkConfiguration);
```

IP configuration

- Installing the IP stack on the UEs and assigning IP address:

```
//Install the IP stack on the UEs and assign IP address
InternetStackHelper internet;
internet.Install(ueNodes);
Ipv4InterfaceContainer ueIpIface;
ueIpIface = epcHelper->AssignUeIpv4Address(NetDeviceContainer(ueDevs));

// set the default gateway for the UE
Ipv4StaticRoutingHelper ipv4RoutingHelper;
for (uint32_t u = 0; u < ueNodes.GetN(); ++u)
  {
    Ptr<Node> ueNode = ueNodes.Get(u);
    // Set the default gateway for the UE
    Ptr<Ipv4StaticRouting> ueStaticRouting =
             ipv4RoutingHelper.GetStaticRouting(ueNode->GetObject<Ipv4>());
    ueStaticRouting->SetDefaultRoute(epcHelper->GetUeDefaultGatewayAddress(),
                                                                1);
  }
```

The above configuration is similar to any usual LTE example with EPC.

- Attaching the UEs to the eNB:

```
lteHelper->Attach(ueDevs);
```

Application configuration

- Installing applications and activating Sidelink radio bearers. Note that two configuration options (IPv4 and IPv6) are supported; we only document below the IPv4 path through the code:

```
Ipv4Address groupAddress4("225.0.0.0"); //use multicast address as destination
...
remoteAddress = InetSocketAddress(groupAddress4, 8000);
localAddress = InetSocketAddress(Ipv4Address::GetAny(), 8000);
...
OnOffHelper sidelinkClient("ns3::UdpSocketFactory", remoteAddress);
sidelinkClient.SetConstantRate(DataRate("16kb/s"), 200);

ApplicationContainer clientApps = sidelinkClient.Install(ueNodes.Get(0));
//onoff application will send the first packet at :
//(2.9 (App Start Time) + (1600 (Pkt size in bits) / 16000 (Data rate)) = 3.0 sec
clientApps.Start(slBearersActivationTime + Seconds(0.9));
clientApps.Stop(simTime - slBearersActivationTime + Seconds(1.0));

ApplicationContainer serverApps;
PacketSinkHelper sidelinkSink("ns3::UdpSocketFactory", localAddress);
serverApps = sidelinkSink.Install(ueNodes.Get (1));
serverApps.Start(Seconds(2.0));
```

In this example, an "OnOff" application is installed in the UE 1, which sends a 200 byte packet with the constant bit rate of 16 kb/s. On the other hand, the UE 2 is configured with the "PacketSink" application.

The Sidelink radio bearers are activated by calling the `ActivateSidelinkBearer` method of `LteSidelinkHelper` as follows:

```
//Set Sidelink bearers
Ptr<LteSlTft> tft = Create<LteSlTft>(LteSlTft::BIDIRECTIONAL,
                                                    groupAddress4, groupL2Address);
proseHelper->ActivateSidelinkBearer(Seconds(2.0), ueDevs, tft);
```

The `ActivateSidelinkBearer` method takes the following three parameters as input,

1. Time to activate the bearer
2. Devices for which the bearer will be activated
3. The Sidelink traffic flow template

The `tft` in this example is an object of `LteSlTft` class created by initializing its following parameters,

1. Direction of the bearer, i.e., "Transmit", "Receive" or Bidirectional
2. An IPv4 multicast address of the group
3. Sidelink layer 2 group address (used as Sidelink layer 2 group id)

- Activating the Sidelink traces:

```
lteHelper->EnableSlPscchMacTraces();
lteHelper->EnableSlPsschMacTraces();
```

(continues on next page)

(continued from previous page)

```
lteHelper->EnableSlTxPhyTraces();
lteHelper->EnableSlRxPhyTraces();
lteHelper->EnableSlPscchRxPhyTraces();
```

The above code will enable all the Sidelink related traces. We note that, the user can also use the classical function "LteHelper::EnableTraces()", but this will also output the LTE traces.

Upon the completion of the simulation, the following trace files can be found in the repository's root folder,

- SlCchMacStats.txt
- SlSchMacStats.txt
- SlCchRxPhyStats.txt
- SlRxPhyStats.txt

For more information related to the above files please refer to the *Sidelink simulation output* section.

- UePacketTrace.tr

  The information in this file is obtained by using the traces "TxWithAddresses" and "RxWithAddresses" of `OnOff` and `PacketSink` application, respectively. Following table shows the snippet of the data from this file for the two UEs,

Table 5: UePacketTrace.tr

Time (sec)	tx/rx	NodeID	IMSI	PktSize (bytes)	IP[src]	IP[dst]
3	tx	4	1	200	7.0.0.2:49153	225.0.0.0:8000
3.09693	rx	5	2	200	7.0.0.2:49153	7.0.0.3:8000

The first row shows that the UE 1 with IMSI 1 transmits a multicast packet of 200 bytes and the UE 2 receives the packet transmitted by the UE 1. As per the client's application data rate and ON time, i.e., 16 kb/s and 2 seconds, respectively, a total of 20 packets are sent and received by the transmitting and the receiving UE.

lte-sl-in-covrg-comm-mode2

This example simulates an in-coverage MODE 2 ProSe communication using the same scenario *Sidelink simple in-coverage scenario*, used for previous example. The only difference is that the resources are not scheduled by the eNB, instead, the UE 1 selects the resources autonomously from a pool specified by the eNB through `RrcConnectionReconfiguration` message. Besides using the same scenario, this example also has the same application, therefore, in the following, we will discuss only those configurations, which are MODE 2 specific. A user can run the simulation as follows:

```
./ns3 run "lte-sl-in-covrg-comm-mode2 --simTime=7 --enableNsLogs=false"
```

The simulation time is in seconds. Similarly, this example supports IPv6 as an option if the `--useIPv6` argument is provided.

Configuration of LTE and Sidelink default parameters

- Configuring parameters for PSSCH resource selection:

```
// Fixed MCS and the number of RBs
Config::SetDefault("ns3::LteUeMac::SlGrantMcs", UintegerValue(16));
Config::SetDefault("ns3::LteUeMac::SlGrantSize", UintegerValue(5));

// For selecting subframe indicator bitmap
Config::SetDefault("ns3::LteUeMac::Ktrp", UintegerValue(1));
//use default Trp index of 0
Config::SetDefault("ns3::LteUeMac::UseSetTrp", BooleanValue(true));
```

The above parameters of `LteUeMac` class enable the UE to select the time (i.e., frame/subframe) and the frequency (i.e., RBs) resources autonomously. If the attribute "UseSetTrp" is false, a UE will select the TRP index randomly from the range of values depending on the KTRP value [TS36213].

- Configure the error models:

```
// For PSSCH
Config::SetDefault("ns3::LteSpectrumPhy::SlDataErrorModelEnabled",
                                                    BooleanValue(true));

// For PSCCH
Config::SetDefault("ns3::LteSpectrumPhy::SlCtrlErrorModelEnabled",
                                                    BooleanValue(true));

Config::SetDefault("ns3::LteSpectrumPhy::DropRbOnCollisionEnabled",
                                                    BooleanValue(false));
```

Along with the error model configuration, by setting the "DropRbOnCollisionEnabled" attribute, all the TBs collided in time and frequency are dropped. This attribute could be of particular interest for the users, e.g., to study the impact of collisions in MODE 2. In this example, this attribute has no impact as only one UE is acting as a transmitter. It is included just to make the users aware of this attribute.

Topology configuration

- There is no need to configure the scheduler, since this example uses MODE 2 for resource selection.

Sidelink pool configuration

The Sidelink pool configuration is similar to the MODE 1 configuration, e.g., the configuration for Sidelink control is the same. However, the following additional configurations are needed for in-coverage MODE 2.

```
LteRrcSap::SlCommTxResourcesSetup pool;

pool.setup = LteRrcSap::SlCommTxResourcesSetup::UE_SELECTED;
pool.ueSelected.havePoolToRelease = false;
pool.ueSelected.havePoolToAdd = true;
pool.ueSelected.poolToAddModList.nbPools = 1;
pool.ueSelected.poolToAddModList.pools[0].poolIdentity = 1;

//Data
pfactory.SetDataBitmap(0xFFFFFFFFFF);
pfactory.SetDataOffset(8); //After 8 subframes of PSCCH
pfactory.SetDataPrbNum(25);
pfactory.SetDataPrbStart(0);
pfactory.SetDataPrbEnd(49);
```

Compared to MODE 1, the pool resources are now indicated as "UE_SELECTED" along with other parameters. Moreover, now we need to configure the pool parameters related to the data, i.e., PSSCH. In addition to the subframe indicator bitmap specified by KTRP and iTRP, Mode 2 introduces another level of subframe filtering for the subframe pool via "DataBitmap" to limit the number of possible values for iTRP. The PSSCH transmission occurs on the filtered subframes after applying TRP bitmap on this "DataBitmap". Users interested to learn about how it is done are referred to *4. SidelinkCommPoolPsschTestCase*.

Finally, at the end of the simulation the trace files, similar to the previous example, can be found in the repository's root folder. One important difference, compared to MODE 1, is that in the trace files "SlCchMacStats.txt" and "SlCchRxPhyStats.txt" the parameters, e.g., PSCCH resource index and the starting RB for PSSCH are randomly selected by the UE for every SC period. Furthermore, a similar amount of packets, i.e. 20, are sent and received by the UEs.

lte-sl-out-of-covrg-comm

This example simulates an out-of-coverage MODE 2 ProSe communication by using the following scenario.

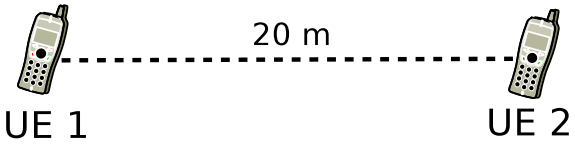


Fig. 25: Sidelink simple out-of-coverage scenario

This example allows a user to configure the simulation time and the output of NS logs of the specified classes by using the corresponding command line variables in the simulation script. For example, a user can run the simulation as follows:

```
./ns3 run "lte-sl-out-of-covrg-comm --simTime=7 --enableNsLogs=false"
```

The simulation time is in seconds, and `--useIPv6` can be used to run the example with IPv6 instead of IPv4.

Configuration of LTE and Sidelink default parameters

- Configuring parameters for PSSCH resource selection:

```
// Fixed MCS and the number of RBs
Config::SetDefault("ns3::LteUeMac::SlGrantMcs", UintegerValue(16));
Config::SetDefault("ns3::LteUeMac::SlGrantSize", UintegerValue(5));

// For selecting subframe indicator bitmap
Config::SetDefault("ns3::LteUeMac::Ktrp", UintegerValue(1));
//use default Trp index of 0
Config::SetDefault("ns3::LteUeMac::UseSetTrp", BooleanValue(true));
```

The above parameters of `LteUeMac` class enable the UE to select the time (i.e., frame/subframe) and the frequency (i.e., RBs) resources autonomously. If the attribute "UseSetTrp" is false, a UE will select the TRP index randomly from the range of values depending on the KTRP value [TS36213].

- Configure the frequency:

```
uint32_t ulEarfcn = 18100;
uint16_t ulBandwidth = 50;
```

Here, it is not necessary to configure the EARFCNs and the bandwidth of the UE and eNB for the two reasons. First, in this example we will not instantiate the eNB node, thus, setting these attributes would have no impact. Second, both the UEs will use only the Sidelink to communicate, therefore, the EARFCN and the bandwidth are specified in the pool configuration. At this stage, the above two variables are initialized to be used later to configure the pathloss model and the Sidelink pool. Similar, to the previous simple examples, we use the LTE Band 1 for Sidelink communication.

- Configure the error models:

```
// For PSSCH
Config::SetDefault("ns3::LteSpectrumPhy::SlDataErrorModelEnabled",
                                                    BooleanValue(true));

// For PSCCH
Config::SetDefault("ns3::LteSpectrumPhy::SlCtrlErrorModelEnabled",
                                                    BooleanValue(true));

Config::SetDefault("ns3::LteSpectrumPhy::DropRbOnCollisionEnabled",
                                                    BooleanValue(false));
```

Along with the error model configuration, by setting the "DropRbOnCollisionEnabled" attribute all the TBs collided in time and frequency are dropped. This attribute could be of particular interest for the users, e.g., to study the impact of collisions in MODE 2. In this example, this attribute has no impact as only one UE is acting as a transmitter. It is included just to make the users aware of this attribute.

- Configure the transmit power for the UEs:

```
Config::SetDefault("ns3::LteUePhy::TxPower", DoubleValue(23.0));
```

The power configured is in dBm.

Topology configuration

- Instantiating LTE, EPC and Sidelink helpers:

```
Ptr<LteHelper> lteHelper = CreateObject<LteHelper>();
Ptr<PointToPointEpcHelper> epcHelper = CreateObject<PointToPointEpcHelper>();
lteHelper->SetEpcHelper(epcHelper);

Ptr<LteSidelinkHelper> proseHelper = CreateObject<LteSidelinkHelper>();
proseHelper->SetLteHelper(lteHelper);
```

- Enabling the Sidelink:

```
lteHelper->SetAttribute("UseSidelink", BooleanValue(true));
```

Note : Attribute ‘‘UseSidelink‘‘ must be set before installing the UE devices.

- Configuring the pathloss model and bypass the use of eNB nodes

```
(1)

  lteHelper->SetAttribute("PathlossModel",
                          StringValue("ns3::Cost231PropagationLossModel"));

(2)
```

(continues on next page)

(continued from previous page)

```
  lteHelper->Initialize();

(3)

  double ulFreq = LteSpectrumValueHelper::GetCarrierFrequency(ulEarfcn); //18100
  NS_LOG_LOGIC ("UL freq: " << ulFreq);
  Ptr<Object> uplinkPathlossModel = lteHelper->GetUplinkPathlossModel();
  Ptr<PropagationLossModel> lossModel = uplinkPathlossModel->
                                        GetObject<PropagationLossModel>();
  NS_ABORT_MSG_IF (lossModel == NULL, "No PathLossModel");
  bool ulFreqOk = uplinkPathlossModel->
                    SetAttributeFailSafe("Frequency", DoubleValue(ulFreq));

  if (!ulFreqOk)
  {
      NS_LOG_WARN ("UL propagation model does not have a Frequency attribute");
  }
```

The use of eNB nodes can be bypassed by using the above commands strictly in the order they are listed. The command "lteHelper->Initialize ()" basically performs the channel model initialization of all the component carriers. Therefore, it is necessary to configure any desired pathloss model before issuing this command. The commands in step 3 are to properly configure the frequency attribute of the pathloss model used, which is normally done in `InstallSingleEnb` method of `LteHelper`.

- Creating the UE nodes and setting their mobility:

```
NodeContainer ueNodes;
ueNodes.Create(2);

Ptr<ListPositionAllocator> positionAllocUe1 =
                                      CreateObject<ListPositionAllocator>();
positionAllocUe1->Add(Vector(0.0, 0.0, 1.5));
Ptr<ListPositionAllocator> positionAllocUe2 =
                                      CreateObject<ListPositionAllocator>();
positionAllocUe2->Add (Vector(20.0, 0.0, 1.5));

//Install mobility
MobilityHelper mobilityUe1;
mobilityUe1.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe1.SetPositionAllocator(positionAllocUe1);
mobilityUe1.Install(ueNodes.Get(0));

MobilityHelper mobilityUe2;
mobilityUe2.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe2.SetPositionAllocator(positionAllocUe2);
mobilityUe2.Install(ueNodes.Get(1));
```

- Installing LTE devices to the nodes:

```
NetDeviceContainer ueDevs = lteHelper->InstallUeDevice(ueNodes);
```

Sidelink pool configuration

This example simulates an out-of-coverage scenario, therefore, both the UEs are configured with a pre-configured Sidelink communication pool. As mentioned in *RRC*, in this scenario the `LteSlUeRrc` class will be responsible for

holding this per-configured pool configuration. The pool configuration starts by setting a flag in `LteSlUeRrc` as an indication that the Sidelink is enabled. It is configured as follows:

```
Ptr<LteSlUeRrc> ueSidelinkConfiguration = CreateObject<LteSlUeRrc>();
ueSidelinkConfiguration->SetSlEnabled(true);
```

For configuring Sidelink communication pre-configured pool parameters, the IE "SL-Preconfiguration" defined in the standard [TS36331] is converted into a C++ structure and similar to the basic LTE layer 3 messages it can be found in `LteRrcSap` class. This example uses this structure to configure the Sidelink communication pool parameters. The pool configuration is done by using the `LteSlPreconfigPoolFactory` in the following manner,

```
LteRrcSap::SlPreconfiguration preconfiguration;

preconfiguration.preconfigGeneral.carrierFreq = ulEarfcn; //18100
preconfiguration.preconfigGeneral.slBandwidth = ulBandwidth; // 50 RBs
preconfiguration.preconfigComm.nbPools = 1;

LteSlPreconfigPoolFactory pfactory;

//Control
pfactory.SetControlPeriod("sf40");
pfactory.SetControlBitmap(0x00000000FF); //8 subframes for PSCCH
pfactory.SetControlOffset(0);
pfactory.SetControlPrbNum(22);
pfactory.SetControlPrbStart(0);
pfactory.SetControlPrbEnd(49);

//Data
pfactory.SetDataBitmap(0xFFFFFFFFFF);
pfactory.SetDataOffset(8); //After 8 subframes of PSCCH
pfactory.SetDataPrbNum(25);
pfactory.SetDataPrbStart(0);
pfactory.SetDataPrbEnd(49);

preconfiguration.preconfigComm.pools[0] = pfactory.CreatePool();

ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);
lteHelper->InstallSidelinkConfiguration (ueDevs, ueSidelinkConfiguration);
```

In addition to the subframe indicator bitmap specified by KTRP and iTRP, Mode 2 introduces another level of subframe filtering for the subframe pool via "DataBitmap" to limit the number of possible values for iTRP. The PSSCH transmission occurs on the filtered subframes after applying TRP bitmap on this "DataBitmap". Users interested to learn about how it is applied are referred to *4. SidelinkCommPoolPsschTestCase*.

IP configuration

- Installing the IP stack on the UEs and assigning IP addresses (IPv4 code statements shown below):

```
InternetStackHelper internet;
internet.Install(ueNodes);
uint32_t groupL2Address = 255;
Ipv4Address groupAddress4("225.0.0.0");//use multicast address as destination
remoteAddress = InetSocketAddress(groupAddress4, 8000);
localAddress = InetSocketAddress(Ipv4Address::GetAny(), 8000);
...
Ipv4InterfaceContainer ueIpIface;
```

(continues on next page)

(continued from previous page)

```
ueIpIface = epcHelper->AssignUeIpv4Address(NetDeviceContainer(ueDevs));
Ipv4StaticRoutingHelper ipv4RoutingHelper;
for (uint32_t u = 0; u < ueNodes.GetN(); ++u)
  {
    Ptr<Node> ueNode = ueNodes.Get(u);
    // Set the default gateway for the UE
    Ptr<Ipv4StaticRouting> ueStaticRouting =
             ipv4RoutingHelper.GetStaticRouting(ueNode->GetObject<Ipv4>());
    ueStaticRouting->SetDefaultRoute
                               (epcHelper->GetUeDefaultGatewayAddress(), 1);
  }
```

Application configuration

- Installing applications and activating Sidelink radio bearers:

```
//Set Application in the UEs
OnOffHelper sidelinkClient("ns3::UdpSocketFactory", remoteAddress);
sidelinkClient.SetConstantRate(DataRate("16kb/s"), 200);

ApplicationContainer clientApps = sidelinkClient.Install(ueNodes.Get(0));
//onoff application will send the first packet at :
//(2.9 (App Start Time) + (1600 (Pkt size in bits) / 16000 (Data rate)) = 3.0 sec
clientApps.Start(slBearersActivationTime + Seconds(0.9));
clientApps.Stop(simTime - slBearersActivationTime + Seconds(1.0));

ApplicationContainer serverApps;
PacketSinkHelper sidelinkSink("ns3::UdpSocketFactory", localAddress);
serverApps = sidelinkSink.Install(ueNodes.Get(1));
serverApps.Start(Seconds(2.0));
```

In this example, an "OnOff" application is installed in the UE 1, which sends a 200 byte packet with the constant bit rate of 16 kb/s. On the other hand, the UE 2 is configured with the "PacketSink" application.

The Sidelink radio bearers are activated by calling the `ActivateSidelinkBearer` method of `LteSidelinkHelper` as follows:

```
//Set Sidelink bearers
Ptr<LteSlTft> tft = Create<LteSlTft>(LteSlTft::BIDIRECTIONAL,
                                              groupAddress4, groupL2Address);
proseHelper->ActivateSidelinkBearer(Seconds(2.0), ueDevs, tft);
```

The `ActivateSidelinkBearer` method takes the following three parameters as input,

1. Time to activate the bearer
2. Devices for which the bearer will be activated
3. The Sidelink traffic flow template

The `tft` in this example is an object of `LteSlTft` class created by initializing its following parameters,

1. Direction of the bearer, i.e., "Transmit", "Receive" or Bidirectional
2. An IPv4 multicast address of the group
3. Sidelink layer 2 group address (used as Sidelink layer 2 group id)

- Activating the Sidelink traces:

```
lteHelper->EnableSlPscchMacTraces();
lteHelper->EnableSlPsschMacTraces();

lteHelper->EnableSlTxPhyTraces();
lteHelper->EnableSlRxPhyTraces();
lteHelper->EnableSlPscchRxPhyTraces();
```

The above code will enable all the Sidelink related traces.

Upon the completion of the simulation, the following trace files can be found in the repository's root folder,

- SlCchMacStats.txt
- SlSchMacStats.txt
- SlCchRxPhyStats.txt
- SlRxPhyStats.txt

For more information related to the above files please refer to the *Sidelink simulation output* section.

- UePacketTrace.tr

  The information in this file is obtained by using the traces "TxWithAddresses" and "RxWithAddresses" of `OnOff` and `PacketSink` application, respectively. Following table shows the snippet of the data from this file for the two UEs,

Table 6: UePacketTrace.tr

Time (sec)	tx/rx	NodeID	IMSI	PktSize (bytes)	IP[src]	IP[dst]
3	tx	3	1	200	7.0.0.2:49153	225.0.0.0:8000
3.08893	rx	4	2	200	7.0.0.2:49153	7.0.0.3:8000

The first row shows that the UE 1 with IMSI 1 transmits a multicast packet of 200 bytes and the UE 2 receives the packet transmitted by the UE 1. As per the client's application data rate and ON time, i.e., 16 kb/s and 2 seconds, respectively, a total of 20 packets are sent and received by the transmitting and the receiving UE.

lte-sl-in-covrg-relay

This example simulates an in-coverage UE-to-Network Relay scenario, which uses both direct discovery and direct communication. The topology of the access network is the one depicted in *Sidelink simple in-coverage scenario* in which two UEs are in-coverage of a single eNB. In this scenario, UE 1 has the role of *Relay UE* and UE 2 has the role of *Remote UE*. Additionally, there is a Remote Host in the network with which the Remote UE communicates, using the connection to the eNB at the beginning of the simulation, and using the Relay UE once connected to it.

When the relay service starts in the simulation, both UEs are configured to perform UE-to-Network Relay discovery for their respective roles. Thus, the Remote UE will be able to detect the Relay UE, select it, and hence establish a one-to-one communication connection with it (See section *RRC* for more information about the protocols). Then, all the packets the Remote UE application sends to the Remote Host are transmitted on the Sidelink to the Relay UE, which then forwards them to the network via the eNB, to finally be routed in the network to the Remote Host. Packets from the Remote Host towards the Remote UE follow the reverse path (i.e., Remote Host -> Network -> eNB -> Relay UE -> Remote UE).

The scenario uses MODE 2 (UE_SELECTED) for both direct discovery and direct communication.

The user can configure the simulation time (in seconds) and to output the configured NS logs by using the corresponding command line variables in the simulation script. For example, a user can run the simulation as follows:

```
./ns3 run "lte-sl-in-covrg-relay --simTime=10 --enableNsLogs=false"
```

Configuration of LTE and Sidelink default parameters

- Configuring parameters for PSSCH resource selection:

```
// Fixed MCS and the number of RBs
Config::SetDefault("ns3::LteUeMac::SlGrantMcs", UintegerValue(16));
Config::SetDefault("ns3::LteUeMac::SlGrantSize", UintegerValue(5));

// For selecting subframe indicator bitmap
Config::SetDefault("ns3::LteUeMac::Ktrp", UintegerValue(1));
//use default Trp index of 0
Config::SetDefault("ns3::LteUeMac::UseSetTrp", BooleanValue(false));
```

The scenario uses MODE 2 for the Sidelink communication, where the UEs select the resources autonomously from a pool specified by the eNB through `RrcConnectionReconfiguration` message. The configuration here is similar to the one in the lte-sl-in-covrg-comm-mode2 scenario.

- Configure the frequency and the bandwidth:

```
Config::SetDefault("ns3::LteEnbNetDevice::DlEarfcn", UintegerValue(100));
Config::SetDefault("ns3::LteUeNetDevice::DlEarfcn", UintegerValue(100));
Config::SetDefault("ns3::LteEnbNetDevice::UlEarfcn", UintegerValue(18100));
Config::SetDefault("ns3::LteEnbNetDevice::DlBandwidth", UintegerValue(50));
Config::SetDefault("ns3::LteEnbNetDevice::UlBandwidth", UintegerValue(50));
```

We use the LTE Band 1 for both LTE and Sidelink communication.

- Reduce frequency of Downlink CQI report to allow for sidelink transmissions:

```
Config::SetDefault("ns3::LteUePhy::DownlinkCqiPeriodicity",
                   TimeValue(MilliSeconds(79)));
```

The Downlink CQI report is sent in the Uplink. In this implementation, the Sidelink uses the same spectrum as the Uplink and Uplink transmissions have higher priority than Sidelink transmissions. Thus, we need to space out in time the Downlink CQI report to avoid too frequent Sidelink transmission drops. Setting the attribute value to 79 ms in the scenario script results in a periodicity of 80 ms in simulation time.

- Configure the transmission power of the Radio Access Network nodes:

```
// Set the UEs power in dBm
Config::SetDefault("ns3::LteUePhy::TxPower", DoubleValue(23.0));
// Set the eNBs power in dBm
Config::SetDefault("ns3::LteEnbPhy::TxPower", DoubleValue(30.0));
```

Topology configuration

- Instantiating LTE, EPC and Sidelink helpers and connecting them:

```
Ptr<LteHelper> lteHelper = CreateObject<LteHelper>();
Ptr<PointToPointEpcHelper> epcHelper = CreateObject<PointToPointEpcHelper>();
lteHelper->SetEpcHelper(epcHelper);
Ptr<LteSidelinkHelper> proseHelper = CreateObject<LteSidelinkHelper>();
proseHelper->SetLteHelper(lteHelper);
```

- Configuring the Sidelink UE Controller:

```
Config::SetDefault("ns3::LteSlBasicUeController::ProseHelper",
                                  PointerValue(proseHelper));
```

As described in the *RRC* section, the Sidelink UE controller was created to provide a framework for the implementation of custom UE-to-Network Relay configurations and algorithms. The default controller in the model is the `LteSlBasicUeController`, which uses the instance of `LteSidelinkHelper` in the scenario to perform some of its tasks and must be connected to it using the attribute system as shown above.

- Configuring the pathloss model:

```
lteHelper->SetAttribute("PathlossModel",
                            StringValue("ns3::Hybrid3gppPropagationLossModel"));
```

- Enabling the Sidelink:

```
lteHelper->SetAttribute("UseSidelink", BooleanValue (true));
```

- Creating the Remote Host and the Internet:

```
Ptr<Node> pgw = epcHelper->GetPgwNode();
// Create a single RemoteHost
NodeContainer remoteHostContainer;
remoteHostContainer.Create(1);
Ptr<Node> remoteHost = remoteHostContainer.Get(0);
InternetStackHelper internet;
internet.Install(remoteHostContainer);
// Create the Internet
PointToPointHelper p2ph;
p2ph.SetDeviceAttribute("DataRate", DataRateValue(DataRate("100Gb/s")));
p2ph.SetDeviceAttribute("Mtu", UintegerValue(1500));
p2ph.SetChannelAttribute("Delay", TimeValue(Seconds (0.010)));
NetDeviceContainer internetDevices = p2ph.Install(pgw, remoteHost);
```

- Creating the eNB and UE nodes and setting their position and mobility:

```
NodeContainer enbNode;
enbNode.Create(1);
NodeContainer ueNodes;
ueNodes.Create(2);
Ptr<ListPositionAllocator> positionAllocEnb =
                                    CreateObject<ListPositionAllocator>();
positionAllocEnb->Add(Vector(0.0, 0.0, 30.0));
Ptr<ListPositionAllocator> positionAllocUe1 =
                                    CreateObject<ListPositionAllocator>();
positionAllocUe1->Add(Vector(10.0, 0.0, 1.5));
Ptr<ListPositionAllocator> positionAllocUe2 =
                                    CreateObject<ListPositionAllocator>();
positionAllocUe2->Add(Vector(-10.0, 0.0, 1.5));
//Install mobility
MobilityHelper mobilityeNodeB;
mobilityeNodeB.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityeNodeB.SetPositionAllocator(positionAllocEnb);
mobilityeNodeB.Install(enbNode);
MobilityHelper mobilityUe1;
mobilityUe1.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe1.SetPositionAllocator(positionAllocUe1);
```

(continues on next page)

(continued from previous page)

```
mobilityUe1.Install (ueNodes.Get(0));
MobilityHelper mobilityUe2;
mobilityUe2.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe2.SetPositionAllocator(positionAllocUe2);
mobilityUe2.Install (ueNodes.Get(1));
```

- Installing LTE devices to the nodes:

```
NetDeviceContainer enbDevs = lteHelper->InstallEnbDevice(enbNode);
NetDeviceContainer ueDevs = lteHelper->InstallUeDevice(ueNodes);
```

Sidelink pool configuration

In this example, the eNB will be configured with the Sidelink communication and discovery pools to be used, which are hold by the `LteSlEnbRrc` class as mentioned in *RRC*. The pool configuration starts by setting a flag in `LteSlEnbRrc` as an indication that the Sidelink is enabled and it is configured as follows:

```
Ptr<LteSlEnbRrc> enbSidelinkConfiguration = CreateObject<LteSlEnbRrc>();
enbSidelinkConfiguration->SetSlEnabled(true);
```

- Configure Sidelink communication pool:

```
enbSidelinkConfiguration->SetDefaultPool(
                      proseHelper->GetDefaultSlCommTxResourcesSetupUeSelected());
```

The above code uses the Sidelink helper to obtain a Mode 2 (UE_SELECTED) Sidelink communication pool configured with default values, which is set as the default pool to be used by all the UEs attached to the eNB.

- Enable Sidelink discovery:

```
enbSidelinkConfiguration->SetDiscEnabled(true);
```

- Configure Sidelink discovery pool:

```
enbSidelinkConfiguration->AddDiscPool(
                      proseHelper->GetDefaultSlDiscTxResourcesSetupUeSelected());
```

Similarly to the communication pool, the Sidelink helper is used to obtain a Sidelink discovery pool configured with default values, which is then installed as the discovery pool to be used by all the UEs attached to the eNB.

- Configure UE-to-Network Relay selection parameters:

```
enbSidelinkConfiguration->SetDiscConfigRelay(
                      proseHelper->GetDefaultSib19DiscConfigRelay());
```

The parameters for UE-to-Network Relay (re)selection are broadcasted in the SIB19 to the UEs attached to the eNB. These parameters must be configured in the scenario when using UE-to-Network Relay, as they are required by the `LteUeRrc` for SD-RSRP measurement and Relay Selection. In this case, the Sidelink helper is used to obtain a default configuration for those parameters, which are then installed in the eNB Sidelink configuration.

- Install Sidelink configuration on the eNB:

```
lteHelper->InstallSidelinkConfiguration(enbDevs, enbSidelinkConfiguration);
```

- Configure and install Sidelink preconfiguration on the UEs:

```
Ptr<LteSlUeRrc> ueSidelinkConfiguration = CreateObject<LteSlUeRrc>();
ueSidelinkConfiguration->SetSlEnabled(true);
LteRrcSap::SlPreconfiguration preconfiguration;
ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);
ueSidelinkConfiguration->SetDiscEnabled(true);
uint8_t nTxResources = 3;
ueSidelinkConfiguration->SetDiscTxResources(nTxResources);
ueSidelinkConfiguration->SetDiscInterFreq(enbDevs.Get(0)->GetObject
↪<LteEnbNetDevice>()->GetUlEarfcn());
lteHelper->InstallSidelinkConfiguration(ueDevs, ueSidelinkConfiguration);
```

IP configuration

- Installing the IP stack on the UEs and assigning IP address (**Please note that the UE-to-Network Relay functionality only supports IPv6 configuration**):

```
// Install the IP stack on the UEs and assign network IP addresses
internet.Install(ueNodes);
Ipv6InterfaceContainer ueIpIface;
ueIpIface = epcHelper->AssignUeIpv6Address(NetDeviceContainer(ueDevs));

// Set the default gateway for the UEs
Ipv6StaticRoutingHelper Ipv6RoutingHelper;
for (uint32_t u = 0; u < ueNodes.GetN(); ++u)
{
    Ptr<Node> ueNode = ueNodes.Get(u);
    // Set the default gateway for the UE
    Ptr<Ipv6StaticRouting> ueStaticRouting =
        Ipv6RoutingHelper.GetStaticRouting(ueNode->GetObject<Ipv6>());
    ueStaticRouting->SetDefaultRoute(epcHelper->GetUeDefaultGatewayAddress6(), 1);
}

// Configure IP for the nodes in the Internet (PGW and RemoteHost)
Ipv6AddressHelper ipv6h;
ipv6h.SetBase(Ipv6Address("6001:db80::"), Ipv6Prefix(64));
Ipv6InterfaceContainer internetIpIfaces = ipv6h.Assign(internetDevices);
internetIpIfaces.SetForwarding(0, true);
internetIpIfaces.SetDefaultRouteInAllNodes(0);

// Set route for the Remote Host to join the LTE network nodes
Ipv6StaticRoutingHelper ipv6RoutingHelper;
Ptr<Ipv6StaticRouting> remoteHostStaticRouting =
    ipv6RoutingHelper.GetStaticRouting(remoteHost->GetObject<Ipv6>());
remoteHostStaticRouting
    ->AddNetworkRouteTo("7777:f000::", Ipv6Prefix(60), internetIpIfaces.
↪GetAddress(0, 1), 1, 0);

//Configure UE-to-Network Relay network
proseHelper->SetIpv6BaseForRelayCommunication("7777:f00e::", Ipv6Prefix(60));

//Configure route between PGW and UE-to-Network Relay network
proseHelper->ConfigurePgwToUeToNetworkRelayRoute(pgw);
```

The above configuration is similar to any usual LTE example with EPC and IPv6, except the two last commands, which are used to configure the UE-to-Network Relay subnetwork and to add a route in the PGW for the packets directed to it. In this example, this route will be used for the packets from the Remote Host towards the Remote UE

once connected to the Relay UE.

- Attaching the UEs to the eNB:

```
lteHelper->Attach(ueDevs);
```

Application configuration

- Installing applications:

```
// interface 0 is localhost, 1 is the p2p device
Ipv6Address remoteHostAddr = internetIpIfaces.GetAddress(1, 1);
uint16_t echoPort = 8000;

// Set echo server in the remote host
UdpEchoServerHelper echoServer(echoPort);
ApplicationContainer serverApps = echoServer.Install(remoteHost);
serverApps.Start(Seconds(1.0));
serverApps.Stop(Seconds(10.0));

// Set echo client in the remote UEs
UdpEchoClientHelper echoClient(remoteHostAddr, echoPort);
echoClient.SetAttribute("MaxPackets", UintegerValue(20));
echoClient.SetAttribute("Interval", TimeValue(Seconds(0.5)));
echoClient.SetAttribute("PacketSize", UintegerValue(200));
ApplicationContainer clientApps = echoClient.Install(ueNodes.Get(1));
clientApps.Start(Seconds(2.0));
clientApps.Stop(Seconds(10.0));
```

In this example, an `UdpEchoClient` application is installed on UE 2, i.e., the Remote UE, and an `UdpEchoServer` application is installed on the Remote Host. The Remote UE application will send packets every 0.5 seconds to the Remote Host, whose application then echoes back the packet to the Remote UE upon reception.

Service configuration

- Configure network bearer for UE-to-Network Relay packet routing:

```
//Setup dedicated bearer for the Relay UE
Ptr<EpcTft> tft = Create<EpcTft>();
EpcTft::PacketFilter dlpf;
dlpf.localIpv6Address = proseHelper->GetIpv6NetworkForRelayCommunication();
dlpf.localIpv6Prefix = proseHelper->GetIpv6PrefixForRelayCommunication();
tft->Add(dlpf);
EpsBearer bearer(EpsBearer::NGBR_VIDEO_TCP_DEFAULT);
lteHelper->ActivateDedicatedEpsBearer(ueDevs.Get(0), bearer, tft);
```

In the above code, we configure an EPS bearer for the Relay UE with a packet filter corresponding to the UE-to-Network Relay subnetwork. This will be used to support the routing of packet directed to the Remote UE towards the Relay UE.

- Configure start of UE-to-Network Relay services:

```
uint32_t serviceCode = 33;
Simulator::Schedule(Seconds(2.0), &LteSidelinkHelper::StartRelayService,
                        proseHelper, ueDevs.Get(0), serviceCode,
```

(continues on next page)

(continued from previous page)

```
                          LteSlUeRrc::ModelA, LteSlUeRrc::RelayUE);

Simulator::Schedule (Seconds(4.0), &LteSidelinkHelper::StartRelayService,
                          proseHelper, ueDevs.Get(1), serviceCode,
                          LteSlUeRrc::ModelA, LteSlUeRrc::RemoteUE);
```

The function `LteSidelinkHelper::StartRelayService` is the one used from the scenario to start the UE-to-Network Relay functionality on a given UE. In this example, both UEs are configured with the same service code and to use direct discovery Model A. We observe the role differentiation in the last parameter of the respective function calls. As described before, UE 1 will have the role of Relay UE and UE 2 will have the role of Remote UE. The Relay UE service will start first, at 2 seconds of simulation time. Then, the Relay UE will start to advertise the service code on its UE-to-Network Relay Discovery Announcements. At 4 seconds of simulation time, the Remote UE starts its service and proceeds to monitor the announcements and to measure their SD-RSRP. Eventually (4 discovery periods), the Relay Selection procedure will be executed, the Remote UE will select UE 1 as its Relay UE (as it is the only one in the system) and then it will start the one-to-one connection procedure to connect to it. Please note that even though the Remote UE service starts at 4 s, the Remote UE will start sending its traffic using the Relay UE (instead of the network) only after the one-to-one connection is established.

Upon the completion of the simulation, the following trace files contain the generated output:

- AppPacketTrace.txt

  The information in this file is obtained by using the traces `TxWithAddresses` and `RxWithAddresses` of `UdpEchoClient` and `UdpEchoServer` applications. The following table shows a snippet of the data from this file. Please note that the IPv6 address used by the Remote UE changes in simulation time. An IPv6 address belonging to the UE-to-Network Relay subnet is assigned to it when it connects to the Relay UE. This last is unknown during the scenario configuration when the trace sinks are connected. Thus, for simplicity and legibility, the IPv6 address of the Remote UE is set to Zero when the trace sinks of the UEs are connected, and thus appear as Zero on the traces. This has no impact on the behavior of the scenario whatsoever, and the actual IPv6 address the Remote UE is using can be observed in the traces by looking to the source address of the packets received by the Remote Host.

Table 7: AppPacketTrace.txt

time (s)	tx/rx	Node ID	IMSI	PktSize (bytes)	IP[src]	IP[dst]
3	tx-UE	6	2	200	:::49153	6001:db80::200:ff:fe00:7:8000
3.02193	rx-RH	3	nan	200	7777:f00d::2:49153	6001:db80::200:ff:fe00:7:8000
3.02193	tx-RH	3	nan	200	6001:db80::200:ff:fe00:7:8000	7777:f00d::2:49153
3.035	rx-UE	6	2	200	6001:db80::200:ff:fe00:7:8000	:::49153
...	...	...	...	...	...	...
7	tx-UE	6	2	200	:::49153	6001:db80::200:ff:fe00:7:8000
7.11293	rx-RH	3	nan	200	7777:f00e::4:49153	6001:db80::200:ff:fe00:7:8000
7.11293	tx-RH	3	nan	200	6001:db80::200:ff:fe00:7:8000	7777:f00e::4:49153
7.21093	rx-UE	6	2	200	6001:db80::200:ff:fe00:7:8000	:::49153

  We observe that the packet received by the Remote Host before the relay service starts for the Remote UE, i.e., with t < 4.0 s, comes from (and it is transmitted back to) the address `7777:f00d::2` which belongs to the network. However, the packet received at 7.11593 s comes from (and it is transmitted back to) the address `7777:f00e::4` which belongs to the UE-to-Network Relay network. In both cases, the packet is successfully received back by the Remote UE, which means that the bidirectional communication with the Remote Host is possible before and after connecting to the Relay UE.

- DlPdcpStats.txt
- UlPdcpStats.txt

These two traces are standard to the ns-3 LTE simulations, but in this example they allow us to verify which UE is using the Downlink and the Uplink for communicating with the network. Tables *UlPdcpStats.txt* and *DlPdcpStats.txt* show snippets of the data on the corresponding files. In this example, the Relay UE IMSI is 1 and the Remote UE IMSI is 2. We see that before the relay service is activated for the Remote UE (t < 4.0 s), this last is using the Uplink to send its packet towards the Remote Host and the Downlink to receive the corresponding echoed packet. Once the Remote UE connects to the Relay UE, we see in the traces that it is the Relay UE who is communicating with the network using Uplink and Downlink, i.e., the Relay UE is forwarding the packets received from the Remote UE in the Sidelink to the network using the Uplink and receiving packets in the Downlink and forwarding them to the Remote UE on the Sidelink.

Table 8: UlPdcpStats.txt

start (s)	end (s)	CellId	IMSI	RNTI	LCID	nTxPDUs	TxBytes	...
2.75	3	1	2	2	3	1	250	...
3.25	3.5	1	2	2	3	1	250	...
...	...	...	...	...	...	...	...	...
7	7.25	1	1	1	4	1	250	...
7.5	7.75	1	1	1	4	1	250	...

Table 9: DlPdcpStats.txt

start (s)	end (s)	CellId	IMSI	RNTI	LCID	nTxPDUs	TxBytes	...
3.5	3.75	1	2	2	3	1	250	...
4	4.25	1	2	2	3	1	250	...
...	...	...	...	...	...	...	...	...
7	7.25	1	1	1	4	1	250	...
7.5	7.75	1	1	1	4	1	250	...

In the following we will discuss the detailed examples mentioned above, however, we will mainly discuss about the topology and the Sidelink configuration, since the example scripts already contain the details of the default attributes and also, most of them are already covered by the previous examples.

wns3-2017-synch

The scenario in this example is composed of 1 hexagonal cell site divided into 3 sectors. Each sector has 1 UE, i.e., 3 in total, which are randomly dropped. These UEs are then grouped in a single group by choosing randomly one UE as a transmitter while the other 2 UEs act as receivers.

Topology configuration

In the following, we walk through the example line by line and discuss each important topology configuration. However, some configuration parameters are skipped, which are already elaborated in the previous sections of LTE. Therefore, it is highly recommended for the new users to go through the basic LTE configuration before digging into D2D examples.

- Randomizing the frame and subframe number of the UEs:

```
Config::SetDefault("ns3::LteUePhy::UeRandomInitialSubframeIndication",
                                                    BooleanValue(unsyncSl));
```

As mentioned in the design documentation *Sidelink synchronization*, by default all the UEs are perfectly synchronized, i.e., all the UEs in a simulation upon being initialized pick the same frame and subframe 1 to start with. Therefore, to

simulate synchronization and to make every UE to pick a random frame and subframe number the attribute "UeRandomInitialSubframeIndication" should be set.

- Instantiating `LteSidelinkHelper` and setting `LteHelper`:

```
Ptr<LteSidelinkHelper> proseHelper = CreateObject<LteSidelinkHelper>();
proseHelper->SetLteHelper(lteHelper);
```

- Instantiating `Lte3gppHexGridEnbTopologyHelper` and setting `LteHelper`

```
Ptr<Lte3gppHexGridEnbTopologyHelper> topoHelper =
                            CreateObject<Lte3gppHexGridEnbTopologyHelper>();
topoHelper->SetLteHelper(lteHelper);
```

- Disabling the eNB for out-of-coverage scenario:

```
lteHelper->DisableEnbPhy(true);
```

Note : Call to disable eNB PHY should happen before installing the eNB devices

- Configure the parameters for hexagonal ring topology:

```
topoHelper->SetNumRings(numRings); // 1 Ring
topoHelper->SetInterSiteDistance(isd); // 500 m
topoHelper->SetMinimumDistance(10); // in meter
topoHelper->SetSiteHeight(32); // in meter
```

The code above will set the parameters to build 1 ring, i.e., 1 hexagonal site, which will have 500 meters of inter-site distance between the sites, if the number of rings is more than 1. The minimum distance between an eNB and the UEs is set to 10 m and the eNB is at the height of 32 m above the ground.

- Creating eNB nodes:

```
NodeContainer sectorNodes;
sectorNodes.Create(topoHelper->GetNumNodes());
```

Call to `topoHelper->GetNumNodes()` will return 3, which in turns would create 3 sector nodes, since, it requires 3 eNBs to cover one hexagon.

- Fixing eNB mobility and installing eNB devices:

```
MobilityHelper mobilityeNodeB;
mobilityeNodeB.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityeNodeB.Install(sectorNodes);
NetDeviceContainer enbDevs = topoHelper->
                                  SetPositionAndInstallEnbDevice(sectorNodes);
```

The call `SetPositionAndInstallEnbDevice` will compute the position of each site and the antenna orientation of each eNB. The antenna orientation of each eNB in a hexagon is 30, 150 and 270 degrees, respectively [TS25814] [TR36814]. As mentioned before, even if the eNB PHY is disabled, the hexagonal topology requires to instantiate the eNB nodes to form the sectors of an hexagon. It is also to be noted that the use of antenna models, e.g., `Parabolic3dAntennaModel`, `ParabolicAntennaModel`, or `CosineAntennaModel` is needed to configure antenna orientation. Moreover, the function `SetPositionAndInstallEnbDevice` is also responsible to call `InstallEnbDevice` of `LteHelper` for each eNB in this scenario.

- Creating UE nodes:

```
NodeContainer ueResponders;
ueResponders.Create(ueRespondersPerSector * sectorNodes.GetN());
```

This will create 3 UE nodes in total, 1 for each sector of hexagonal site.

- Enabling Sidelink

```
lteHelper->SetAttribute("UseSidelink", BooleanValue(true));
```

Note : Attribute ‘‘UseSidelink‘‘ must be set before installing the UE devices.

- Fixing UE mobility and installing UE devices:

```
MobilityHelper mobilityResponders;
mobilityResponders.SetMobilityModel("ns3::ConstantPositionMobilityModel");
topoHelper->DropUEsUniformlyPerSector(ueResponders);
```

This will place 1 UE per sector by choosing their position randomly with in the sector, taking into account the configured minimum eNB to UE distance of 10 m. This will also install the UE devices by calling `InstallUeDevice` of `LteHelper`. The installation of the IP stack and IP address assignment is done in a similar fashion as explain in lte-design:ref:*sec-evolved-packet-core*

- Creating groups of UEs:

```
double ulEarfcn = enbDevs.Get(0)->GetObject<LteEnbNetDevice>()->GetUlEarfcn();
double ulBandwidth = enbDevs.Get(0)->GetObject<LteEnbNetDevice>()->
                                                              GetUlBandwidth();

std::vector<NetDeviceContainer> createdgroups;
createdgroups = proseHelper->AssociateForBroadcast(ueTxPower, ulEarfcn,
                                                    ulBandwidth, ueRespondersDevs,
                                                    -112, numGroups,
                                                    LteSidelinkHelper::
                                                                  SLRSRP_PSBCH);
```

The `AssociateForBroadcast` function will basically form the specified number of groups, i.e., 1 in this example, by choosing the transmitting UE randomly from the total number of UEs. After choosing the transmitting UE, the receiving UEs of the group are selected if the Sidelink RSRP, calculated as per the method defined in [TS36214], between the transmitting UE and the receiving UE is higher than -112 dBm. In this example, UE node 7 with IMSI 2 is selected as a transmitter while remaining 2 UEs will act as receivers. It is to be noted that `AssociateForBroadcast` is only responsible for forming the groups and not installing any application.

All the UEs in a group are stored in a `NetDeviceContainer`, such that the first UE in this container is the transmitter of the group. Finally, this `NetDeviceContainer` is stored in a vector (createdgroups), which is used to install appropriate application in UE devices.

Application configuration

- Installing applications and activating Sidelink radio bearers:

```
**Note : Only IPV4 is supported at this stage**

// Client Application :

 uint32_t groupL2Address = 0x00;

 Ipv4AddressGenerator::Init(Ipv4Address("225.0.0.0"), Ipv4Mask("255.0.0.0"));
 Ipv4Address groupRespondersIpv4Address = Ipv4AddressGenerator::
                                          NextAddress(Ipv4Mask("255.0.0.0"));
```

(continues on next page)

(continued from previous page)

```
 std::vector <NetDeviceContainer>::iterator gIt;
 for (gIt = createdgroups.begin(); gIt != createdgroups.end(); gIt++)
  {
   //Create sidelink bearers
   //Use Tx for the group transmitter and Rx for all the receivers
   //split Tx/Rx
   NetDeviceContainer txUe((*gIt).Get(0));
   NetDeviceContainer rxUes = proseHelper->
                                        RemoveNetDevice((*gIt), (*gIt).Get(0));
   Ptr<LteSlTft> tft = Create<LteSlTft> (LteSlTft::TRANSMIT,
                                    groupRespondersIpv4Address, groupL2Address);

   //Sidelink bearer activation
   proseHelper->ActivateSidelinkBearer (Seconds(1.0), txUe, tft);
   tft = Create<LteSlTft>
               (LteSlTft::RECEIVE, groupRespondersIpv4Address, groupL2Address);
   proseHelper->ActivateSidelinkBearer (Seconds(1.0), rxUes, tft);

     .
     .
     .

   UdpEchoClientHelper echoClientHelper
                                (groupRespondersIpv4Address, grpEchoServerPort);
   clientRespondersApps = echoClientHelper.Install((*gIt).Get(0)->GetNode());

     .
     .
     .

   groupL2Address++;
   groupRespondersIpv4Address = Ipv4AddressGenerator::
                                          NextAddress (Ipv4Mask("255.0.0.0"));
  }

// Server Application :

   PacketSinkHelper clientPacketSinkHelper("ns3::UdpSocketFactory",
                            InetSocketAddress (Ipv4Address::GetAny(), echoPort));
   ApplicationContainer clientRespondersSrvApps =
                                    clientPacketSinkHelper.Install(ueResponders);
```

In the previous step we obtained the vector `createdgroups`, containing a group formed by one transmitter UE and 2 receiver UEs. Now, it is easy to anticipate that the client application will be installed in the transmitter UE and the receiver UEs will have server application. In this example, a user can configure any of the two client applications, i.e., `OnOff` and `UdpEchoClientApplication`, where the later one is the default client application.

The Sidelink radio bearers for Tx (for transmitting UE) and Rx (for receiving UEs) are activated by calling the *ActivateSidelinkBearer'* method of `LteSidelinkHelper`.

Finally, a `PacketSink` application is installed as a server application in the receiving UEs.

Sidelink pool configuration

This example simulates an out-of-coverage scenario, therefore, all the UEs will be configured with a pre-configured Sidelink pool. As mentioned in *RRC*, in this scenario the `LteSlUeRrc` class will be responsible for holding this

per-configured pool configuration. The pool configuration starts by setting a flag in `LteSlUeRrc` as an indication that the Sidelink is enabled. It is configured as follows:

```
Ptr<LteSlUeRrc> ueSidelinkConfiguration = CreateObject<LteSlUeRrc>();
ueSidelinkConfiguration->SetSlEnabled(true);
```

For configuring Sidelink pre-configured pool parameters, the IE "SL-Preconfiguration" defined in the standard [TS36331] is converted into a C++ structure and similar to the basic LTE layer 3 messages it can be found in `LteRrcSap` class. This example uses this structure to configure the Sidelink pool parameters for control, data and synchronization in the following manner,

```
LteRrcSap::SlPreconfiguration preconfiguration;

preconfiguration.preconfigGeneral.carrierFreq = 23330;
preconfiguration.preconfigGeneral.slBandwidth = 50;
preconfiguration.preconfigComm.nbPools = 1;

//control
preconfiguration.preconfigComm.pools[0].scCpLen.cplen = LteRrcSap::SlCpLen::NORMAL;
preconfiguration.preconfigComm.pools[0].scPeriod.period = LteRrcSap::
                                                    PeriodAsEnum(slPeriod).period;
preconfiguration.preconfigComm.pools[0].scTfResourceConfig.prbNum = pscchRbs;
preconfiguration.preconfigComm.pools[0].scTfResourceConfig.prbStart = 3;
preconfiguration.preconfigComm.pools[0].scTfResourceConfig.prbEnd = 46;
preconfiguration.preconfigComm.pools[0].scTfResourceConfig.offsetIndicator.offset = 0;
preconfiguration.preconfigComm.pools[0].scTfResourceConfig.subframeBitmap.
                                                          bitmap = pscchTrpNumber;

//data
preconfiguration.preconfigComm.pools[0].dataCpLen.cplen = LteRrcSap::SlCpLen::NORMAL;
preconfiguration.preconfigComm.pools[0].dataHoppingConfig.hoppingParameter = 0;
preconfiguration.preconfigComm.pools[0].dataHoppingConfig.numSubbands = LteRrcSap::
                                                    SlHoppingConfigComm::ns4;
preconfiguration.preconfigComm.pools[0].dataHoppingConfig.rbOffset = 0;

preconfiguration.preconfigComm.pools[0].trptSubset.subset = std::bitset<3>(0x7);
preconfiguration.preconfigComm.pools[0].dataTfResourceConfig.prbNum = 25;
preconfiguration.preconfigComm.pools[0].dataTfResourceConfig.prbStart = 0;
preconfiguration.preconfigComm.pools[0].dataTfResourceConfig.prbEnd = 49;
preconfiguration.preconfigComm.pools[0].dataTfResourceConfig.offsetIndicator.
                                                          offset = pscchLength;
preconfiguration.preconfigComm.pools[0].dataTfResourceConfig.subframeBitmap.bitmap =
                                                    std::bitset<40>(0xFFFFFFFFFF);

preconfiguration.preconfigComm.pools[0].scTxParameters.alpha = LteRrcSap::
                                                          SlTxParameters::al09;
preconfiguration.preconfigComm.pools[0].scTxParameters.p0 = -40;
preconfiguration.preconfigComm.pools[0].dataTxParameters.alpha = LteRrcSap::
                                                          SlTxParameters::al09;
preconfiguration.preconfigComm.pools[0].dataTxParameters.p0 = -40;

//Synchronization
preconfiguration.preconfigSync.syncOffsetIndicator1 = 18;
preconfiguration.preconfigSync.syncOffsetIndicator2 = 29;
preconfiguration.preconfigSync.syncTxThreshOoC = syncTxThreshOoC;
preconfiguration.preconfigSync.syncRefDiffHyst = syncRefDiffHyst;
preconfiguration.preconfigSync.syncRefMinHyst = syncRefMinHyst;
preconfiguration.preconfigSync.filterCoefficient = filterCoefficient;
```

Finally, the configured pool is stored in `LteSlUeRrc` class by calling `SetSlPreconfiguration` function

```
ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);
```

Then, by calling the `InstallSidelinkConfiguration` method of `LteHelper` class it configures the `LteUeRrc` attribute named "SidelinkConfiguration" of a UE, which is nothing but a pointer to the `LteSlUeRrc` object used to configure the pool:

```
lteHelper->InstallSidelinkConfiguratio (ueRespondersDevs, ueSidelinkConfiguration);
```

This interaction of classes to populate the Sidelink pool is also depicted in *ns-3 LTE Sidelink pool configuration flow*.

Alternatively, the pool configuration for control and data can also be done in a semi-automatic way by using the `LteSlPreconfigPoolFactory`. This would ease the configuration of the pool by only changing the parameters of interest and leaving the rest as default. For example, the above configuration can also be done as follows,

```
preconfiguration.preconfigGeneral.carrierFreq = 23330;
preconfiguration.preconfigGeneral.slBandwidth = 50;
preconfiguration.preconfigComm.nbPools = 1;

LteSlPreconfigPoolFactory pfactory;

//Control
pfactory.SetControlPrbStart(3);
pfactory.SetControlPrbEnd(46);

//Data
pfactory.SetDataOffset(pscchLength);

//Synchronization
preconfiguration.preconfigSync.syncOffsetIndicator1 = 18;
preconfiguration.preconfigSync.syncOffsetIndicator2 = 29;
preconfiguration.preconfigSync.syncTxThreshOoC = syncTxThreshOoC;
preconfiguration.preconfigSync.syncRefDiffHyst = syncRefDiffHyst;
preconfiguration.preconfigSync.syncRefMinHyst = syncRefMinHyst;
preconfiguration.preconfigSync.filterCoefficient = filterCoefficient;

preconfiguration.preconfigComm.pools[0] = pfactory.CreatePool();

ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);

lteHelper->InstallSidelinkConfiguration(ueRespondersDevs, ueSidelinkConfiguration);
```

Upon the completion of the simulation, the information related to the topology and the data related to the synchronization process gathered through the traces is written to the following files,

- nPositions.txt

  This file stores the position of all the nodes, i.e., eNBs and the UEs in the simulation.

- prose-connections.txt

  This file stores the information, e.g., Node id, and IMSI of the transmitting and the corresponding receiving UEs.

- FirstScan.txt

  This file stores the start time (in ms) of the first scanning period and the IMSI of each UE nodes. The start time for each UE is randomly chosen between 2000 ms and 4000 ms defined by the variables "firstScanTimeMin" and "firstScanTimeMax" in the simulation script.

- SyncRef.txt

  The information in this file is gathered by listening to the trace "ChangeOfSyncRef" of `LteUeRrc`.

Table 10: SyncRef.txt

Time (ms)	IMSI	prev SLSSID	prev RxOffset	prev FrameNo	prev SframeNo	curr SLSSID	curr RxOffset	curr FrameNo	curr SframeNo
0	1	0	0	0	0	148921	0	0	0
0	2	0	0	0	0	175823	0	0	0
0	3	0	0	0	0	170827	0	0	0
20023	2	175823	0	637	5	20	0	637	5
21407	1	148921	0	224	3	20	2	775	9
21460	3	170827	0	791	7	20	2	781	2
50000	1	20	0	563	5	0	0	0	0
50000	2	20	0	563	5	0	0	0	0
50000	3	20	0	563	5	0	0	0	0

It is to be noted that the data in first and last three rows of above table are written by this simulation script to mark the start and end of the simulation. From row 4 we can observe that the UE with IMSI 1 becomes the SyncRef, this can be deduced by looking at its SLSS-ID, which is `SLSS-ID = IMSI * 10` as explained in *RRC*. On the other hand, UEs with IMSI 2 and 3 selects IMSI 1 as their SyncRef and use the same SLSS-ID, as shown in row 5 and 6.

- pscr-ue-pck.tr

  The information in this file is obtained by using the traces "TxWithAddresses" and "RxWithAddresses" of `UdpEchoClient` and `PacketSink` application respectively. Following table shows the snippet of data from this file for all the three UEs,

Table 11: pscr-ue-pck.tr

Time (ms)	tx/rx	NID	IMSI	UEtype	size (bytes)	IP[src]	IP[dst]
21485	t	7	2	resp	40	7.0.0.3:49153	225.0.0.1:8000
21488	r	6	1	resp	40	7.0.0.3:49153	7.0.0.2:8000
21488	r	8	3	resp	40	7.0.0.3:49153	7.0.0.4:8000

The first row shows that the first UE with node id 4 and IMSI 1 acts as a transmitter and sends a multicast packet of 40 bytes. Similarly, the other two UEs act as receiver.

- TxSlss.txt

  This file stores the information gathered by listening to the trace "SendSLSS" of `LteUeRrc`. Following is the snippet of the information stored in this file.

Table 12: TxSlss.txt

Time (ms)	IMSI	SLSSID	txOffset (ms)	inCoverage	FrameNo	SframeNo
20038	2	20	29	0	639	4
20078	2	20	29	0	643	4
20118	2	20	29	0	647	4
20158	2	20	29	0	651	4

It can be observed that UE with IMSI 1 being a SyncRef is transmitting SLSS with a fixed periodicity of 40 ms. Moreover, at the time when this UE chose to become a SyncRef it randomly selects one of the two configured offsets,

i.e., syncOffsetIndicator1 and syncOffsetIndicator2. In this example, it has selected syncOffsetIndicator2, which is set to 29 ms.

wns3-2017-discovery

This example deploys 10 out-of-coverage UEs, distributed randomly within an area of 100 m x 100 m. This example script also illustrates how an out-of-coverage Sidelink related simulation can be performed without instantiating the eNB nodes.

Topology configuration

Compared to the topology of the synchronization example that we discussed before, this example is very simple. For instance, it does not use hexagonal topology or form groups of UEs, i.e., it does not use `Lte3gppHexGridEnbTopologyHelper` and `LteSidelinkHelper`. The `LteHelper` and `PointToPointEpcHelper` are initialized in the same way like any other LTE example with EPC. Also, the call to disable the eNB PHY can also be skipped, since it does not instantiate the eNB nodes. Moreover, this example only focuses on simulating ProSe out-of-coverage discovery, therefore, there is no need to configure the IP of the UEs since discovery is purely a MAC layer application where the messages are filtered on the basis of ProSeAppCode.

The first Sidelink related configuration in this example is to enable the Sidelink.

- Enabling Sidelink

```
lteHelper->SetAttribute("UseSidelink", BooleanValue(true));
```

Note : Attribute ‘‘UseSidelink‘‘ must be set before installing the UE devices.

- Work-around to bypass the use of eNB nodes

```
(1)

  lteHelper->SetAttribute ("PathlossModel",
                           StringValue("ns3::FriisPropagationLossModel"));

(2)

  lteHelper->Initialize ();

(3)

  double ulFreq = LteSpectrumValueHelper::GetCarrierFrequency(23330);
  NS_LOG_LOGIC("UL freq: " << ulFreq);
  Ptr<Object> uplinkPathlossModel = lteHelper->GetUplinkPathlossModel();
  Ptr<PropagationLossModel> lossModel = uplinkPathlossModel->
                                  GetObject<PropagationLossModel>();
  NS_ABORT_MSG_IF (lossModel == NULL, "No PathLossModel");
  bool ulFreqOk = uplinkPathlossModel->
         SetAttributeFailSafe("Frequency", DoubleValue(ulFreq));

  if (!ulFreqOk)
    {
      NS_LOG_WARN("UL propagation model does not have a Frequency attribute");
    }
```

The use of eNB nodes can be bypassed by using the above commands strictly in the order they are listed. The command “lteHelper->Initialize ()” basically performs the channel model initialization of all the component carriers.

Therefore, it is necessary to configure any desired pathloss model before issuing this command. The commands in step 3 are to properly configure the frequency attribute of the pathloss model used, which is normally done in `InstallSingleEnb` method of `LteHelper`.

- Creating the UE node, fixing their mobility, and installing UE devices:

```
NodeContainer ues;
ues.Create(nbUes);

//Position of the nodes
Ptr<ListPositionAllocator> positionAllocUe =
                        CreateObject<ListPositionAllocator>();

for (uint32_t u = 0; u < ues.GetN(); ++u)
{
    Ptr<UniformRandomVariable> rand = CreateObject<UniformRandomVariable>();
    double x = rand->GetValue(-100,100);
    double y = rand->GetValue(-100,100);
    double z = 1.5;
    positionAllocUe->Add(Vector(x, y, z));
}

// Install mobility

MobilityHelper mobilityUe;
mobilityUe.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityUe.SetPositionAllocator(positionAllocUe);
mobilityUe.Install(ues);

NetDeviceContainer ueDevs = lteHelper->InstallUeDevice(ues);
```

The above commands will simply create 10 UEs and position each UE at a random location in an area of 100 m x 100 m. The UE devices are installed in a usual way by calling `InstallUeDevice` of `LteHelper` class.

Service configuration

- Configuring discovery applications and starting the discovery process:

```
std::map<Ptr<NetDevice>, std::list<uint32_t>> announceApps;
std::map<Ptr<NetDevice>, std::list<uint32_t>> monitorApps;
for (uint32_t i = 1; i <= nbUes; ++i)
{
    announceApps[ueDevs.Get(i - 1)].push_back((uint32_t)i);
    for (uint32_t j = 1; j <= nbUes; ++j)
    {
        if (i != j)
        {
            monitorApps[ueDevs.Get(i - 1)].push_back((uint32_t)j);
        }
    }
}

for (uint32_t i = 0; i < nbUes; ++i)
{
    // true for announce
    Simulator::Schedule(Seconds(2.0),
                        &SlStartDiscovery,
```

(continues on next page)

(continued from previous page)

```
                        lteHelper,
                        ueDevs.Get(i),
                        announceApps.find(ueDevs.Get(i))->second,
                        true);

    // false for monitor
    Simulator::Schedule(Seconds(2.0),
                        &SlStartDiscovery,
                        lteHelper,
                        ueDevs.Get(i),
                        monitorApps.find(ueDevs.Get(i))->second,
                        false);
}
```

Given the broadcast nature of the discovery process, the applications are configured such that each UE will be able to transmit discovery messages with one ProSeAppCode, i.e., one announce app and will be able to monitor/receive the discovery messages from all other UEs (Notice the two `for` loops at the beginning). The UE device index in the `NetDeviceContainer` is used as a ProSeAppCode, which is later in `LteUeRrc` is converted into a 184 bitset [TS23003]. Finally, at 2 sec the method "SlStartDiscovery" implemented in this example calls the `StartDiscovery` of `LteHelper` class to start the discovery process.

Sidelink pool configuration

This example simulates an out-of-coverage scenario, therefore, all the UEs will be configured with a pre-configured Sidelink discovery pool. As mentioned in *RRC*, in this scenario the `LteSlUeRrc` class will be responsible for holding this per-configured pool configuration. The pool configuration starts by setting a flag in `LteSlUeRrc` as an indication that the Sidelink discovery is enabled. It is configured as follows:

```
Ptr<LteSlUeRrc> ueSidelinkConfiguration = CreateObject<LteSlUeRrc> ();
ueSidelinkConfiguration->SetDiscEnabled (true);
```

For configuring Sidelink discovery pre-configured pool parameters, the IE "SL-Preconfiguration" defined in the standard [TS36331] is converted into a C++ structure and similar to the basic LTE layer 3 messages it can be found in `LteRrcSap` class. This example uses this structure to configure the Sidelink discovery pool parameters in the following way,

```
LteRrcSap::SlPreconfiguration preconfiguration;

preconfiguration.preconfigGeneral.carrierFreq = 23330;
preconfiguration.preconfigGeneral.slBandwidth = 50;
preconfiguration.preconfigDisc.nbPools = 1;

preconfiguration.preconfigDisc.pools[0].cpLen.cplen = LteRrcSap::SlCpLen::NORMAL;
preconfiguration.preconfigDisc.pools[0].discPeriod.period = LteRrcSap::
                                                            SlPeriodDisc::rf32;
preconfiguration.preconfigDisc.pools[0].numRetx = 0;
preconfiguration.preconfigDisc.pools[0].numRepetition = 1;
preconfiguration.preconfigDisc.pools[0].tfResourceConfig.prbNum = 10;
preconfiguration.preconfigDisc.pools[0].tfResourceConfig.prbStart = 10;
preconfiguration.preconfigDisc.pools[0].tfResourceConfig.prbEnd = 49;
preconfiguration.preconfigDisc.pools[0].tfResourceConfig.offsetIndicator.offset = 0;
preconfiguration.preconfigDisc.pools[0].tfResourceConfig.subframeBitmap.
                                                    bitmap = std::bitset<40>(0x11111);
preconfiguration.preconfigDisc.pools[0].txParameters.txParametersGeneral.alpha =
```

(continues on next page)

(continued from previous page)

```
                                                    LteRrcSap::SlTxParameters::al09;

preconfiguration.preconfigDisc.pools[0].txParameters.txParametersGeneral.p0 = -40;

preconfiguration.preconfigDisc.pools[0].txParameters.txProbability =
                              SidelinkDiscResourcePool::TxProbabilityFromInt (txProb);

NS_LOG_INFO("Install Sidelink discovery configuration in the UEs...");
ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);
lteHelper->InstallSidelinkConfiguration(ueDevs, ueSidelinkConfiguration);
```

Alternatively, the discovery pool configuration can also be done in a semi-automatic way by using the `LteSlDiscResourcePoolFactory`. This would ease the configuration of the pool by only changing the parameters of interest and leaving the rest as default. For example, the above configuration can also be done as follows,

```
preconfiguration.preconfigGeneral.carrierFreq = 23330;
preconfiguration.preconfigGeneral.slBandwidth = 50;
preconfiguration.preconfigComm.nbPools = 1;

LteSlDiscPreconfigPoolFactory pfactory;
pfactory.SetDiscPrbEnd(49);

preconfiguration.preconfigDisc.pools[0] = pfactory.CreatePool();

ueSidelinkConfiguration->SetSlPreconfiguration(preconfiguration);
lteHelper->InstallSidelinkConfiguration(ueDevs, ueSidelinkConfiguration);
```

Upon the completion of the simulation a user can find the following files containing the information of the simulated discovery process.

- discovery_nodes.txt

  This file stores the position of all the UEs in the simulation.

- SlDchMacStats.txt

  This file logs the discovery message transmissions (announcements) by listening to the "SlPsdchScheduling" trace of `LteUeMac` class.

- SlRxPhyStats.txt

  This file contains the information about the reception of the discovery announcements at the physical layer of a recipient UE. In particular, it is obtained by listening to the "SlPhyReception" trace of `LteSpectrumPhy` class.

By comparing the traces from UE MAC and UE PHY, one can notice that the phy layer receives a particular discovery messages exactly after 4 ms of the processing delay between UE MAC and PHY. The CellId is 0 because of out-of-coverage scenario.

- SlDchRxRrcStats.txt

  This file contains the information about the reception of discovery messages by each UE. The data is obtained by listening to the "DiscoveryMonitoring" trace of `LteUeRrc` class. Following is snippet from this file,

Table 13: SlDchRxRrcStats.txt

Time (ms)	IMSI	CellId	RNTI	DisType	ContentType	DiscModel	Content
2240	2	0	2	1	0	1	1
2240	3	0	3	1	0	1	1
2240	4	0	4	1	0	1	1
2240	5	0	5	1	0	1	1

From the above table it can be observed that the UE with IMSI 2, 3, 4, and 5 have received a discovery message. This is a message of type open discovery (indicated by DiscType) used to announce (indicated by ContentType) a discovery model A (indicated by DiscModel) content. The content in this message is the ProSeAppCode of IMSI 1. In other words, these UEs have discovered a UE with ProSeAppCode 1. For more information about the discovery message parameters please have a look at [TS24334].

2.4 Testing Documentation

2.5 LTE Sidelink tests

2.5.1 Unit Tests

Sidelink Communication pool

The test suite `Sidelink-comm-pool` checks the validity of Sidelink communication parameters computed during a simulation, e.g., the expected starting frame and subframe number, index of the starting Resource Block (RB), and the number of RBs in a subframe. The test suite contains a pre-configured communication pool for which these parameters are computed. Table *Sidelink communication pool configuration* below lists the configuration of the pool used.

Table 14: Sidelink communication pool configuration

Parameter	Value
Control period	sf120 (120 ms)
Control bitmap	0x0000000003 (40 bits)
Number of control RBs	1
First control RB index	10
End control RB index	40
Control Offset	80 ms
Data Bitmap	0xCCCCCCCCCC (40 bits)
Number of data RBs	1
First data RB index	0
End data RB index	49
Data Offset	8 ms

There are four test cases:

1. SidelinkCommPoolNextFrameSubframeTestCase

This test checks the starting frame and subframe number of the next Sidelink Control (SC) period given the current frame and subframe number. Let Fr be the current frame number and Sf the current subframe number. The absolute

subframe number Sf_{abs}, since the $Sf = 0$ of $Fr = 0$ is given by :

$$Sf_{\text{abs}} = 10 * (Fr \bmod 1024) + Sf \bmod 10;$$

The number of current Sidelink control period $ScPr_{\text{cur}}$ is calculated as,

$$ScPr_{\text{cur}} = \left\lfloor \frac{Sf_{\text{abs}} - Sc_{\text{offset}}}{ScPr} \right\rfloor$$

where Sc_{offset} is the Sidelink control offset and $ScPr$ is the Sidelink control period from Table *Sidelink communication pool configuration*. The range of $ScPr_{\text{cur}}$ is : $0 \leq ScPr_{\text{cur}} < \frac{10240}{ScPr}$

If the frame number is the last period, the next SC period will start after Sc_{offset}

Finally, the next start frame and subframe number is calculated as,

$$Next_{\text{start}} = Sc_{\text{offset}} + ScPr_{\text{cur}} * ScPr$$
$$Fr_{\text{next}} = \frac{Next_{\text{start}}}{10} \bmod 1024$$
$$Sf_{\text{next}} = Next_{\text{start}} \bmod 10$$

Three different configuration has been tested:

1. $Fr = 0$ and $Sf = 5$

For this configuration, the expected behavior is that the next Sidelink control period will start in frame number 8 and subframe 0 because of the Sc_{offset} of 80 ms.

2. $Fr = 1023$ and $Sf = 0$

This configuration tests the roll over upon reaching the maximum frame number range. Here, the expected behavior is to have a Sc_{offset} of 80 ms once again starting from $Sf = 0$ of $Fr = 0$.

3. $Fr = 1002$ and $Sf = 0$

This configuration reflects the condition when the current frame and subframe number are part of an ongoing Sidelink control period. The expected behavior is that the next period will start in frame 1004 and subframe 0, i.e., right after the remaining frames/subframes of the current Sidelink control period.

2. SidelinkCommPoolResourceOpportunityTestCase

This test takes the PSCCH resource and transmission number as an input and verifies the frame, subframe number, index of the starting RB, and the number of RBs available in a subframe. The total number of resources in a PSCCH pool are calculated as per [TS36213] 14.2.1.1.

$$RES_{\text{PSCCH}} = L_{\text{PSCCH}} * \left\lfloor \frac{RB_{\text{PSCCH}}}{2} \right\rfloor$$

where

$L_{\text{PSCCH}} = 2$ is the total number of subframes available for PSCCH in this pool

$RB_{\text{PSCCH}} = 2$ (i.e., RB 10 and RB 40) is the total number of resource blocks per subframe available for PSCCH in this pool.

Hence, $RES_{\text{PSCCH}} = 2$

These resources in time and frequency are indexed from 0 to $RES_{\text{PSCCH}} - 1$. In this test, it will range from 0 -> 1, therefore, we test both the PSCCH resources to verify the frame/subframe number, index of the starting RB and the total number of RBs in a subframe available for Sidelink Control Information (SCI) transmission. We note that, to

ensure the reliability of the SCI message delivery, each message is transmitted in two identical instances, where each instance occupies one RB. The two instances are transmitted over two different subframes. Therefore, for each PSCCH resource there will be two transmissions. As per [TS36213] 14.2.1.1, for a resource index n_{PSCCH} first transmission occurs in RB $a1$ of subframe $b1$ where

$$a1 = \left\lfloor \frac{n_{\mathrm{PSCCH}}}{L_{\mathrm{PSCCH}}} \right\rfloor$$
$$b1 = n_{\mathrm{PSCCH}} \bmod L_{\mathrm{PSCCH}}$$

and the second transmission occurs in RB $a2$ of subframe $b2$ where

$$a2 = \left\lfloor \frac{n_{\mathrm{PSCCH}}}{L_{\mathrm{PSCCH}}} \right\rfloor + \left\lfloor \frac{RB_{\mathrm{PSCCH}}}{2} \right\rfloor$$
$$b2 = (n_{\mathrm{PSCCH}} + 1 + \frac{n_{\mathrm{PSCCH}}}{L_{\mathrm{PSCCH}}} \bmod (L_{\mathrm{PSCCH}} - 1)) \bmod L_{\mathrm{PSCCH}}$$

$RES_{\mathrm{PSCCH}} = 2$ would mean that we have following four configuration to verify,

1. $n_{\mathrm{PSCCH}} = 0$, Transmission 0:

For $n_{\mathrm{PSCCH}} = 0$, the expected behavior is that the first transmission of PSCCH would be in $RB = 10$ of $Fr = 0$ and $Sf = 0$ and number of RBs for this transmission would be 1.

2. $n_{\mathrm{PSCCH}} = 0$, Transmission 1:

For $n_{\mathrm{PSCCH}} = 0$, the expected behavior is that the second transmission of PSCCH would be in $RB = 40$ of $Fr = 0$ and $Sf = 1$ and number of RBs for this transmission would be 1.

3. $n_{\mathrm{PSCCH}} = 1$, Transmission 0:

For $n_{\mathrm{PSCCH}} = 1$, the expected behavior is that the first transmission of PSCCH would be in $RB = 40$ of $Fr = 0$ and $Sf = 0$ and number of RBs for this transmission would be 1.

4. $n_{\mathrm{PSCCH}} = 1$, Transmission 1:

For $n_{\mathrm{PSCCH}} = 1$, the expected behavior is that the second transmission of PSCCH would be in $RB = 10$ of $Fr = 0$ and $Sf = 1$ and number of RBs for this transmission would be 1.

3. SidelinkCommPoolSubframeOpportunityTestCase

This test verifies index/indices of the RBs available for a given frame - subframe number. The following three configurations have been tested.

1. $Fr = 8$ and $Sf = 0$

This configuration refers to the start of SC period, i.e., after 80 ms of offset from $Fr = 0$ and $Sf = 0$. Thus, we should be on the first subframe of the control channel. The expected RB indices available for this subframe are 10 and 40.

2. $Fr = 8$ and $Sf = 1$

This configuration refers to second subframe of the SC period. The expected RB indices available for this subframe are 10 and 40.

3. $Fr = 8$ and $Sf = 2$

This configuration refers the subframe number, which is not part of SC period as indicated by control bit map. Thus, in this subframe none of the resource block will be allocated for PSCCH transmission.

4. SidelinkCommPoolPsschTestCase

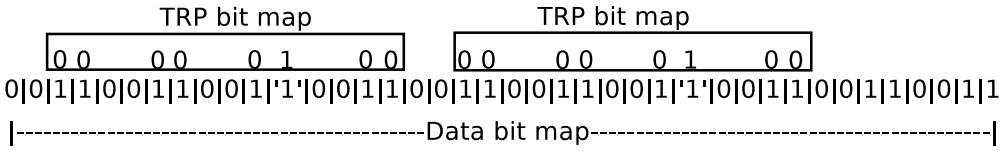
This test checks for the frame-subframe number for PSSCH transmissions. In this test beside the parameters from the pool configuration, an important parameter, i.e., Time Resource Pattern index (iTRP) is provided. All the tests are performed with $iTRP = 5$, which corresponds to the bit map of (0,0,0,0,0,1,0,0) given by Table 14.1.1.1.1-1 of [TS36213]. Every transmission on PSSCH is transmitted using 4 HARQ processes using 4 different subframes. Therefore, this test verifies the frame and subframe number of all the 4 transmissions.

1. $Fr = 0$, $Sf = 5$, $iTRP = 5$, Transmission 0

Starting from $Fr = 0$, the first PSSCH transmission should occur on subframe :

$$Sf_{\mathrm{T0}} = Sc_{\mathrm{offset}} + Data_{\mathrm{offset}} + Sf_{\mathrm{TRP}}$$

where Sc_{offset} and $Data_{\mathrm{offset}}$ values are from the Table *Sidelink communication pool configuration* and Sf_{TRP} are obtained after applying TRP bit map on active bits of data bit map. For example,



If we apply the TRP bit map on the active bits of data bit map, starting from the least significant bit (lbs) of TRP bit map the sixth bit indicates an active subframe. In other words, an offset of 11 subframes as per data bit map, hence, $Sf_{\mathrm{TRP}} = 11$. By using above equation the first PSSCH transmission since subframe 0, would occur on 80 + 8 + 11 = 99 subframe, i.e. $Fr = 9$ and $Sf = 9$.

Similarly, the remaining three transmissions and all other transmissions will occur every 16 ms. This has been tested by the following tests.

2. $Fr = 0$, $Sf = 5$, $iTRP = 5$, Transmission 1

The transmission should occur after 16 ms of transmission 0, i.e., in $Fr = 11$, $Sf = 5$.

3. $Fr = 0$, $Sf = 5$, $iTRP = 5$, Transmission 2

The transmission should occur after 16 ms of transmission 1, i.e., in $Fr = 13$, $Sf = 1$.

4. $Fr = 0$, $Sf = 5$, $iTRP = 5$, Transmission 3

The transmission should occur after 16 ms of transmission 2, i.e., in $Fr = 14$, $Sf = 7$.

Sidelink discovery pool

The test suite `Sidelink-disc-pool` checks the validity of Sidelink discovery parameters computed during a simulation, e.g., the expected starting frame and subframe number, index of the starting Resource Block (RB) and the number of RBs in a subframe. The test suite contains a pre-configured discovery pool for which these parameters are computed. Table *Sidelink discovery pool configuration* below lists the configuration of the pool used.

Table 15: Sidelink discovery pool configuration

Parameter	Value
Discovery cyclic prefix length	NORMAL
Discovery period	rf32 (320 ms)
Number of retransmissions	0
Number of repetition	1
Number of RBs	1
First RB index	10
End RB index	11
Offset	80 ms
Bitmap	0x0000000003 (40 bits)

There are three test cases:

1. SidelinkDiscPoolNextFrameSubframeTestCase

This test checks the starting frame and subframe number of the next Discovery Period (DiscPr) given the current frame and subframe number. Let Fr be the current frame number and Sf the current subframe number. The absolute subframe number Sf_{abs} , since $Sf = 0$ of $Fr = 0$ is given by :

$$Sf_{\text{abs}} = 10 * (Fr \bmod 1024) + Sf \bmod 10;$$

The number of current discovery period $DiscPr_{\text{cur}}$ is calculated as,

$$DiscPr_{\text{cur}} = \left\lfloor \frac{Sf_{\text{abs}} - Disc_{\text{offset}}}{DiscPr} \right\rfloor$$

where $Disc_{\text{offset}}$ is the discovery offset and $DiscPr$ is the discovery period from Table *Sidelink discovery pool configuration*. The range of $DiscPr_{\text{cur}}$ is : $0 \leq DiscPr_{\text{cur}} < \frac{10240}{DiscPr}$

If the frame number is the last period, the next DiscPr will start after $Disc_{\text{offset}}$

Finally the next start frame and subframe number are calculated as,

$$Next_{\text{start}} = Disc_{\text{offset}} + DiscPr_{\text{cur}} * DiscPr$$
$$Fr_{\text{next}} = \frac{Next_{\text{start}}}{10} \bmod 1024$$
$$Sf_{\text{next}} = Next_{\text{start}} \bmod 10$$

Four different configurations have been tested:

1. $Fr = 0$ and $Sf = 5$

For this configuration, the expected behavior is that the next discovery period will start in frame number 8 and subframe 0 because of the $Disc_{\text{offset}}$ of 80 ms.

2. $Fr = 8$ and $Sf = 0$

Here, we are at the start of first discovery period the next discovery period will start after $DiscPr = 320ms$ in frame number 40 and subframe 0.

3. $Fr = 1023$ and $Sf = 0$

This configuration tests the roll over upon reaching the maximum frame number range. Here the expected behavior is to have a $Disc_{\text{offset}}$ of 80 ms once again starting from frame number 0.

4. $Fr = 800$ and $Sf = 0$

This configuration reflects the condition when the current frame and subframe number are part of an ongoing discovery period. The expected behavior is that the next discovery period will start in frame 808 and subframe 0, i.e., right after the remaining frames/subframes of the current discovery period.

2. SidelinkDiscPoolResourceOpportunityTestCase

This test takes as input the PSDCH resource number from the configured pool and checks for the correct frame/subframe number of the starting RB and the total number of RBs in a subframe assigned for Sidelink discovery. The computation of these parameters are done as per 14.3.1 and 14.3.3 of [TS36213]. Specifically, the methods `GetPsdchTransmissions` and `ComputeNumberOfPsdchResources` in `lte-sl-pool.cc`, respectively implements the mentioned clauses of the standard.

The total number of resources for PSDCH are calculated as,

$$RES_{\mathrm{PSDCH}} = \left\lfloor \frac{L_{\mathrm{PSDCH}}}{Retx+1} \right\rfloor * \left\lfloor \frac{RB_{\mathrm{PSDCH}}}{2} \right\rfloor$$

where

$L_{\mathrm{PSDCH}} = 2$ is the total number of subframes available for PSDCH in this pool

$Retx = 0$ is the total number of retransmission allowed

$RB_{\mathrm{PSDCH}} = 2$ (i.e., RB 10 and RB 11) is the total number of resource blocks per subframe available for PSDCH in this pool.

These resources in time and frequency are indexed from 0 to $RES_{\mathrm{PSDCH}} - 1$. In this test, it will range from 0 -> 1, therefore, we test both the PSDCH resources to verify the frame/subframe number, index of the starting RB and the total number of RBs in a subframe available for Sidelink discovery.

1. PsdchResourceNo:0

By selecting the PsdchResourceNo = 0 the expected frame and subframe would be 0 and since, the first RB index of the configured pool is 10 the index of the starting RB is expected to be 10 and, at last, the total number of resource blocks are expected to be 2.

2. PsdchResourceNo:1

Compared to the PsdchResourceNo:0 case, by choosing resource number 1 we expect the discovery transmission to occur on subframe = 1.

3. SidlelinkDiscPoolRbsPerSubframeTestCase

This test checks for the total number of resource blocks available in a single subframe and also verifies the indices of the resource blocks. The following three configurations have been tested.

1. $Fr = 8$ and $Sf = 0$

If we go to 80 ms, we should be on the first subframe of the discovery channel and RB 10 and 11 should be assigned for discovery transmission.

2. $Fr = 8$ and $Sf = 1$

If we go to 81 ms, we should be on the second subframe of the discovery channel and RB 10 and 11 should be assigned for discovery transmission.

2. $Fr = 8$ and $Sf = 2$

If we go to 82 ms, we should be outside the discovery channel and there should be no resource block assignment during this subframe.

System Tests

In-coverage Sidelink communication

The test suite `sidelink-in-coverage-comm` is a system test to test in-coverage Sidelink communication. In particular, it contains two cases to test Mode 1 and Mode 2 for Sidelink communication, respectively. In Mode 1, D2D communications are assisted by the eNodeB, i.e., resource scheduling is performed dynamically by the eNodeB. In Mode 2, UEs manage resource scheduling autonomously relying on pre-configured settings, and both, PSCCH and PSSCH, resources are selected at random from their respective resource pools.

This test creates a scenario with two UEs, which are in coverage of one eNodeB. One of the UE will send traffic directly to the other UE via Sidelink as per configured mode, i.e., Mode 1 or Mode 2. Default configuration will send 10 packets per second for 2 seconds. The expected output is that the receiver UE node would receive 20 packets.

Out of coverage Sidelink communication

Similar to the in-coverage test, the test suite `sidelink-out-of-coverage-comm` is a system test to test the out of coverage Sidelink communication. This test creates a scenario with two out of coverage UEs, where one of the UE will send traffic directly to the other UE via Sidelink. Default configuration will send 10 packets per second for 2 seconds. The expected output is that the receiver UE node would receive 20 packets.

Sidelink synchronization

The test suite `sidelink-synch` is a system test to test the synchronization of the UEs in a test scenario with a selected SyncRef UE. In particular, it simulates the following two topologies by varying the traffic type, traffic direction in the sidelink TFT, and the random stream.

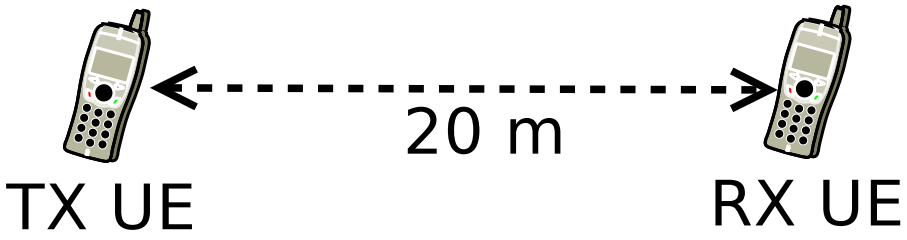


Fig. 26: Sidelink synchronization test topology 1

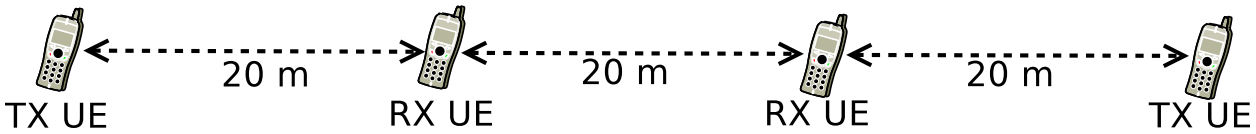


Fig. 27: Sidelink synchronization test topology 2

There are four test cases, two for each topology. Each of the four tests uses ten different random number streams. The reason behind changing the random stream value is to make sure that at the time of applying the change of timings, the newly computed frame and subframe number is aligned with the selected SyncRef UE since the computation of an absolute subframe number is sensitive to the combination of the frame and subframe number used. Currently, all the tests are performed using an OnOff application configured to generate CBR traffic. The test for the alignment of the frame and subframe number is performed after 0.3 ms the trace `ChangeOfSyncRef` gets trigger from the RRC layer of the synchronizing UE. This delay of 0.3 ms is needed because if the SynchRef UE is instantiated after the

synchronizing UE in the simulation script, the event triggering the subframe indication method, which is responsible to update the frame and subframe number at the physical layer of the SyncRef will happen after this UE.

Let's take an example of node 5 (RX UE) and node 6 (TX UE/SyncRef) in topology 2. In the following logs, node 5 is synchronizing with node 6 by changing its frame to 982 and subframe to 7. At this stage, the frame and subframe number at the SyncRef UE (i.e., Node 6) would be 982 and 6, respectively. This is because the `SubframeIndication` method with frame 982 and subframe 7 for node 6 is later in the simulator event list. Therefore, we chose to delay the CheckSfnAlignment method in the test.

```
+6.440000000s 5 LteUePhy:SubframeIndication(0xf6e430, 973, 10)

+6.440000000s 5 LteUePhy:ChangeOfTiming(0xf6e430, 973, 10)

+6.440000000s 5 LteUePhy:ChangeOfTiming(): The UE is not currently transmitting
                Sidelink communication... Applying the change of timing

+6.440000000s 5 LteUePhy:ChangeOfTiming(): mibTime frame/subframe = 982/3

+6.440000000s 5 LteUePhy:ChangeOfTiming(): currentTime frame/subframe = 973/10

+6.440000000s 5 LteUePhy:ChangeOfTiming(): rxSubframe frame/subframe = 973/6

+6.440000000s 5 LteUePhy:ChangeOfTiming(): UE RNTI 3 does not have a Tx pool and
                changed the Subframe Indication from 973/10 to 982/7

+6.440000000s 5 LteUeRrc:DoReportChangeOfSyncRef(0xf70080)

+6.440000000s 5 LteUeRrc:DoReportChangeOfSyncRef(): 0xf70080 UE IMSI 3
                reported successful change of SyncRef, selected SyncRef SLSSID
                40 offset 36

+6.440000000s 5 LteUePhy:SubframeIndication(): (re)synchronization successfully
                performed
 .
 .
+6.440000000s 6 LteUePhy:SubframeIndication(0xf73e60, 982, 7)

+6.440299999s Time of CheckSfnAlignment in the test
```

Finally, the test would pass if the synchronizing UE has a similar frame and subframe number as its SyncRef UE.

In-coverage Relay communication

The test suite `sl-in-covrg-1relay-1remote-regular` is a system test to test the in-coverage sidelink communication scenario between a Relay UE and a Remote UE. The test uses as parameters the simulation time, number of packets for the echo client application, the interval between packets for the echo client application, and the packet size for the echo client application. The echo client is installed on the Remote UE and the echo server is installed on a remote host machine on the network. After the successful setup of the one-to-one communication state machine, the packets are expected to flow from the Remote UE to the Relay UE and packet gateway and ultimately end up at the remote host, which echoes the packets back on the reverse route. The test is considered to have passed if both the UEs reach the right state by transmitting the 'Remote UE Info Request' and 'Remote UE Info Response' messages between each other and if at least one packet has been echoed between the Remote UE and the remote host.

The test suite `sl-in-covrg-1relay-1remote-keepalive` modifies the timers associated with the one-to-one communication state machine to force the Relay UE and the Remote UE to transmit 'Direct Communication Keepalive' and 'Direct Communication Keepalive Acknowledgement' messages between each other. The test is considered to have passed if both the UEs keep the link alive by receiving the respective keepalive messages.

The test suites `sl-in-covrg-1relay-1remote-disconnect-remote` and `sl-in-covrg-1relay-1remote-disconnect-relay` forces the Remote UE and the Relay UE respectively to relinquish the established one-to-one communication link. The tests are considered to have passed if the appropriate UE sends the 'Direct Communication Release' message and the other UE responds with the 'Direct Communication Release Accept' message.

The test suites `sl-in-covrg-1relay-1remote-reconnect-remote` and `sl-in-covrg-1relay-1remote-reconnect-relay` forces the Remote UE and the Relay UE respectively to relinquish the established one-to-one communication link and reestablish it again later in the simulation. Since the UEs are in coverage of the network, once the sidelink path for the packets between the UEs is removed, the Remote UE uses the uplink path to send the packets to the remote host machine. The tests are considered to have passed if the packets inspected at the remote host have appropriate source IP addresses at specific times in the scenario.

The test suite `sl-in-covrg-1relay-2remote-regular` is a system test to test the in-coverage sidelink communication scenario between a Relay UE and two Remote UEs. Each Remote UE sends certain number of packets to the remote host and receives the echoes of the packets from the remote host. The test is considered to have passed if the collective number of packets sent from the Remote UEs is equal to the collective number of packets received at the Remote UEs.

The test suite `sl-in-covrg-2relay-1remote-regular` is a system test to test the in-coverage sidelink communication scenario between two Relay UEs and a Remote UE. The remote host can receive packets originating from the Remote UE through either Relay UEs. To ensure the packets have taken both paths through each Relay UE, the source address of the packets received at the remote host are inspected for two different IP addresses. The test is considered to have passed if the number of packets received at the remote host is equal to the number of packets sent from the remote host and the received packets at the remote host are from two different source IP addresses.

The test suite `sl-in-covrg-2relay-1remote-disconnect-relay` builds on the `sl-in-covrg-2relay-1remote-regular` by disconnecting the Relay UEs one by one. Once the first Relay UE is disconnected, the Remote UE is forced to use the second Relay UE. Once the second Relay UE is also disconnected, the Remote UE is forced to use its uplink. The test is considered to have passed if the received packets at the remote host are from three different source IP addresses, namely Remote UE uplink IP address, Remote UE IP address under first Relay UE and Remote UE IP address under second Relay UE.

Out of coverage Relay communication

The test suite `sl-ooc-1relay-1remote-regular` adds to the `sl-in-covrg-1relay-1remote-regular` test suite by having the necessary preconfiguration in the Remote UE to help the UEs operate out of network coverage. The out-of-coverage and in-coverage scenarios are similar in input parameters and the passing conditions for the individual test cases. The test suite `sl-ooc-1relay-1remote-misaligned` is a special case of the test suite `sl-ooc-1relay-1remote-regular` where the preconfiguration parameters can be modified for the Remote UE. The comments provided in the test suite tabulate the modifications that can be made to the variables and the responses that can be expected from running the test suite.

The test suites `sl-ooc-relay-generic-traffic-rm2rh` and `sl-ooc-relay-generic-traffic-rm2rm` are system tests to verify the UE-to-Network Relay functionality in scenarios with multiple Relay UEs and with multiple out-of-coverage Remote UEs that are not attached to the network. Both tests suites share the same topology comprising 'nRelayUes' Relay UEs that are deployed around the eNB uniformly on a circle of a given radius. Each Relay UE has a cluster of 'nRemoteUesPerRelay' Remote UEs deployed around itself uniformly on a circle of another given radius. The cluster of Remote UEs around a given Relay UE are interested only in its Relay Service Code and thus will connect only to that Relay UE. The UEs start their relay service sequentially in time. First the Relay UE, then the cluster of Remote UEs associated to that Relay UE (sequentially as well), then the next Relay UE, and so on.

The difference between both test suites is the direction of the traffic. In `sl-ooc-relay-generic-traffic-rm2rh`, each Remote UE sends traffic to a Remote Host in the network, which echoes back the message to the Remote UE. In `sl-ooc-relay-generic-traffic-rm2rm`, the

first Remote UE connected to the first Relay UE is the echoServer. Each of the other Remote UEs send traffic to that Remote UE, which echoes back the message to the corresponding initiating Remote UE. In both test suites, each initiating Remote UE starts sending traffic 1.00 s after the start of the one-to-one connection procedure with its Relay UE and remain active during 10.0 s. The simulation time is calculated so that the last Remote UE can have its 10.0s of traffic activity. Each test suite contains several test cases with different number of Relay UEs (nRelayUes) and Remote UEs per Relay UE (nRemoteUesPerRelay). Each test case passes if at least 50% of the initiating Remote UEs in the test received back at least 50% of their transmitted packets. Given that the Remote UEs are out-of-coverage, the only way that they can reach the peer node (Remote Host or first Remote UE depending on the suite) and receive back the echoed message, is by successfully connecting to the Relay UE. Thus, these two test suites can be used to verify both UE-to-Network Relay one-to-one link setup and UE-to-Network Relay communication.

In-coverage Relay discovery

The test suite `sl-in-covrg-discovery-1relay-1remote` modifies `sl-in-covrg-1relay-1remote-regular` by making one UE the Relay UE and the other UE the Remote UE. The passing condition for the single test case in `sl-in-covrg-discovery-1relay-1remote` is that in both Model A and Model B discovery modes, the Remote UE can successfully discover the relay service code broadcasted by the Relay UE.

CHAPTER

THREE

BUILDINGS MODULE

This chapter is an excerpt of the Buildings module documentation chapter, containing the 3GPP-related models that have been added to the *ns-3* buildings module.

3.1 Design documentation

3.1.1 3GPP aligned propagation loss models

In the following we describe all the propagation loss models, which are compliant with 3GPP standards. In particular, following propagation loss models are implemented:

- Hybrid3gppPropagationLossModel
- IndoorToIndoorPropagationLossModel
- OutdoorToIndoorPropagationLossModel
- OutdoorToOutdoorPropagationLossModel
- ScmUrbanMacroCellPropagationLossModel
- UrbanMacroCellPropagationLossModel

Hybrid3gppPropagationLossModel

The `Hybrid3gppPropagationLossModel` pathloss model is a combination of the following pathloss models:

- IndoorToIndoorPropagationLossModel
- OutdoorToIndoorPropagationLossModel
- OutdoorToOutdoorPropagationLossModel
- UrbanMacroCellPropagationLossModel

This wrapper class is created to make it easier to evaluate the pathloss in different environments, e.g., Macro cell, D2D outdoor, indoor, hybrid (i.e. outdoor to indoor) and with buildings. The following pseudo-code illustrates how the different pathloss models are integrated in `Hybrid3gppPropagationLossModel`:

```
if (Macro Cell Communication)
   then
     L = UrbanMacroCell
else (D2D communication)
   if (NodeA is outdoor)
     then
```

(continues on next page)

(continued from previous page)

```
    if (NodeB is outdoor)
       then
         L = OutdoorToOutdoor
       else
         L = OutdoorToIndoor
  else (NodeA is indoor)
    if (NodeB is indoor)
       then
         L = IndoorToIndoor
       else
         L = OutdoorToIndoor
```

IndoorToIndoorPropagationLossModel

The model is defined by 3GPP for D2D indoor to indoor scenarios [TR36843] [TR36814]. It considers LOS and NLOS scenarios for 700 MHz frequency (Public Safety use cases) by taking into account the shadowing according to a log-normal distribution. For the case when UE is inside the same building as hotspot the standard deviation is 3 dB and 4 dB for LOS and NLOS, respectively. For the scenario when UE is in a different building the standard deviation is 10 dB.

UE is inside a different building as the indoor hotzone

$$L_{\mathrm{NLOS}}[\mathrm{dB}] = max(131.1 + 42.8\log_{10}(R), 147.4 + 43.3\log_{10}(R))$$

UE is inside the same building as the indoor hotzone

$$L_{\mathrm{LOS}}[\mathrm{dB}] = 89.5 + 16.9\log_{10}(R)$$
$$L_{\mathrm{NLOS}}[\mathrm{dB}] = 147.4 + 43.3\log_{10}(R)$$

where the probability of LOS is given as:

$$Prob(R) = \begin{cases} 1 & \text{if } R \leq 0.018Km \\ e^{\frac{-(R-0.018)}{0.027}} & \text{if } 0.018Km \leq R \leq 0.037Km \\ 0.5 & \text{if } R \geq 0.037Km \end{cases}$$

According to the standard [TR36843], the pathloss for 700 MHz band is computed by applying $20\log_{10}(f_c)$ to the pathloss at 2 GHz as follows,

$$LOSS[\mathrm{dB}] = \quad LOSS + 20\log_{10}(\tfrac{f_c}{2}) \quad \text{if } 0.758GHz \leq f_c \leq 0.798GHz \ .$$

where

f_c : frequency [GHz]

R : distance between the hotspot and UE [Km]

OutdoorToIndoorPropagationLossModel

This model is implemented for outdoor to indoor scenarios as per the specifications in [TR36843]. The model supports both Line-of-Sight (LOS) and Non Line-of-Sight (NLOS) scenarios by taking in to account the shadowing according to a log-normal distribution with standard deviation of 7 dB for both the scenarios.

The pathloss equations used by this model is:

$$L_{\mathrm{LOS}}[\mathrm{dB}] = PL\_B1\_tot(d_{in} + d_{out}) + 20 + 0.5d_{in}$$

$$L_{\mathrm{NLOS}}[\mathrm{dB}] = PL\_B1\_tot(d_{in} + d_{out}) + 20 + 0.5d_{in} - 0.8h_{ms}$$

PL_B1_tot is computed as follows,

$$PL\_B1\_tot(d_{in} + d_{out}) = max(PL_{freespace}(d), PL\_B1(d_{in} + d_{out}))$$

where $PL_{freespace}$ is free space path loss from Eq. 4.24 in [winner].

$$PL_{freespace} = 20\log_{10}(d) + 46.4 + 20\log_{10}(\frac{f_{\mathrm{c}}}{5})$$

and PL_B1 is the path loss from Winner + B1 channel model for LOS and NLOS scenarios in hexagonal layout [winnerfinal]:

For LOS

$$PL\_B1_{\mathrm{LOS}}[\mathrm{dB}] = \begin{cases} 22.7\log_{10}(d_{in} + d_{out}) + 27 + 20\log_{10}(f_{\mathrm{c}}) + LOS_{offset} & \text{if } 3m \leq d \leq d_{\mathrm{BP}} \\ 40\log_{10}(d_{in} + d_{out}) + 7.56 - 17.3\log_{10}(h_{\mathrm{bs}}^{'}) - & \text{if } d_{\mathrm{BP}} \leq d \leq 5000m \\ 17.3\log_{10}(h_{\mathrm{ms}}^{'}) + 2.7\log_{10}(f_{\mathrm{c}}) + LOS_{offset} & \end{cases}$$

where the LOS_{offset} is 0 dB and the breakpoint distance is given by:

$$d_{\mathrm{BP}} \approx 4h_{\mathrm{bs}}^{'}h_{\mathrm{ms}}^{'}(\frac{f_{\mathrm{c}}[Hz]}{c})$$

the LOS probability is computed as follows:

$$P_{\mathrm{LOS}} = min(\frac{18}{d}, 1)(1 - e^{\frac{-d}{36}}) + e^{\frac{-d}{36}}$$

and the effective antenna height of the eNB and UE is computed as:

$$h_{\mathrm{bs}}^{'} = h_{\mathrm{bs}} - 1$$

$$h_{\mathrm{ms}}^{'} = h_{\mathrm{ms}} - 1$$

For NLOS

The model supports frequency bands of 700 MHz for Public Safety and 2 GHz for general scenarios in NLOS. The pathloss equations used are the following:

for 700 MHz:

$$PL\_B1_{\mathrm{NLOS}}[\mathrm{dB}] = \begin{cases} (44.9 - 6.55\log_{10}(h_{\mathrm{bs}}))\log_{10}(d_{in} + d_{out}) + 5.83\log_{10}(h_{\mathrm{bs}}) + & \text{if } 3m \leq d \leq 2000m \\ 16.33 + 26.16\log_{10}(f_{\mathrm{c}}) + NLOS_{offset} & \end{cases}$$

for 2 GHz:

$$PL\_B1_{\mathrm{NLOS}}[\mathrm{dB}] = \begin{cases} (44.9 - 6.55\log_{10}(h_{\mathrm{bs}}))\log_{10}(d_{in} + d_{out}) + 5.83\log_{10}(h_{\mathrm{bs}}) + & \text{if } 3m \leq d \leq 2000m \\ 14.78 + 34.97\log_{10}(f_{\mathrm{c}}) + NLOS_{offset} \end{cases}$$

where the $NLOS_{offset}$ is 5 dB.

The remaining parameters used in the above equations are:

f_{c} : frequency [GHz]

d : distance between the eNB and UE [m]

d_{in} : distance from the wall to the indoor terminal [m]

d_{out} : distance between the outdoor terminal and the point on the wall that is nearest to the the indoor terminal [m]

h_{bs} : eNB antenna height above the ground [m]

h_{ms} : UE antenna height above the ground [m]

$h_{\mathrm{bs}}^{'}$: effective antenna height of the eNB [m]

$h_{\mathrm{ms}}^{'}$: effective antenna height of the UE [m]

LOS_{offset} : line-of-sight offset

$NLOS_{offset}$: non line-of-sight offset

c : speed of light in vacuum (3x$10^8 m/s$)

OutdoorToOutdoorPropagationLossModel

This propagation loss model is defined by 3GPP for Device to Device (D2D) outdoor to outdoor scenario [TR36843]. The model supports both LOS and NLOS scenarios by taking in to account the shadowing according to a log-normal distribution with standard deviation of 7 dB for both the scenarios.

The pathloss equation used by this model is:

$$PL\_B1\_tot(d) = max(PL_{freespace}(d), PL\_B1(d))$$

where $PL_{freespace}$ is free space path loss from Eq. 4.24 in [winner].

$$PL_{freespace} = 20\log_{10}(d) + 46.4 + 20\log_{10}(\frac{f_{\mathrm{c}}}{5})$$

and PL_B1 is the path loss from Winner + B1 channel model [winnerfinal] for hexagonal layout and is given by:

$$L_{\mathrm{LOS}}[\mathrm{dB}] = \begin{cases} 22.7\log_{10}(d) + 27 + 20\log_{10}(f_{\mathrm{c}}) + LOS_{offset} & \text{if } 3m \leq d \leq d_{\mathrm{BP}} \\ 40\log_{10}(d) + 7.56 - 17.3\log_{10}(h_{\mathrm{bs}}^{'}) - 17.3\log_{10}(h_{\mathrm{ms}}^{'}) + 2.7\log_{10}(f_{\mathrm{c}}) + LOS_{offset} & \text{if } d_{\mathrm{BP}} \leq d \leq 5000m \end{cases}$$

where the breakpoint distance is given by:

$$d_{\mathrm{BP}} \approx 4h_{\mathrm{bs}}^{'}h_{\mathrm{ms}}^{'}(\frac{f_{\mathrm{c}}[Hz]}{c})$$

The implemented model supports two range of frequency bands 700 MHz and 2 GHz in NLOS scenarios. The pathloss equations are the following:

for 700 MHz:

$$L_{\mathrm{NLOS}}[\mathrm{dB}] = \begin{cases} (44.9 - 6.55\log_{10}(h_{\mathrm{bs}}))\log_{10}(d) + 5.83\log_{10}(h_{\mathrm{bs}}) + & \text{if } 3m \leq d \leq 2000m \\ 16.33 + 26.16\log_{10}(f_{\mathrm{c}}) + NLOS_{offset} \end{cases}$$

for 2 GHz:

$$L_{\mathrm{NLOS}}[\mathrm{dB}] = \begin{cases} (44.9 - 6.55\log_{10}(h_{\mathrm{bs}}))\log_{10}(d) + 5.83\log_{10}(h_{\mathrm{bs}}) + & \text{if } 3m \leq d \leq 2000m \\ 14.78 + 34.97\log_{10}(f_{\mathrm{c}}) + NLOS_{offset} \end{cases}$$

and the probability of LOS is:

$$P_{\mathrm{LOS}} = min(\frac{18}{d}, 1)(1 - e^{\frac{-d}{36}}) + e^{\frac{-d}{36}}$$

According to the standard while calculating Winner + B1 pathloss the following values shall be used

$$h_{\mathrm{bs}} = h_{\mathrm{ms}} = 1.5m$$

$$h_{\mathrm{bs}}^{'} = h_{\mathrm{ms}}^{'} = 0.8m$$

$$LOS_{offset} = 0dB$$

$$NLOS_{offset} = -5dB$$

where

f_{c} : frequency [GHz]

d : distance between the eNB and UE [m]

h_{bs} : eNB antenna height above the ground [m]

h_{ms} : UE antenna height above the ground [m]

$h_{\mathrm{bs}}^{'}$: effective antenna height of the eNB [m]

$h_{\mathrm{ms}}^{'}$: effective antenna height of the UE [m]

LOS_{offset} : line-of-sight offset

$NLOS_{offset}$: non line-of-sight offset

c : speed of light in vacuum (3x$10^8 m/s$)

We note that, the model returns a free space path loss value if the distance between a transmitter and a receiver is less than 3 m.

ScmUrbanMacroCellPropagationLossModel

This propagation loss model is based on the specifications defined for 3GPP Spatial Channel Model (SCM) [TR25996] for NLOS urban macro-cell scenario. The pathloss is based on the modified COST231 Okumura Hata urban propagation model for frequencies ranging from 150 – 2000 MHz. The model also considers shadowing according to a log-normal distribution with standard deviation of 8 dB, as defined in the standard [TR25996].

The pathloss expression used by this model is:

$$L[\mathrm{dB}] = (44.9 - 6.55\log_{10}(h_{\mathrm{bs}}))\log_{10}(\frac{d}{1000}) + 45.5 + (35.46 - 1.1(h_{\mathrm{ms}}))\log_{10}(f_{\mathrm{c}}) - 13.82\log_{10}(h_{\mathrm{bs}}) + 0.7(h_{\mathrm{ms}}) + C$$

where

f_{c} : frequency [MHz]

h_{bs} : eNB antenna height above the ground [m]

h_{ms} : UE antenna height above the ground [m]

d : distance between the eNB and UE [m]

C : Constant factor

The value of $C = 3dB$ for urban macro-cell scenario.

UrbanMacroCellPropagationLossModel

This propagation loss model is developed and documented by 3GPP in [TR36814]. The implemented model covers an urban macro-cell scenario for the frequency range of 2 - 6 GHz with different antennas, building heights and street widths. It is designed for both LOS and NLOS scenarios by taking in to account the shadowing according to a log-normal distribution with standard deviation of 4 dB and 6 dB, for LOS and NLOS, respectively.

The pathloss expressions used by this model are:

$$L_{\mathrm{LOS}}[\mathrm{dB}] = \begin{cases} 22\log_{10}(d) + 28 + 20\log_{10}(f_{\mathrm{c}}) & \text{if } 10m \leq d \leq d_{\mathrm{BP}} \\ 40\log_{10}(d) + 7.8 - 18.0\log_{10}(h_{\mathrm{bs}}^{'}) - 18.0\log_{10}(h_{\mathrm{ms}}^{'}) + 2\log_{10}(f_{\mathrm{c}}) & \text{if } d_{\mathrm{BP}} \leq d \leq 5000m \end{cases}$$

$$L_{\mathrm{NLOS}}[\mathrm{dB}] = \begin{cases} 161.04 - 7.1\log_{10}(W) + 7.5\log_{10}(h) - & \text{if } 10m \leq d \leq 5000m \\ (24.37 - 3.7(\frac{h}{h_{\mathrm{bs}}})^2)\log_{10}(h_{\mathrm{bs}}) + (43.42 - 3.1\log_{10}(h_{\mathrm{bs}}))(\log_{10}(d) - 3) + \\ 20\log_{10}(f_{\mathrm{c}}) - (3.2 - (\log_{10}(11.75h_{\mathrm{ms}}))^2 - 4.97) \end{cases}$$

where the breakpoint distance is given by:

$$d_{\mathrm{BP}} \approx 4h_{\mathrm{bs}}^{'}h_{\mathrm{ms}}^{'}(\frac{f_{\mathrm{c}}[Hz]}{c})$$

The probability of LOS is given by:

$$P_{\mathrm{LOS}} = min(\frac{18}{d}, 1)(1 - e^{\frac{-d}{63}}) + e^{\frac{-d}{63}}$$

and the effective antenna heights of the eNB and UE are computed as:

$$h_{\mathrm{bs}}^{'} = h_{\mathrm{bs}} - 1$$

$$h_{\mathrm{ms}}^{'} = h_{\mathrm{ms}} - 1$$

and the above parameters are

f_{c} : frequency [GHz]

d : distance between the eNB and UE [m]

h : average height of the building [m]

W : street width [m]

h_{bs} : eNB antenna height above the ground [m]

h_{ms} : UE antenna height above the ground [m]

$h_{\mathrm{bs}}^{'}$: effective antenna height of the eNB [m]

$h_{\mathrm{ms}}^{'}$: effective antenna height of the UE [m]

c : speed of light in vacuum (3x10$^8 m/s$)

The model returns 0 dB loss if the distance between a transmitter and a receiver is less than 10 m. Therefore, a user should carefully deploy the UEs, such that, the distance between an eNB and a UE is 10 m or above.

CHAPTER

FOUR

ANTENNA MODULE

This chapter is an excerpt of the Antenna module documentation chapter, containing the parabolic antenna models that have been added to the *ns-3* antenna module.

4.1 Design documentation

4.1.1 Provided models

In this section we describe the PSC-related antenna radiation pattern models that are included within the antenna module.

Parabolic3dAntennaModel

Another 3GPP-defined antenna model is the `Parabolic3dAntennaModel`, drawn from 3GPP TR 36.814 [TR36814]. The model, for 3-sector cell sites with fixed antenna patterns, is defined in Table A.2.1.1-2, 3GPP Case 1 and 3 (Macro-cell). Both a horizontal and vertical antenna pattern is defined, and an equation for combining the two methods is provided. So in contrast to `ParabolicAntennaModel`, different horizontal and vertical configuration parameters are required. The attributes `HorizontalBeamwidth`, `Orientation`, `MaxHorizontalAttenuation`, `VerticalBeamwidth`, and `MaxVerticalAttenuation` are configured with the suggested default values. In addition, attributes for mechanical and electrical tilt are defined; these help to adjust the azimuth angle with respect to the reference system of the antenna.

4.2 Testing Documentation

In this section we describe the PSC-related test suites included with the antenna module that verify its correct functionality.

The unit test suite `parabolic-3d-antenna-model` is based on the `ParabolicAntennaModel` tests. A sequence of test cases at different directions is defined:

1. test horizontal plane with a 60 deg beamwidth; gain is -20dB at +-77.460 degrees from boresight
2. test positive orientation with a 60 deg beamwidth; gain is -10dB at +-54.772 degrees from boresight
3. test negative orientation and different beamwidths with a 80 deg beamwidth; gain is -20dB at +- 73.030 degrees from boresight
4. test vertical plane
5. test tilt

The reference gain is calculated by hand. Each test case passes if the reference gain in dB is equal to the value returned by `Parabolic3dAntennaModel` within a tolerance of 0.001, which accounts for the approximation done for the calculation of the reference values.

CHAPTER

FIVE

SIP MODULE

This document describes an ns-3 model for the Session Initiation Protocol (SIP) [RFC3261]. The source code for the new module lives in the directory `src/sip`.

5.1 Model Description

SIP is a large protocol, and the *ns-3* model only implements a portion, focusing on the most typical aspects from [RFC3261]. The focus is on modeling SIP transactions and dialogs involved in session establishment and release, including the timing and persistence of retransmission of lost messages, and as perceived by end users. Network activities that may occur somewhat transparently to a user (outside of increased message processing delays), such as messaging related to authorization and charging, are not in scope.

SIP is part of the 3GPP IP Multimedia Subsystem (IMS) for cellular, and this model was developed to support Mission-Critical Push-To-Talk (MCPTT) operation, so those aspects have been emphasized. The following usage of SIP in the IMS are described in TS 24.229 [TS24.229].

For the procedures at the UE (Section 5.1):

- **Registration and authentication:** Not implemented.
- **Subscription and notification:** Not implemented.
- **Call initiation:** Supported
- **Session modification:** Not yet supported, but possible for future.
- **Call release:** Supported
- **Emergency service:** Not yet supported, but possible for future.

The *ns-3* model doesn't distinguish between different variants of the CSCF (I-CSCF, P-CSCF, S-CSCF); they are all abstracted as a server 'proxy' function. The procedures supported in network (at the proxy) match those of the UE listed above.

5.1.1 Design

SIP is a text-based protocol, and fields like addresses and URIs are usually defined as strings. For simplicity, we use four-byte unsigned integers to represent URIs and addresses, although the model could be extended or changed to use strings in the future.

The main idea in the *ns-3* SIP design is for a client (user) of this model to delegate the responsibility to manage a SIP dialog, involving multiple SIP transactions, to SIP agents that will call back to the client to inform it of events such as "Call progressing" or "Call ending". Responsibility for setting up communications channels or resolving URIs to IP

addresses is outside the scope of the SIP model. SDP is also outside of the scope; any SDP messages are passed into the SIP model by the client, and the SIP model does not interpret any message payloads.

The *ns-3* SIP model is also not an `ns3::Application`; it is created and used by an *ns3::Application'* instance as a service (although it is possible to reuse it across multiple application instances, in most use cases, there will be only one such user). As a result, SIP is not started or stopped like other applications, nor is it initialized automatically at *ns-3* initialization time (such as other members of an `ns3::Node` are automatically initialized). Instead, the lifecycle of the SIP agents are bound to and managed by the other *ns-3* applications that instantiate and make use of the SIP agent. So, for instance, in the case of MCPTT, each MCPTT user application and MCPTT server application will create its own instance of a `SipAgent` and `SipProxy`, respectively, and will release its pointer to the SIP element upon the application deletion (usually at the end of simulation).

The base class for `SipAgent` and `SipProxy` is the `SipElement` class. Most of the logic is implemented in this base class because there is not much distinction in the *ns-3* model between the agent and proxy roles, in terms of how transactions are managed.

With this background, consider a client that wishes to use the SIP model to establish a session. Consider also that it wishes to use SIP to convey some configuration in the form of notional SDP messages. The client would form an `ns3::Packet` and push an SDP header onto it, and then would hand the packet over to the `SipAgent`. The client would ask the `SipAgent` to send an INVITE request with this packet to a specific URI, and provide to the agent the following: the From and To fields for the SIP header, the packet (possibly with SDP header already appended), the address (IPv4 or IPv6) and port of the proxy, and a callback that can be used by SIP to send the resulting packet down the network stack. The client would also register its own callback to receive notifications of events such as call in progress (100 Trying), call establishment (200 OK), or call termination or failure. A similar process supports the teardown of a call (sending a SIP BYE instead of an INVITE) or session modification (re-INVITE).

The implementation contains classes to hold state for SIP transactions and dialogs, including timers to protect against loss or lack of response. The following timers are used:

- **Timer A:** INVITE request retransmit interval (UDP only); doubles upon retransmission.
- **Timer B:** INVITE transaction timeout timer; report failure upon expiry.
- **Timer C:** proxy INVITE transaction timeout timer; doubles upon retransmission. If provisional 100 was sent, then the server needs to ensure that final response gets sent, or generate CANCEL request if not.
- **Timer E:** non-INVITE request retransmit interval (UDP only); doubles upon retransmission. Similar to Timer A.
- **Timer F:** non-INVITE transaction timeout interval (UDP only). Similar to Timer B.
- **Timer I:** Wait time for ACK retransmits (server absorb ACKs for a time before moving to Terminated). This is part of INVITE server transaction.
- **Timer J:** Wait time for non-INVITE request retransmits (server Completed to Terminated transition, to respond to duplicate requests).
- **Timer K:** Wait time for response retransmits (client absorbs any retransmitted responses for a time before moving from Completed to Terminated). This is part of non-INVITE client transaction.

The following timers from [RFC3261] are not supported:

- **Timer D:** time that client will wait in Completed state, after a 300-699 ACK response has been sent. Since the model does not send these responses, Timer D (and transition from Completed to Terminated) is not needed.
- **Timer G: INVITE response retransmit interval; doubles upon retransmission** Timer G pertains to non-200 response from server
- **Timer H: Wait time for ACK receipt (server waits for ACK to its response)** Timer H pertains to non-200 response from server

5.1.2 Scope and Limitations

Only a few possible requests and responses are handled. The SIP INVITE request, BYE request, and CANCEL request are implemented, along with the '100 Trying' and '200 OK responses'. Failure responses such as '404 Not Found' are not implemented. Unknown requests and responses will cause a simulation error.

[RFC4028] describes a mechanism to send keepalive INVITE and/or UPDATE messages for SIP sessions. This model does not support [RFC4028].

5.1.3 References

If references are not rendered here, they can be found in the bibliography section at the end of the document.

5.2 Usage

5.2.1 Helpers

There are no helper classes for the SIP models.

5.2.2 Attributes

The following attributes exist for class `ns3::SipElement`:

- **ReliableTransport** Whether the transport is reliable (TCP, SCTP) or unreliable (UDP)
- **T1** RTT estimate timer
- **T2** Maximum retransmit interval for non-INVITE requests and INVITE response
- **T4** Maximum duration a message will remain in the network

The following attributes exist for class `ns3::SipProxy`:

- **ProxyInviteTransactionTimeout** Timer C timeout value

No attributes exist for class `ns3::SipAgent`.

5.2.3 Output

ns-3 logging components are defined for SipElement, SipAgent, SipHeader, and SipProxy.

The following trace sources exist for class `ns3::SipElement`:

- **TxTrace** The trace for capturing transmitted messages
- **RxTrace** The trace for capturing received messages
- **DialogState** Trace of Dialog state changes
- **TransactionState** Trace of Transaction state changes",

No trace sources exist for classes `ns3::SipProxy` and `ns3::SipAgent`:

5.2.4 Examples

No standalone examples are presently written, although one could be created from the test suite, initial test case (SIP dialog).

5.3 Validation

This model has been validated by comparing messages with published MCPTT traces from Nemergent and with traces from experiments conducted with Nemergent equipment by NIST Boulder. The tests also serve as regression tests.

The main objective of the tests has been to first confirm the normal operation of the protocol in a MCPTT group call configuration, without any message loss, and then to force various message losses to confirm that different SIP timers are operating as expected to protect against message loss and to notify users about transaction failures.

The test configuration creates three SIP users and one SIP proxy in a group call context; one user initiates a call to the other two users, and later (in some cases) terminates the call. The call causes multiple SIP dialogs and transactions to be established between the entities. In a simulation, the protocol that is making use of SIP for call control (such as an MCPTT application) would register callbacks to learn of event notifications, as well as to receive SIP messages. The test suite registers methods to call as callbacks and to receive messages and checks that the expected events occur at the expected times.

The following tests are written:

1. **SipDialogTest:** Test a full dialog with no message losses (normal operation). This also tests operation with UDP of Timer I (INVITE), J (BYE), K(BYE client).
2. **SipInitiatorInviteLossTest:** Test the recovery from the loss of initial INVITE from Client 1 (towards the Proxy). This tests the operation of Timer A.
3. **SipInitiatorInviteFailureTest:** Test the outcome from the failure of initial INVITE (all retransmissions) from Client 1. This tests the operation of Timer B.
4. **SipInitiatorByeLossTest:** Test the recovery from the loss of initial BYE from Client 1 (towards the Proxy). This tests the operation of Timer E.
5. **SipInitiatorByeFailureTest:** Test the outcome from the failure of BYE (all retransmissions) from Client 1. This tests the operation of Timer F.
6. **SipProxyInviteLossTest:** Test the outcome from the loss of first INVITEs from proxy to Clients 2 and 3, testing the proxy Timer A handling.
7. **SipProxyInviteFailureTest:** Test the outcome from the failure of INVITEs from proxy to Clients 2 and 3. This generates a 408 Request Timeout back to the Client 1.

BIBLIOGRAPHY

[TS22179] 3GPP TS 22.179 "Mission Critical Push To Talk (MCPTT) Mission Critical Push to Talk (MCPTT) over LTE; Stage 1"

[TS24379] 3GPP TS 24.379 "Mission Critical Push To Talk (MCPTT) call control; Protocol specification"

[TS24380] 3GPP TS 24.380 "Mission Critical Push To Talk (MCPTT) media plane control; Protocol specification"

[TS365792] 3GPP TS 36.579-2 "Mission Critical Push To Talk (MCPTT) User Equpment (UE) Protocol conformance specification"

[NIST.IR.8206] Frey, J., Pieper, J., and Thompson, T., "Mission Critical Voice QoE Mouth-to-Ear Latency Measurement Methods", February 2018.

[NIST.IR.8236] Varin, P., Sun, Y., and Garey, W., "Test Scenarios for Mission Critical Push-To-Talk (MCPTT) Off-Network Mode Protocols Implementation", October 2018.

[NIST2016] Rouil, R., Cintrón, F.J., Ben Mosbah, A. and Gamboa, S., "An LTE Device-to-Device module for ns-3", in Proceedings of the Workshop on ns-3, 15-16 June 2016, Seattle (Washington).

[NIST2017] Rouil, R., Cintrón, F.J., Ben Mosbah, A. and Gamboa, S., "Implementation and Validation of an LTE D2D Model for ns-3", in Proceedings of the Workshop on ns-3, 13-14 June 2017, Porto (Portugal).

[NIST2019] Gamboa, S., Thanigaivel, R. and Rouil, R., "System Level Evaluation of UE-to-Network Relays in D2D-enabled LTE Networks", in Proceedings of the 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, 11-13 September 2019, Limassol (Cyprus).

[NIST2019b] Rouil, R., Liu, C., Izquierdo Manzanares, A., "Guidelines for Generating Public Safety Benchmark Scenario Set", NIST Interagency/Internal Report (NISTIR) - 8247, May 2019.

[NIST2021] Garey, W., Henderson, T.R., Sun, Y., Rouil, R. and Gamboa, S., "Modeling MCPTT and User Behavior in ns-3", in Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (Simultech 2021), 7-9 July 2021, Paris (France).

[NIST2021b] Black, E., Gamboa, S. and Rouil, R., "NetSimulyzer: a 3D Network Simulation Analyzer for ns-3", in Proceedings of the Workshop on ns-3, 23-25 June 2021, Virtual Event (USA).

[NIST2021c] Rouil, R., Izquierdo Manzanares, A., Liu, C. and Garey, W., "Modeling Public Safety Communication Scenarios: School Shooting Incident", NIST Technical Note (NIST TN) - 2186, October 2021.

[TS24334] 3GPP TS 24.334 "Proximity-services (ProSe) User Equipment (UE) to ProSe function protocol aspects; Stage 3"

[TS23303] 3GPP TS 23.303 "Technical Specification Group Services and System Aspects; Proximity-based services (ProSe); Stage 2"

[TS23003] 3GPP TS 23.003 "Technical Specification Group Core Network and Terminals; Numbering, addressing and identification; V15"

[TR36814] 3GPP TR 36.814 "E-UTRA Further advancements for E-UTRA physical layer aspects"

[TR36843] 3GPP TR 36.843 "Study on LTE Device to Device Proximity Services; Radio Aspects"

[TS36331] 3GPP TS 36.331 "Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification"

[TS36300] 3GPP TS 36.300 "Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2"

[TS36323] 3GPP TS 36.323 "Evolved Universal Terrestrial Radio Access (E-UTRA); Packet Data Convergence Protocol (PDCP) specification"

[TS36212] 3GPP TS 36.212 "Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding"

[TS36213] 3GPP TS 36.213 "Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures"

[TS36322] 3GPP TS 36.322 "Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Link Control (RLC) protocol specification"

[TS36214] 3GPP TS 36.214 "Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer; Measurements"

[TS25814] 3GPP TS 25.814 "Physical layer aspect for evolved Universal Terrestrial Radio Access (UTRA)"

[NIST2016] Rouil, R., Cintrón, F.J., Ben Mosbah, A. and Gamboa, S., "An LTE Device-to-Device module for ns-3", in Proceedings of the Workshop on ns-3, 15-16 June 2016, Seattle (Washington).

[NIST2017] Rouil, R., Cintrón, F.J., Ben Mosbah, A. and Gamboa, S., "Implementation and Validation of an LTE D2D Model for ns-3", in Proceedings of the Workshop on ns-3, 13-14 June 2017, Porto (Portugal).

[NISTBLERD2D] J. Wang, R. Rouil "BLER Performance Evaluation of LTE Device-to-Device Communications. Technical Report. National Institute of Standards and Technology, Gaithersburg, MD.",.

[NISTFREQHOPP] Cintrón, F.J., "Performance Evaluation of LTE Device-to-Device Out-of-Coverage Communication with Frequency Hopping Resource Scheduling",.

[NIST2019] Gamboa, S., Thanigaivel, R. and Rouil, R., "System Level Evaluation of UE-to-Network Relays in D2D-enabled LTE Networks", in Proceedings of the 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, 11-13 September 2019, Limassol (Cyprus).

[TR25996] 3GPP TR 25.996 v6.1.0 "Spatial channel model for Multiple Input Multiple Output (MIMO) simulations"

[winnerfinal] D5.3 "WINNER+ Final Channel Models"

[winner] D1.1.2 V1.2 "WINNER II Channel Models"

[itur] Draft new Report ITU-R M.[IMT.EVAL] "Guidelines for evaluation of radio interface technologies for IMT-Advanced, Document 5/69-E."

[TR36814] 3GPP TR 36.814, "Further Advancements for E-UTRA Physical Layer Aspects (Release 9)"

[RFC3261] RFC 3261 "SIP: Session Initiation Protocol"

[RFC4028] RFC 4028 "Session Timers in the Session Initiation Protocol (SIP)"

[TS24.229] 3GPP TS 24.229 "IP multimedia call control protocol based on Session Initiation Protocol (SIP) and Session Description Protocol (SDP); Stage 3"