A SOFTWARE TESTBED FOR SIMULATION OF CELLULAR WIRELESS NETWORKS

THESIS

Submitted in Partial Fulfillment

of the Requirements for the

Degree of

MASTER OF SCIENCE (Electrical Engineering)

at the

POLYTECHNIC INSTITUTE OF NEW YORK UNIVERSITY

by

Russell Douglas Ford

January 2012

Approved:

_____
Adviser Signature

_____
Date

_____
Department Head Signature

_____
Date

Copy No. 1

Vita

Russ Ford was born in Sarasota, Florida in May of 1987. He received a dual Bachelor of Science degree in Electrical Engineering and Computer Engineering from Florida State University in 2010. He soon thereafter moved to Brooklyn, New York to join the Master of Science in Electrical Engineering graduate program at the Polytechnic Institute of New York University. In 2011, he was awarded the National Science Foundation IGERT traineeship and now plans to continue his studies as a doctoral candidate in the Computer Science program at NYU Poly. His background and research interests include discrete-event network simulation, cellular network architecture and wireless sensors networks.

This document details a line of investigation and development work that has been ongoing from September, 2010 up until December, 2011. The project is part a larger effort on the part of faculty and student researchers at NYU Poly and the Center for Advanced Technology in Telecommunications to contribute to the advancement of future wireless systems.

This work is dedicated to my parents, Edwin and Kathleen.
Thank you for being there.

Acknowledgements

I would like to thank my advisor, Professor Sundeep Rangan, for his support of my graduate research. I thank the Center for Advanced Technology in Telecommunications at NYU Poly for providing funding for the cluster hardware and for use of the WICAT lab. I also extend my gratitude to the Centre Tecnològic de Telecomunicacions de Catalunya as well as Mr. Guillaume Seguin for the use of their independently-developed source code.

AN ABSTRACT

A SOFTWARE TESTBED FOR SIMULATION OF CELLULAR WIRELESS NETWORKS

by

Russell Douglas Ford

Advisor: Sundeep Rangan, Ph.D.

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science (Electrical Engineering)

January 2012

Cellular wireless technology has been in a perpetual state of accelerated evolution in attempt to keep up with growing demand for mobile data services and the mounting requirements of cellular networks. Researches are continually discovering novel ways to increase throughput and spectral efficiency over the wireless channel, mitigate the effects of radio interference, optimize protocols, streamline network architecture, extend battery life for terminals, enhance data security and privacy and refine every aspect of mobile communications. Researchers often employ real-time *hardware-in-the-loop* (HIL) testbeds to design and validate protocols and algorithms in a simulated cellular deployment environment. These testbeds typically consist of hardware that may be expensive, difficult to configure and limited in terms of the scale and complexity of the simulated environments that can be reproduced in the lab. We propose an entirely software-based alternative to costly and inflexible HIL platforms.

The 3GPP LTE/SAE specifications for the 4G radio interface and mobile network architecture are now widely recognized by the industry and academic community for having the potential to meet the challenges of next-generation cellular networks. By making use of the open-source ns-3 network simulation framework, we implement several protocol models as part of our extended effort to develop a consummate model of a LTE/SAE network. By designing our software to take advantage of parallel computing architectures, we attempt to achieve real-time performance for simulating not only the LTE radio interface but higher-layer protocols in the access and core network as well, thereby offering all of the realism and accuracy of hardware emulators with the cost and configurability of a computer simulation. We finally demonstrate a proof-of-concept experiment involving end-to-end communication between mock hosts and User Equipment with real application data transmitted over a simulated network. Through our efforts, we make strides toward creating a powerful tool that can be put into the

hands of researchers, enabling them to do rapid prototyping of the technology that will bring us closer to the realization of the future mobile Internet.

Contents

List of Figures

List of Tables

Part I
Introduction

Cellular telephony has become an indispensable hallmark of modern life. With mobile device adoption reaching staggering new numbers with each passing year, few technologies can be said to be as successful. The ability to communicate and connect to people, information and content is now always close at hand, and broadband mobile services will soon turn ubiquitous Internet connectivity into a commonplace affair. The role of the archetypal cell phone is now diminished in the presence of smartphones, tablets and other mobile broadband-capable devices that offer a more rich end-user experience than ever before. Although these are bold and perhaps subjective statements, the figures speak for themselves. The Cisco Global Mobile Data Traffic Forecast Update for 2010-2015 estimates that, by 2015, mobile device adoption will have reached one device *per capita* globally [1]. In 2010, smartphone usage doubled and three million tablets and 94 million laptops were connected wirelessly, resulting in a near tripling of global mobile data volume for the third consecutive year. The predictions for the coming years are even more astounding, with a projected 26-fold increase in mobile data traffic between 2010 and 2015. While data-enabled mobile devices are becoming a part of everyday life, the technology behind modern cellular networks is far from commonplace. Cellular systems represent the culmination of some of the most advanced accomplishments in contemporary engineering. The technology has been in a perpetual state of accelerated evolution in attempt to keep up with growing demand and the mounting requirements of cellular networks.

Even as 3G systems such as 3GPP UMTS/WCDMA and now "3.5G" systems like HSPA+ are still in the process of being rolled out, we are beginning to see new standards emerge onto the mobile scene that promise to satisfy the ever-increasing thirst for data services. 3GPP Long Term Evolution (LTE) has come about amid a slew of competing next-generation cellular standards and has set the bar very high in terms of low-latency and high-data rate performance. LTE, along with its counterpart standard known as System Architecture Evolution (SAE), is already being recognized for its potential to meet the challenges of 4G. It is the product of major leaps forward in the science of wireless communications, the labor of thousands of engineers, and intense research efforts on the part of the industry and academic community.

Research institutions seem to have grasped the potential of LTE, as countless related studies are released every year that contribute to improving the standard and the fundamental technology. As such, it is still very much a work in progress, with significant enhancements over the original standard being incorporated into each subsequent release of the LTE specifications. Researches are continually finding novel ways to increase throughput and spectral efficiency over the wireless channel, mitigate the effects of radio interference, optimize protocols, streamline network architecture, extend battery life for terminals, enhance data security and privacy and refine every aspect of cellular communications. Such developments have enabled applications for mobile platforms that provide a level of functionality and access to content and multimedia over the Internet that was previously the exclusive domain of personal computers with wireline broadband connections.

In order to design and prototype new protocols and algorithms and improve upon existing ones, researchers establish wireless *testbed* systems on which perform exper-

iments. These test environments are designed to reproduce the conditions of actual cellular deployments in a lab setting and consist of a combination of software and hardware components such as channel emulators (for recreating the effects of an impaired channel on a wireless signal) and equipment for emulating base stations and mobile terminals. The majority of these testbed implementations are largely hardware-based and, while some vendors offer application-specific test equipment, many research testbeds make use of custom (often DSP and FPGA-based) hardware that can be programmed and tailored to suit specific experiments. Even so, the expense of such hardware and the difficulty of programming and configuring custom electronics places practical limitations on the nature and scale of the simulated environments that can be employed for testing mobile systems.

Experiments involving real-time hardware emulation of components combined with actual software and hardware are referred to as *Hardware-in-the-Loop* or HIL simulations. The emulation hardware reproduces the "black-box" input and output characteristics of the respective system, while the components being tested feed inputs to the black box and respond to output ideally as they would in a real-world setting.

We propose an entirely software-based alternative to potentially costly and inflexible HIL platforms used in the design and validation of next-generation cellular technologies. We investigate a *software-in-the-loop* system by which actual applications and protocols can be integrated with a self-contained software simulation of a virtual LTE/SAE network. By making use of the open-source ns-3 network simulation framework, we implement several protocol models as part of our extended effort to develop a consummate model of a LTE/SAE network. By designing our software to take advantage of parallel computing architectures, we attempt to achieve real-time performance for simulating not only the LTE radio interface but higher-layer protocols in the access and core network as well, thereby offering all of the realism and accuracy of hardware emulators with the cost and configurability of a computer simulation. We finally demonstrate a proof-of-concept experiment involving end-to-end communication between mock hosts and User Equipment with real application data transmitted over a simulated network. Through our efforts, we hope to make strides toward creating a powerful tool that can be put into the hands of researchers enabling them to do rapid prototyping of the technology that will make the vision of the future mobile Internet a reality.

This document is organized as follows. In Section 1, we outline the purposes and motivations behind our investigation and development efforts. In Sections 2 and 3, we lay down the high-level goals for the project followed by an assessment of specific system-level requirements. We then provide some background relevant to our software implementation. Section 4 presents some fundamentals of discrete event simulation. We then go over the basics of LTE/SAE protocols and architecture in Section 6 to aid in understanding the model implementation presented in Section 9. We test our core simulator and network model performance and demonstrate some proof-of-concept simulations in Section 10. Lastly, in Section IV, we conclude this thesis with some remarks on our results and progress in achieving our project goals and discuss the future direction of our work.

1 Motivations

There were many driving factors in seeking a novel software simulation platform. Chief among these are our desire to scale our simulations to a degree well beyond conventional

research testbeds while still producing accurate, real-time behavior. Additionally, we require an enhanced level of flexibility and control in order to manipulate every minute detail of our simulations. We now expand on each of these motivating factors.

1.1 Related Work - Limitations of Hardware Testbeds

As a part of a larger research effort to investigate enhancements to LTE, this project came about after looking into alternative testbed solutions and realizing the shortcomings of available off-the-shelf products and custom development boards. The existing solutions that presented themselves generally fall into two categories, the first of which being a system consisting of multiple proprietary hardware emulator boxes.[1] While a range of these products exist for emulating various network elements in the radio access and core network, they are typically only good for serving the role of one such device. Simulating any sort of practical and interesting cellular deployment scenario like the examples given in the next section would require an array of several of these devices.[2] The cost associated with each product makes this approach prohibitively expensive.[3] Furthermore, the most flexible and extensible of these products still do not permit tuning and configuration of every subsystem and protocol layer for simulated LTE devices such as user equipment. The other direction more often followed by academic research groups is the use of development boards, which provide embedded FGPAs that may be programmed with the LTE stack along with DSP modules for reproducing the low-level PHY and channel characteristics.[4] For most existing research testbeds of this variety, we again see the necessity of investing in several of these devices that must be individually programmed to simulate the role of a single network entity. For the types of experiments involving simulating large-scale networks we envisage, the complexity of configuring and interconnecting a large number of these boards becomes intractable.

1.2 Testing and Prototyping of 4G Cellular Systems

Although production implementations of the LTE/SAE standard are, at present, offered for commercial use and networks are beginning to be deployed by operators globally, research and development activities are by no means stagnant. Long Term Evolution, by its very name, implies upgradability and extensibility in the long-term. The ambitious 3GPP Release 10 system, known as LTE-Advanced, is already planned as the next milestone along the LTE upgrade path that will bring us a true 4G-compliant standard. Such ambitious requirements, as outlined in Section 6, arguably will be crucial for operators and service providers to remain competitive in providing the fast, ubiquitous, on-demand connectivity and access to content services that future subscribers will come to expect. The challenges of future networks are many and must be addressed with innovative solutions from all angles. Here we have just a small selection of examples that are the focus of our research group alone.

- *Diverse, heterogeneous, self-organized networks*: The future mobile landscape is anything but uniform and homogeneous when it comes to interconnected technologies and administrative domains. Already we are seeing diverse multi-operator

[1] Products such as LTE eNodeB and UE emulators are available through vendors like Aiglent [9].

[2] Section 6.8.3 of [55] describes a testbed system employing four base station emulators for simulating a single multi-RAT cell.

[3] Some of the least expensive products we found, such as the NI test suite in [2], are still in the five to ten-thousand USD range.

[4] Some examples of FPGA-based LTE testbeds are given in [11, 12].

environments that support a variety of radio access and core network technologies that must be seamlessly integrated to provide subscribers with constant, uninterrupted service while roaming. It is well understood that central to the ability to handle the data traffic volumes that we are seeing in modern cell networks is the notion of *offloading* of traffic from macrocell to picocell and femntocell base stations and WLAN access points as well. The resulting environment is a dense, ad-hoc assortment of access technologies. Furthermore, the complexity of integrating 4G and legacy 3GPP and non-3GPP technologies within a single operator's network combined with the necessity of internetworking with other domains will require novel methods for Inter-Cell Interference Coordination, Dynamic Spectrum Access, cognitive radio and self-organizing, ad-hoc networking capabilities [3, 4, 5].

- *Mobility*: Mobility has been a chief concern of 3GPP technical specifications groups and, while the current standard incorporates procedures for seamless handover between various RATs, investigations are still ongoing in areas such as vertical handover between LTE and WLAN accesses and application-layer mobility [6, 7].
- *Transport and application-layer protocol optimization*: Many protocols running "over-the-top" of mobile networks, such as TCP and HTTP, were never designed for the traffic patterns and conditions applicable to mobile settings. This lack of cross-layer optimization potentially impairs performance, and so is an area worthy of investigation on its own.
- *Mobile cloud applications*: Cloud-based applications and services are ushering in a new era of computing in the way of functionality and robust end-user experience. Software, storage and computation capacity are moving away from the end-user device and into the network, where they can be provisioned on-demand from anywhere with an Internet connection. The possibilities for business models of mobile cloud and utility computing are already catching the attention of the industry since value-added services are a key source of revenue for operators and service providers. From an engineering perspective, however, the low latency and high data rate requirements of many cloud services make their integration into cellular environments difficult.

1.3 Opportunities for Collaboration and Cooperation

It has always been our intention for our work to be a free, open source contribution to mobile research community at large. As our software implementation is itself based on the open source ns-3 framework, we hope to see our modifications and additions incorporated into the ns-3 project and made available to anyone interested in performing large-scale network simulations. Any source code for the protocols or algorithms implemented on top of our simulation platform are also obtainable, along with documentation, via the Web for individuals and research groups to experiment with and modify, hopefully inviting peer review and encouraging other research efforts to be combined with our own.[5] This level of collaboration would certainly not be possible if we had adopted a platform based on proprietary hardware and software.

[5] See Section 13 for links to our project page and code repository.

2 Goals

One of the greatest challenges in research and development of wireless communications systems is that the standards and technology are in a constant state of flux. Hardware-based testbeds may not offer the opportunity to extend the underlying components to reflect the changing technology. Software, on the other hand, can be modified with minimal cost or effort. While a software-based approach clearly wins out in this regard, for the purpose of testing actual applications and protocols in real time, these advantages cannot come at the cost of inaccurate emulated behavior or results. From these considerations, we arrive at the following fundamental goals for the project.

2.1 Accurate Cellular Wireless Emulation in Software

To provide a true emulation of something, the emulated system must appear to have the exact black-box input and output characteristics of the actual system. This of course means that processes of the emulation must mirror, in real time, the behavior of the real-world physical system. As we examine in Section 4.5, specific measures must be taken to enable real-time performance for software running on a non-real time operating system. In terms of our protocol representation, the effects of the wireless channel shall be simulated based on industry-standard models for statistical radio propagation, interference and error rate calculation. Upper layer protocols in the LTE stack and in the core network can then run ”as-is.” In other words, while our simulated version of said protocols may only provide some essential subset of features found in commercial applications of the same, they shall be correct in the sense of conforming to applicable specifications and RFCs and effectively be no different than the “real thing” in the way of bit-level accuracy of headers and encapsulated user-plane data as well as control-plane messages. Finally, through the use of virtual interfaces, known as *TUN/TAP* devices for Linux systems, we create a means of transporting real application data into and out from the simulation as if an application were running on a networked device. After accounting for each of these considerations, we believe our software can be made to realistically mimic the behavior of real-world cellular networks.

2.2 Flexibility of Simulation Scenarios

We have already emphasized our goals for simulation flexibility, configurability and scalability. For the users of our platform who wish to design specific simulated topologies and scenarios, the tools for building such scenarios come in the form of a modular Application Programming Interface (API). With this common API, users can instantiate network objects such as nodes, devices and links and define object parameters in a simple program or script, which can then be executed from the user’s shell. Our API makes full use of the ns-3 object model for aggregating instances of protocols and devices to network nodes in a very user-friendly fashion. The ability to collect statistics and observe network objects is also an essential feature provided by the ns-3 trace system.

2.3 Support for High-Performance Computing Systems

When simulating large-scale networks with many objects in real-time, computational performance is key. For a system to be considered *real time*, it must be guaranteed to meet certain timing deadlines. So, in the case of *discrete event* simulation, as we shall introduce in Section 4, *events* occurring in the *simulation time* domain must be synchronized with the real time domain to a certain acceptable degree of deviation, beyond which simulation results can no longer be considered accurate. When adding to the algorithmic complexity or size (in the sense of number of objects) of a simulation, at

a certain point the processing resources needed in order to maintain real-time performance will exceed what a single host machine can provide. To address this problem, our software must be designed to exploit parallel programming paradigms to make use of multi-core processors and clustered computing. High-Performance Computing (HPC) clusters can be built cheaply from off-the-shelf components and can be augmented with computational capacity simply by adding additional processors or *compute nodes* to the cluster.[6] It is our intention to develop algorithms that efficiently take advantage of clustered architectures so that simulations can be scaled with the underlying hardware platform.

3 Needs Assessment

From these general goals, we outline the following specific requirements for the system we are undertaking to implement in this and future iterations.

1. *Accurate physical-layer emulation*: The PHY layer protocols and channel model shall be accurate to the extent allowed by the floating point representation and operations of the underlying OS. Industry-standard models for representing interference, fading, shadowing, and propagation loss shall be employed. Data rate, delay and block or bit error rate experienced by air-link signals shall correctly reflect varying channel conditions along with the specific modulation and coding scheme, channel bandwidth and resource allocation scheme.
2. *Bit-level accuracy of user-plane protocols*: Essential functionality of LTE and Evolved Packet Core user-plane protocols and procedures shall conform to 3GPP and IETF specifications. For the sake of efficiency, some control-plane functionality may be more loosely defined, however the overall sequence of control messages or events should obey the standards.
3. *Real-time synchronization of event sequence*: Simulation events shall occur with within a specific range of wall-clock time. This range depends on the nature of the event and affected systems and is predetermined based on perceived effects on simulation accuracy. Events that fail to meet real-time deadlines shall be logged along with any causally-related events for analysis.
4. *Parallel and distributed discrete event simulation*: Algorithms for parallel and distributed DES shall be used that automatically (or with minimum interaction on the part of the user) take into account the nature of the simulation scenario along with the underlying computing platform and operating system capabilities to most efficiently distribute computation across available processing resources.
5. *Application data tunneling through virtual network interfaces*: A TUN/TAP device, which is a virtual link-layer interface, shall be provided by the operating system through which data generated from the IP layer and above (including auxiliary protocols such as ARP) can be tunneled between the OS and simulator program. Multiple instances of Virtual Machines (VMs) with TAP devices can be mapped to wireless User Equipment or wired hosts represented in the simulated network and real application data generated by these VMs can be thereby be transmitted and received across the network.
6. *Modular object model and API*: As discussed, network objects shall be constructed in a modular and extensible way, allowing simulation parameters to be

[6] See Section 8 for more on compute clusters.

defined in a script or at simulation runtime. We achieve this using the ns-3 object model and configuration system. Classes defining new applications protocols should be able to be easily integrated on top of existing protocol and network element objects.

7. *Configurable trace and logging system*: Logging of interesting statistics along with Operations, Administration, and Maintenance (OA&M) data can be enabled for collecting information on any given simulation object. This shall be done in an efficient way that minimizes I/O operations (i.e. disk reads/writes) while balancing memory usage.

Part II
Background

4 Discrete Event Network Simulation

Simulation is an invaluable tool for researchers across many disciplines, from the "hard" sciences and engineering to economics and social sciences. It allows us to analyze the behavior of interesting real-world and hypothetical processes under various conditions by reproducing that behavior in a virtual space [13]. A simulation, in the most general sense, is any system that imitates the behavior of another system according to a finite set of rules that govern the progression of the state of the system being modeled (or more specifically, the states of some composite elements of system) over time [14]. From this abstract definition, we can derive the essential ingredients of a simulation: the mathematical model of the physical system that we wish to represent, the states of the model, and the time over which the states are changing. The model dictates, either deterministically or stochastically, the transition of simulation states based on previous states and given inputs and initial conditions. As many complex physical processes are composed of a large set of variables that give way to an infinite set of possible states, any practical model must identify and isolate specific variables and data that are meaningful to providing an accurate imitation of the system, where the accuracy is proportional to the complexity of the model. While computers are the essential instrument for conducting simulations, they have certain inherent limitations that pose analytical challenges in addition to the underlying difficulty in expressing complex physical systems. In the following sections and throughout this document, we expand on these issues in the direction of understanding discrete-event network simulation on a parallel and distributed microprocessor computers while making a point of identifying limitations and analyzing their effects on providing accurate and useful simulation results.

Discrete-Event Simulation (DES) is a simulation technique that lends to modeling telecommunications networks. DES involves formulating state transitions as a sequence of *events*, which occur at discrete points in time. DES has its roots in discrete event system theory, which is a rigorously-defined mathematical framework that we shall not directly pay any mind to because, when applied to representing communications networks, the concept can be simplified greatly [15]. If one grasps the very basics of packet-switched networking, it is straightforward to see how DES can be applied to simulate such systems. The model is made up of individual network elements, such as nodes, devices (i.e. interfaces), and links (channels), each element having its own set of states. For instance, the states of a node such as a IP router might consist of a packet queue for each port on the router. Network element are defined by some state transition logic that determines its behavior based on its current states and input. Coming back to the router example, this logic may take the form of a routing table for making forwarding decisions. As mentioned, state transitions are driven by events, which are "scheduled" by being inserted into their chronological position in the future event set or queue. Events can be scheduled in order to represent the transmission or reception of a packet by our router. We can break down each of these aspects into three encompassing data structures: the set of state variables, a single event list, and a global simulation clock [14]. Parallel DES, which we shall later introduce, puts some interesting twists on these aspects. For now, we elaborate on these common components

of DES.

4.1 Time

As stated in [14], there are several notions of time to consider when simulating physical systems.

- *Physical time*: This is the actual time of the physical system being modeled. For most real-world systems, the physical time domain is continuous.
- *Wall-clock time*: The real time that elapses during simulation execution, which is independent from the physical time.
- *Simulation time*: The representation of physical time in the simulation. Unlike the former cases, which are the same as the conventional notion of time, *sim time* is an entirely abstract notion. The simulation clock increments not with some regular period but, instead, hops from one event timestamp to the next as it progresses chronologically through the sequence of events. These timestamps are mapped into the continuous, physical time domain with the accuracy of whatever numeric data type, such as a double-precision floating point number, is used. In other words, events can be scheduled to occur at any instant of sim time that can be represented by the respective data type.

4.2 Event Scheduling Algorithms

Events, as we mentioned, are any conceivable occurrence that causes the progression of simulation states. Events can be anything from atomic operations, such as assignments to state variables, to complex subroutines that rely on dynamic input from multiple simulation objects. Global events (e.g. a command to terminate the simulation after a specific sim time) are those that have an effect on the simulation as a whole. More often, events are scheduled that affect the states of one or more local objects. In DES simulation of telecommunications networks, these local events may be scheduled to update the state of a particular network component. For instance, a packet transmission event may occur on a host node, triggering the node to then schedule a reception event on the destination node occurring after some transmission delay. When the simulation advances to the reception event, the event is executed and may result in a reply event to acknowledge packet reception. Similarly a host node may schedule an event for itself to retransmit a packet in the case that the host does not receive an acknowledgement after a certain time. As we can see, it is this modeling of a complex, causally-related chain of events (a relationship that may be unintuitive or difficult to discern otherwise) that makes DES useful and interesting.

The future event set or list is a data structure that implements, at minimum, the following operations.

insert(E, t)	Inserts a given event E with timestamp t.
extract$()$	Extracts the event with the lowest timestamp.

When we consider how events may not be scheduled in the same order they are processed, that is, the sim time associated with an event yet to be inserted into the event set may come before (i.e. be lower than) the current lowest timestamp. This is illustrated in Figure 4.1.

The underlying data structure is thus similar to a priority queue with the key difference that insertions can occur at any position. Such a data structure should thus be

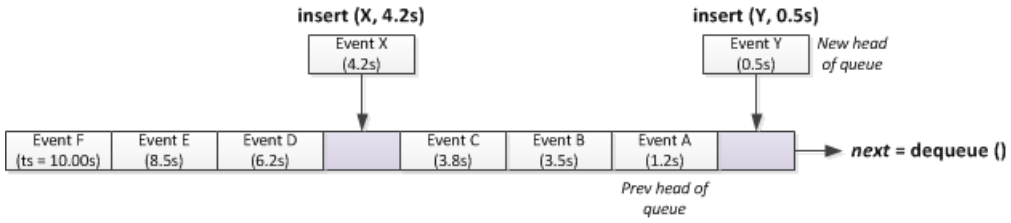


Figure 4.1: Future event set

optimized for fast insertions of this nature. Implementations using doubly-linked lists, set-associative containers such as the C++ STL `std::map` class and others can be found. [7][8] A *calendar queue* is a data structure, first introduced in [31], designed specifically for the future event set problem, and claims $O(1)$ performance for both `insert` and `extract` (or *enqueue* and *dequeue*) operations. Other implementations are given in [32] and [33].

4.3 Random Variables

In our analysis, we have thus far danced around the subject of stochastic aspects in DES a bit. However, random variables are a component common to many brands of simulation, DES included. The range of outcomes of events may be deterministic or they may belong to some probability distribution. In our earlier example of packet loss and retransmission, the communications channel may be modeled statistically using random variables, which will cause packet reception to either be successful or unsuccessful with some probability.

4.4 Parallel and Distributed Simulation

Parallel programming involves decomposing a program into discrete elements that can be executed simultaneously by *tasks* utilizing multiple computational resources, such as microprocessors or computers connected by a network. Many problems that can be solved with a sequential program or algorithm can be dissected into constituent parts to a make parallel version of the same algorithm. Many problems in modeling and simulation are natural candidates for *parallelization*. Although discrete event network simulation undoubtedly belongs in this category, properly identifying the optimal way to break up the simulation, a technique known as domain decomposition, is a non-trivial matter.

Parallel computer architectures come in many shapes and sizes. A Symmetric Multiprocessor (SMP) is a type of Uniform Memory Access (UMA) system where several identical processor *cores*, often fabricated on a single integrated circuit or *die*, share a common memory space [34]. Cache and resident memory are accessed symmetrically with uniform access times for each SMP core. Non-Uniform Memory Access (NUMA) is another type of shared memory architecture combining multiple SMPs that possess their own local memory space, which is mapped into a global address space shared across processors. Programming for shared memory systems is typically done with a construct known as *threads*, which are individual execution paths within a single process.

A distributed memory system involves multiple SMP machines networked together, where memory spaces are local to each machine. As a result, in a distributed program-

[7] Ns-3 implementations of each data structure is documented in [74].

[8] See [30] for the runtime performance of these data structures.

ming model, tasks executing on processors on separate computers must communicate explicitly over the network to access remote resources. This concept is also known as the *message passing* model. Technologies such as Remote Direct Memory Access (RDMA) allow direct access to memory remote machines, or *nodes* in a distributed memory *cluster*, without involvement of the CPU on the remote host [51]. Practical clusters, like the one we describe in 8, are a hybrid of shared and distributed designs and incorporate a multi-level hierarchy of both uniformly and non-uniformly-accessed tiers of memory. Parallel programs utilizing these types of systems should be designed with awareness of the underlying architecture in order to optimize performance in assigning tasks to different processing units and organizing the data set across regions of memory with non-uniform access times.

4.4.0.1 Speedup

Improving program performance (that is, cutting runtime) is the purpose behind all of this added complexity. The potential performance gain or *speedup* offered by a parallel program over a sequential version of the same program is given by Amdahl's Law [34].

$$speedup = \frac{1}{\frac{P}{N} + S} \tag{4.1}$$

Here P is the fraction of code can be effectively parellelized, N is the number of concurrent tasks and S is the serial (sequential) fraction of code ($S = 1 - P$). If the domain of a program could be perfectly decomposed ($P = 1$) and executed independently by N independent tasks, then our program would theoretically run faster by a factor of N. Unfortunately, this special case, known as an *embarassingly parallel* problem, is not what we find in most practical situations. Tasks must often communicate and coordinate with one another to solve a problem and *synchronization* between execution paths must be maintained in order to compute portions of code in the proper order. Communication and synchronization are the foremost factors that limit the possible speedup of a parallel program and make programming with either the multi-threaded or message passing approach challenging. [18].

4.4.1 Multi-Threaded Model Parallelism

As we shall illustrate each of the parallel programming paradigms in this and the following sections with practical examples of how they are applied in our implementation in Section 7, we will keep this part of the discussion as succinct as possible. On Unix and GNU Linux platforms, the standard for multi-threaded programming is the POSIX Threads or *pthreads* library, which is a lightweight solution that gives the developer a simple but powerful API for creating, destroying, controlling and synchronizing user-space threads [35]. Pthreads are spawned or *cloned* from a process (the original instance of a program) and branch off into separate paths of execution. Threads can then be *joined* and merged with the parent process once their concurrent section of code has been executed and their task is complete. They also share the memory space of the parent process and so must coordinate with one another to control access to sections of memory to prevent *race conditions* where multiple threads competing over access to variables may result in memory being placed into an invalid state.[9]

There are several constructs for synchronization using Pthreads. Mutual exclusion

[9] See [36].

or *mutex* statements allow for a *critcal section* of code to be *locked*, permitting access to variables by only one thread at a time. *Sempahores* are often used to implement a pool of resources that can be accessed by a set number of threads at a time. While we shall not cover the all the intricacies of Pthread synchronization primitives, the key concept to take away is that locking is intended to provide *thread safety*, which essentially means a guarantee of program consistency and deterministic behavior when multiple threads are competing for shared resources.

Another situation necessitating synchronization occurs when multiple threads must arrive at a common point in the execution of a program before continuing. This is done by creating a *barrier* at such a point in the code.

4.4.2 Distributed Model Parallelism

With distributed memory systems, processes running concurrently on the same or separate computing resources do not share access to program variables and must coordinate to run a common program by explicitly sending and receiving messages to other processes. We call the case when processes, or *ranks* in the parlance of parallel programming, communicate with other ranks on the same processor or SMP-based machine (that is, specific to one cluster node) *intra-node* communication, whereas *inter-node* communication involves communicating over a network interface with processes on remote nodes. It is straightforward to see how the latencies involved in the latter case are bound to be greater due to added delays incurred by the hardware bus, host-channel interface and networking hardware such as switches. This aspect comes into play in domain decomposition, as certain processing activities that require a lot of interprocess communication are better off being confined to a single node as much as possible. Message Passing Interface (MPI) is the *de facto* standard for programming for distributed memory systems [37]. Both *blocking* and *non-blocking* routines are provided, whereby communication can respectively be done synchronously or asynchronously (i.e. without waiting for the operation to return). Also the notion of a barrier exists in MPI the same as it does in Pthreads.

4.4.3 Hybrid Model Parallelism

We have thus far introduced two popular approaches for programming for shared and distributed-memory architectures. Threads that natively share a memory space can implicitly communicate, making this approach efficient in terms of memory and simplifying the task of programming. Threads are therefore generally regarded as *lighter* than full processes as they require less resources (e.g. memory and registers) and take less time to create [36]. The Pthread API, however, offers no primitives for inter-node communication; Explicit inter-process message passing is required in this case.[10] This presents the need for a hybrid approach that combines the strengths of multi-threaded and message passing programming, where components of a program requiring frequent communication and synchronization between processing elements are parallelized using pthreads spawned from one process, and can communicate with threads belonging to other processes using MPI routines. We have previously brought up the importance

[10] OpenMP is an alternative framework for doing distributed-memory programming that involves combining the local memory space from distributed resources into a single virtual space, with inter-node communication done implicitly. This affords the programmer a layer of abstraction that enables one to program for distributed architectures in much the same way as the multi-threaded model. While we did not assess or compare the two approaches ourselves, research (see [49] and [50]) suggests there are efficiencies offered by the hybrid pthread-MPI model over OpenMP.

of efficiently mapping logical tasks to processing resources in the sense of identifying communications and synchronization bottlenecks in the underlying hardware platform and operating system and designing a program to be conscious of these limitations. With a hybrid model design, threads and processes can be statically or dynamically assigned to resources to provide for efficient hardware resource utilization as we shall see in Section 5.

4.4.4 Logical Process Partitioning

Now that we understand the fundamental tools of parallel programming, we proceed to discuss how they can be applied to improving the performance of discrete-event network simulations by executing subsets of simulation events concurrently. In a practical communications system, nodes such as hosts and routers communicate with other nodes through exchanging packets. Here nodes are the sources and sinks of all data and network events. Similarly, in DES simulation of networks, events are typically associated with specific nodes or, more precisely, with individual network protocols or interfaces. By characterizing these relationships between events and simulation objects (or aggregates of objects), we can identify how the simulation can be *partitioned* into separate components that can be executed simultaneously. We call these partitions *logical processes (LP)* because they are a logical division or decomposition of the overall physical process being modeled. LPs take the form of individual simulations with their own unique event set and clock, however they must collaborate with other LPs to run the simulation as a whole. While an effective partitioning scheme reduces the amount of communication needed between LPs, it is not possible to eliminate it entirely because partitions may not be completely disjoint. Some degree of synchronization is always necessary to maintain simulation consistency and causal order between events assigned to different LPs.

4.4.4.1 Topology Partitioning

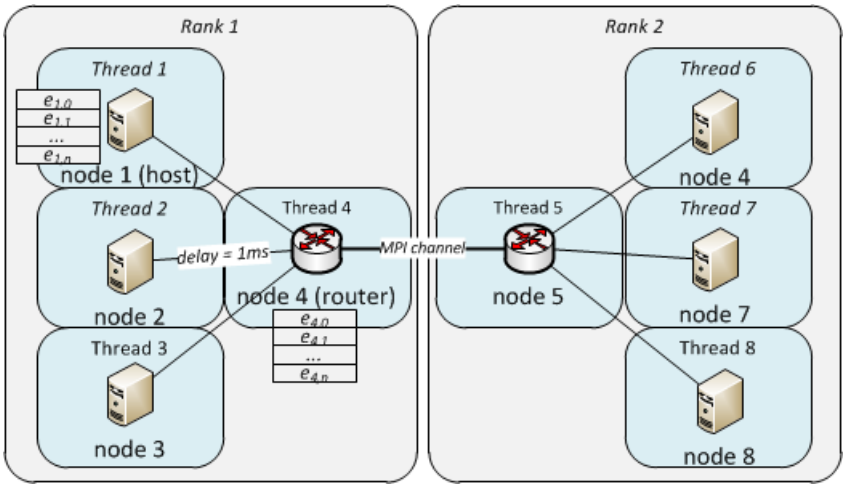


Figure 4.2: Network topology model with one-to-one mapping of nodes to logical processes.

The most straightforward approach to partitioning is on a node-wise basis, that is, assigning single nodes or clusters of nodes to different LPs. To illustrate this concept, Figure 4.2 shows a one-to-one assignment between nodes and LPs, where each LP is executed by a thread. Each LP then has an event set and simulation clock dedicated

to a specific host or router. As the simulated topology is scaled up and the number of nodes exceeds the number of available threads, multiple nodes must be grouped into an LP, with their combined event set processed sequentially. A topology partitioning strategy should efficiently distribute the workload experienced by each LP by balancing the number of nodes assigned to certain LPs.[11] This strategy may need to take into account the properties of the topology, as some nodes may present a serial bottleneck for events. The two routers depicted in Figure 4.2, for instance, may have a higher average workload than the edge hosts since they must forward all packets between each subnetwork. This may require more processing resources to be devoted to routers than to groups of hosts.[12]

4.4.4.2 Horizontal Parallelization

An alternative approach presented in [21] and [27] is the *horizontal* parallelization of events and involves identifying causually-linked sequences of events, termed *expanded events*, that must be processed in sequential order (sequences that cannot be processed in parallel, in other words). As an example, an expanded event might be a series of send and receive events for a specific packet flow that spans multiple nodes, or might even take place within a node as a packet is encapsulated and passed down through protocol layers. A scheduler singleton distributes expanded events to processing resources to be executed concurrently during the *parallelization window* determined by the duration for which each expanded event is known to be causally unrelated. This method may offer a potential performance gain in the presence of workload bottlenecks like we have discussed, since the load can be decomposed with a finer granularity than that of a node. However, there is an inherent complexity in identifying related events and the resulting synchronization window, and performance may suffer (due to frequent synchronization, for instance) if optimal determinations are not made. A simple topological partitioning scheme, on the other hand, does not rely on examining each event, and so the event scheduling algorithm is straightforward and incurs little overhead. Also for larger topologies, it may not make sense to partition with any finer granularity than a cluster of nodes.

4.4.5 Synchronization Algorithms

When a simulation is divided into multiple logical processes that are executed concurrently, it may not be the case that events assigned to different LPs are processed strictly in timestamped order as they would be in a sequential simulation. In fact, if each LP were constrained in such a way, there would be no opportunity for parallelization at all. Nonetheless, a parallel simulation must always produce the same results as its sequential form, meaning that causally-related events must be guaranteed to execute in the proper order. In order to maintain *simulation consistency*, the following constraints must be upheld.[13]

i *State variables cannot be shared between logical processes.* Any changes to local state variables must occur as a result of a local event. This is to avoid situations such as race conditions where an LP, in the process of executing an event, changes the state of a simulation object that affects an event being executed simultaneously

[11] [19] quantifies workload as the average rate of events per unit time for an LP.
[12] Load balancing and topological partitioning strategies are detailed in [19] and [20].
[13] The foundations for parallel simulation we review in this section are taken from [14].

on a remote LP.

ii *Local causality constraint*: Each LP must process its own local events in nondecreasing timestamped order. This includes events generated by remote LPs for the local LP.

While *causality errors* related to the first condition are easily prevented, ensuring LPs adhere to the second constraint is a challenge. In Figure 4.2, *LP 0* (i.e. the LP assigned to *node 0*) must decide if event $e_{1,0}$ can be executed concurrently with $e_{4,0}$ such that the former does not effect the latter or visa-versa. For example, event $e_{1,0}$ might cause the generation of a routing table update or link state advertisement event for *node 4*, which should be scheduled to be received before event $e_{4,0}$. *LP 4* therefore must have some information about the event set for *node 1* before event $e_{4,0}$ is known to be safe for processing. This problem of guaranteeing ordered event execution for local LPs is known as the *synchronization problem*.

4.4.5.1 Conservative Algorithms

In parallel DES, a conservative synchronization algorithm is one that determines ahead of time that an event is *safe* for processing. More generally, its job is to find the maximum simulation time up to which each individual partition clock can safely advance during an iteration of parallel event processing [83]. It must guarantee that no events will be scheduled by remote LPs with timestamps lower than that of any local event due to be processed in the current iteration. Going back again to Figure 4.2, if we know the delay on the simulated channel between *node 0* and *node 4*, we know that any communication between the two nodes takes at least that amount of time (1 ms in this case). We call this minimum transmission delay between partitions the *lookahead* across a simulation boundary. Therefore if *LP 4* knows the lowest timestamp in the event set for *LP 0*, it also knows that *LP 0* cannot schedule local events after this timestamp, which we call the *Lower-Bound Timestamp (LBTS)*, plus the lookahead between partitions 0 and 4. If a packet TX event on *node 0* causes a RX event to be scheduled on *node 4*, the RX must not occur before the timestamp of the TX event plus the 1 ms lookahead delay on the channel.

The classic conservative algorithm for solving the parallel DES synchronization problem is the Chandy-Misra-Bryant algorithm, and works by performing the same causality safety check we just described [22, 23]. Each logical process (partition) calculates the LBTS before each iteration of event processing as follows. To do this, each LP exchanges *null messages* with neighboring LPs containing the lowest local event timestamp. Algorithm 4.1 outlines the general structure of the null message passing and LBTS calculation procedures. In Section 7, we provide both parallel and distributed implementations of Chandy-Misra-Bryant, the key difference between them being how $SendNullMessages()$ and $RecvNulLMessages()$ are performed. A simplified version of the simulator main loop run by each LP is shown in Algorithm 4.2. At the beginning of each iteration, we determine the global LBTS by calling $Synchronize$. We then receive *event messages* from remote LPs. As stated in the first causality condition, events cannot be directly inserted into event set for a remote LP. In order to schedule an event on another partition, an LP must send an event message including the event and its timestamp.[14] The event message parsing loop then scheduled each

[14]Since each LP clock may be at a different simulation time, the event message timestamp should be

Algorithm 4.1 Chandy-Misra-Bryant variant [83]

```
1: procedure SYNCHRONIZE(p)                          ▷ Called for the current partition p
2:     local_lbts ← P.GETNEXTEVENTTS()               ▷ Get the lowest timestamp in the local event set
3:     SENDNULLMESSAGES(local_lbts)                  ▷ Send null message to neighbor LPs
4:     null_messages ← RECVNULLMESSAGES()            ▷ Receive null messages from neighbor LPs
5:     max_ts ← ∞
6:     for all m ∈ null_messages do
7:         if max_ts < m.lbts then
8:             max_ts ← m.lbts
9:         end if
10:    end for
11:    return max_ts
12: end procedure
```

Algorithm 4.2 Simulator main loop [83]

```
1: procedure MAIN_LOOP
2:     while simulation not finished do
3:         for all p ∈ partitions do
4:             p.max_ts ← SYNCHRONIZE(p)                          ▷ Get the global LBTS
5:             P.RECVEVENTMESSAGES()
6:         end for
7:         for all p ∈ partitions do
8:             for all event_msg ∈ p.event_messages do            ▷ Enqueue events from event messages
9:                 P.EVENT_SET.ENQUEUE(event_msg.event, event_msg.timestamp)
10:            end for
11:        end for
12:        while P.GETNEXTEVENTTS() ≤ p.max_ts do                 ▷ Process partition events
13:            PROCESS(p.event_set.Dequeue())
14:        end while
15:    end while
16: end procedure
```

event in the queue for its respective partition. Finally, for each partition, we process all events up until the global LBTS. For didactic purposes, the procedure we provide here is a simplified, sequential algorithm, however it is straightforward to see how each loop might be parallelized. For instance, multiple threads can be dynamically dispatched to process partitions from a pool of partitions in the second *while* loop. We shall delve more deeply into the parallel incarnations of this algorithm in our discussion of core simulator implementation in Section 7.

4.4.5.2 Optimistic Algorithms

Optimistic synchronization algorithms take advantage of the principle that instances of desynchronization (i.e. causality violations) occur infrequently, so a performance gain is offered by allowing LPs to process events unrestrained by periodic synchronization [83, 28]. When a violation of the causality constraint does then occur, the simulator provides a means of recovering by interrupting the simulation and *rolling back* states to the point before the infringing event or series of events. One possible way of implementing this would be to provide a way to *checkpoint* and save the entire simulation state to roll back to in the event of desynchronization. Alternatively, an event *undo* function could be implemented to revert to some incrementally saved-state.

Optimistic algorithms such as used in the Georgia Tech Time Warp (GTW) DES system are still the subject of much ongoing research in the field of parallel DES as they have been shown to offer improved performance over conservative methods [28].

the absolute time (i.e. total elapsed time since the simulation start event) for the sending LP.

Unfortunately, they are fundamentally impractical for a real-time simulator such as we are pursuing. As we shall discuss next, real-time synchronization relies on matching the pace of events in simulation time with wall-clock time. Since events may result in direct real-time feedback to actual applications interfaced with the simulation, there is no opportunity to roll back events if desynchronization were to occur.

4.5 Real-Time Emulation

In a real-time DES, simulation objects are made to mirror the timing of the physical system being simulated [38, 16]. This is achieved by delaying event execution, effectively mapping simulation time into wall-clock time. In our simulations of networks, we wish for the timing behavior of network objects to reflect the real-world devices, nodes, links, etc. as closely and accurately as possible. Unfortunately there are fundamental limitations in real-time simulation that prevent us from reproducing this behavior precisely. Firstly, by the very nature of discrete-event simulation, events are represented as occurring at some instantaneous point in time. We know that, for most real-world systems, the processes simulated by said events exist in the continuous-time domain.[15] We can often get around this problem by decomposing events into *start* and *end* events, if you will, to represent the process as having taking place over the time difference between the two sub-events. We shall further illustrate what we mean by this soon. Another more troublesome issue is that multiple events may be scheduled to execute at the same instant of time, even for the same simulation object. We might see this case with a router node that is receiving packets simultaneously on multiple interfaces. While an actual router may have hardware that supports pipelining of the forwarding plane, our simulation is forced to process these events serially, making it inherently impossible to perfectly capture the timing of the target system. Therefore, even with parallelizing events over multiple concurrent execution paths, some degree of deviation from real-time is unavoidable.

Our motivation for pursuing a real-time simulator is to provide emulation capability, enabling real applications and protocols to interface with out simulated software-in-the-loop environment. Of course, the real-time aspect of such a system is essential, otherwise applications performance will be unrealistic when interacting with the simulation. The simulator must be able to execute the sequence of events fast enough to keep pace with wall-clock time. As we scale up the simulation size to more and more network elements, the amount of necessary computation will eventually exceed the ability of a single processor to maintain this pace. As mentioned, we seek to mitigate this issue by making use of parallel simulation techniques. Furthermore, we believe that if the deviation from real time can be made acceptably small, the matter of imprecise real-time synchronization may not be such a critical drawback in the sense of corrupting results. Since simulation events still happen in the simulation time domain, all internal actions will be represented as taking place at the exact instant they are scheduled to, regardless of the actual wall-clock time that they are delayed by.

To illustrate this point, Figure 4.3 and Figure 4.4 show a hypothetical example involving wireless communication between a base station, STA 1, and a mobile host. At time $t_{sim} = 0$, a transmission event is scheduled for execution (shown in blue in Figure 4.4). There is some lag time or *jitter*, however, between the simulation time at which the event is logged and the real (wall-clock) time when its execution starts and completes. We also see that, while processing the event takes some duration of wall-

[15] Take, for instance, the reception and decoding of an LTE subframe as reviewed in Section **??**.

clock time (during which state variables may be read and written to at various times), to the simulation, the event is perceived as having occurred at a single instant. *Event 1* then triggers a reception event (*event 2*) at the mobile to occur after the propagation delay of 0.001 ms.[16] However, due to jitter in processing, the real-time at which *event 2* ultimately gets queued is already later than the 0.001 ms time at which the event is scheduled to occur. At first glance, this would seem to be problematic when considering the causality constraint. However, we should note that as long as the wall-clock time at execution is greater than or equal to the scheduled simulation time, there can never be a violation of causality.

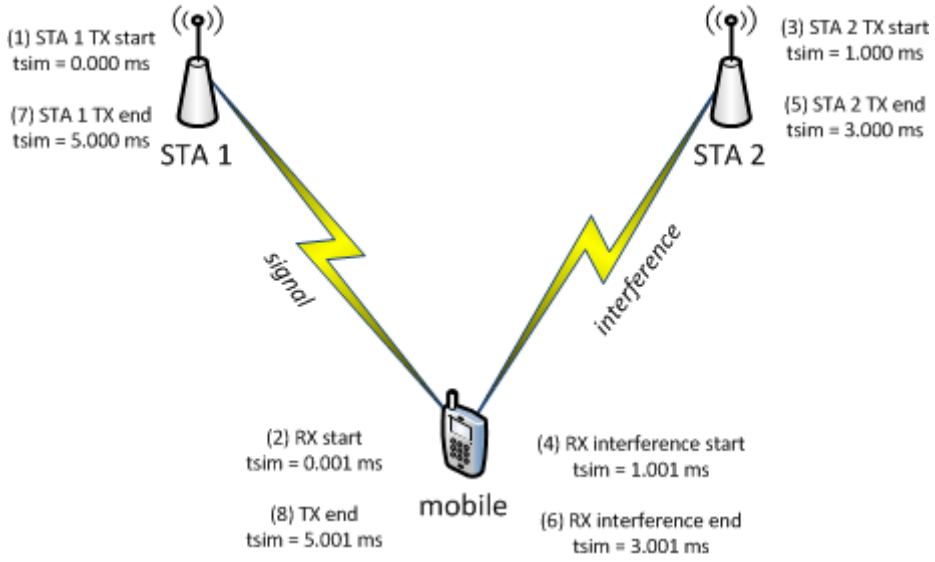


Figure 4.3: Co-channel interference simulation events

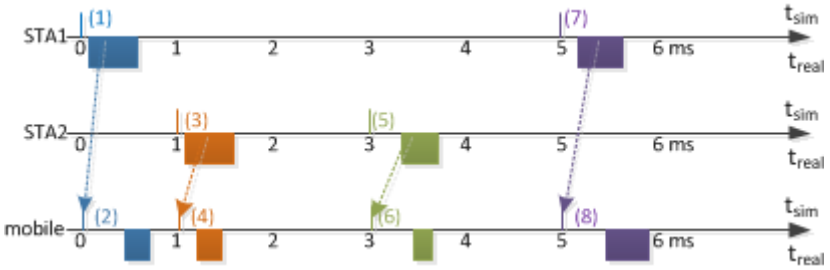


Figure 4.4: Co-channel interference simulation timeline

Next we see a co-channel interferer, STA 2, beginning a transmission to some mobile (not shown) in *event 3*, scheduled for time $t_{sim} = 1ms$ (shown in orange). This transmission is heard as interference at the mobile host after the hypothetical channel delay (also 0.001 ms) in *event 4*. At some fraction of a millisecond after, the mobile initiates calculation of the SINR that will be used to determine the probability of error for the data block in the final *RX end* event (i.e. *event 8*). While *event 3 and 4* are logged as having taken place at precisely 1 ms and 1.001 ms, respectively, we see in Figure 4.4 that both events do not finish until some short wall-clock time later. Furthermore, suppose that *event 3* involves some computationally-intensive operation and takes

[16]This is a purely hypothetical scenario meant to demonstrate real-time delays and is not specific to any wireless technology.

significantly longer to execute than *event 4*. As a result, the transmission event does not complete until after reception! Again, this does not present an issue, as the simulation times at which the events are logged as occurring are still valid. At $t_{sim} = 3ms$, *events 5 and 6* (green) respectively mark the completion of the transmission by STA 2 and the termination of the interference event. Finally, the 5 ms-long transmission from STA 1 completes in *event 7* (purple), which triggers the *RX end* in *event 8*. The ultimate error calculation depends on the duration and power level for both the data and interference signal. If the duration for each signal were based on the wall-clock time elapsed between start and end events, the calculation would clearly be wrong. Fortunately, this is not the case, however, since only the simulation times at which events are scheduled are taken into account.

To consider how real-time jitter can have an effect on results in the context of emulation, suppose that the mobile in our example represents a wireless modem, which is interfaced with a *virtual device* on the physical computer running the simulation.[17] While the simulated network is nothing more than a program on the host system, it appears to the system that it is connected to an entire virtual wireless network through this virtual interface. In this way, the virtual device provides a *tap* into the simulated world. Now suppose that the physical and MAC layers are emulated by the simulator program and the virtual device handles passing data between the IP daemon on the host system and the emulated MAC layer. At the completion of *event 8*, the wireless modem decapsulates the MAC data and passes the IP PDU up the stack. While this entire process was intended to take place at $t_{sim} = 5ms$, the data does not become available to higher layers until nearly a millisecond later. It is clear to see how this non-deterministic jitter between simulation time and real-time can produce invalid black-box behavior in such a situation. Some jitter is always present and should be accounted for when considering the validity of results. In some scenarios, such as measuring the Round-Trip Time (RTT) of an ICMP message between our example base station and mobile host, results would be significantly skewed by any amount of real-time jitter. If we are measuring the RTT between some host several hops away in the network, however, we may be able to chalk up the added latency to some non-deterministic variance in delay that might be found in any real-world network. In cases like these, some degree of timing error may be acceptable, and so the simulation should ensure that certain real-time *deadlines* are met. These deadlines may be defined based on a tolerable jitter for specific types of events, and the simulator should provide some mechanism to record when deadlines are exceeded to indicate potential invalidation of results.

5 Operating System Considerations

In the preceding section, we introduced some fundamental limitations and constraints of real-time, parallel and distributed DES that are universal to all hardware and operating system platforms. However, we cannot hope for our parallel algorithms and methods for enabling real-time performance to perform optimally unless they are designed to take stock of some important hardware and OS aspects. When programming for parallel computing architectures, an understanding of the system *task scheduler* is essential when attempting to optimize a program. Additional considerations must also be made to attain real-time performance of software running on a natively non-real-time operating system. We address these factors in this section and later elaborate on their application

[17] We jump ahead to the topic of virtual (i.e. tap-bridge) devices, discussed in Section 7.2, to demonstrate our point.

in a practical compute cluster in Section 8.

5.1 GNU/Linux Scheduler

The Linux operating system (that is, any GNU distribution based on the Linux kernel) has been widely embraced by the high-performance and scientific computing communities for many reasons, including its stability and configurability as well as the availability of free, community-maintained, open-source software. The heart of the Linux OS is the kernel, which is responsible for mediating between *user space* applications and hardware [41]. Like most modern OSes, Linux supports multi-programming and multi-tasking, allowing multiple tasks to share the CPU and run simultaneously (or, at least, with the illusion of concurrency on a uniprocessor system). Dynamically allocating CPU resources to competing tasks is the job of the kernel scheduler, which allocates CPU time *slices* to tasks based on their priority and factors such as CPU utilization. We shall not touch on all the finer points of the Linux scheduler, as it is an extensive subject on its own, however we focus on several considerations that will allow us to come as close as possible to achieving real-time behavior on an inherently non-real-time OS such as Linux. Foremost we are concerned with controlling context switching, the process by which a task (i.e. process or thread) is interrupted after the expiration of its time slice in order to yield the CPU to other tasks. A task must then must save its state in order to be resumed later. Time spent during context switches amounts to time wasted by the CPU, when no actual work is performed toward executing user-space tasks. For our simulator program to give the best performance, our hardware resources must be dedicated entirely to supporting this program alone. Therefore we should provision for program tasks to utilize as much as the CPU as is allowed by the kernel, which means reducing context switching as much as possible. There are several steps we can take to reduce unnecessary context switching and processor churn, increase CPU utilization, and generally improve the efficiency of the Linux OS.

- *Disabling all non-essential programs and services*, such as file and mail server daemons, will aid in limiting context switching.[18]
- *Disabling the power management service* (i.e. the ACPI daemon) ensures that the system does not throttle CPU utilization in attempt to reduce power usage.
- *CPU affinity* is a property specified for tasks that causes them to be bound to certain processor or core of an SMP, overriding the scheduler function that dynamically handles the assignment of threads and processes to processors. By manipulating CPU affinity, we can reserve specific SMP cores for individual tasks and theoretically eliminate context switching entirely.[19]
- *Setting task priority* is another way in which we hope to manipulate the scheduler to run our simulator program more efficiently.[20]

[18] See [42] for a list of standard daemons on RHEL/CentOS 5 systems.

[19] [44] and [45] document the *taskset* and *cpuset* utilities that are used manipulate the system scheduler. Also [46] gives examples of setting CPU affinity for Open MPI processes.

[20] Real-time performance is possible with the 2.6 Linux kernel compiled with the *CONFIG_PREMPT_RT* patch [47]. This allows user-space tasks to be assigned real-time priority. In Real-Time Linux, the *chrt* utility is used to change the priority of threads and processes [48].

6 LTE/SAE: 4G Cellular Evolution

6.1 LTE Overview

Long Term Evolution (LTE) is the latest series of standards developed by the 3rd Generation Partnership Project (3GPP) to become a mature Radio Access Technology (RAT). It is so named because it seeks to evolve the 3G UMTS radio interface through the next epoch of cellular wireless technology. Specifically, LTE refers to 3GPP Release 8 and beyond of the Evolved UMTS Terrestrial Radio Access Nework or E-UTRAN. Throughout this document, we shall use the terms LTE and E-UTRA/E-UTRAN interchangeably. The developmental LTE Advanced system introduced in Release 10 is planned as an upgrade to the first LTE release and is among the first standards to be certified as 4G-compliant by the International Telecommunication Union. LTE (and, to an even greater extent, LTE Advanced) boasts of some unprecedented, not to mention rather impressive, specifications for data rate, capacity, spectral efficiency and latency made possible by modern advancements in wireless technology and principles such as MIMO, interference coordination and new multiple access schemes. These and other changes to the UMTS radio interface and protocols were necessary to achieve such ambitious requirements.

As an important side note, while Release 8 is presently being rolled out in cellular networks globally, Release 10 has yet to be implemented by vendors in any broad sense and production systems are not anticipated for any time in the near future. We therefore choose to concentrate our efforts on grasping the Rel-8 LTE radio interface and, as a central goal of this project, the development of a simulation model of the same. We believe this would be of greatest immediate value to researchers and developers in the industry and academic community since the intention is to provide a solid foundation for 4G cellular network simulation. The features added in Rel-10 can subsequently be integrated into the basic framework.

The chief features and requirements for LTE laid out by the 3GPP and frozen in the Release 8 specifications are as follows [56, 60, 61].

- *Increased data rates to user equipment:* Targets of 100 Mbps peak rate in the downlink and 50 Mbps in the uplink with 20 MHz system bandwidth.
- *Reduced setup and transfer latency:* Sub-5 ms one-day delay (from when a packet is transmitted until it is available at the IP layer at the UE) and less than 100 ms when transitioning from idle to active mode.
- *Improved spectral efficiency*: Average user throughput per MHz 3-4 times better in the downlink and 2-3 times better in the uplink than UMTS Release 6, owed to new OFDMA (downlink) and SC-FDMA (uplink) multi-access techniques.
- *Flexibility in spectrum usage and support for various cell bandwidths*: Support for 20, 15, 10, 5 MHz bandwidths and under for more scalable cell sizes.
- *Improved UE power management*: Extended terminal battery life using Discontinuous Reception (DRX) and other power-conserving techniques.

6.2 SAE Overview

The System Architecture Evolution (SAE) work item is the core network counterpart to LTE in the RAN and evolves the 2G-3G GPRS network into the entirely packet-switched Evolved Packet Core or EPC, as it is known. The EPC combined with the E-UTRAN is jointly referred to as the Evolved Packet System (EPS), which encompasses

the 3GPP vision of the 4G mobile broadband system in its totality. The Evolved Packet Core is made up of collection of elements that provide backbone data transport (i.e. user plane) and control plane functions such as mobility management, policy and charging control and Authentication, Authorization and Accounting (AAA). The nodes of the EPC are also responsible for setting up and maintaining end-to-end connections with specific QoS requirements. The underlying motif throughout the development of SAE has been the departure from separate circuit-switched and packet-switched components for voice and data traffic, respectively (as has been the case since the introduction of GSM/GPRS), into a much simplified, "all-IP" core network. This factor along with the deliberate separation of RAN and core functions enables a greater flexibility in internetworking, allowing different RATs to be served by the same core network infrastructure. Below are some of the general requirements for the EPC drawn up in the SAE work item [57, 58].

- *Simplified, flat, all-IP network architecture*: As the name implies, the Evolved Packet Core is a packet-based system, with the separate circuit-switched domain used in GSM and UMTS for provisioning voice calls being completely eliminated. The separation of user and control plane functions into different logical network entities is clearly defined. Application layer services for voice, data and multimedia, such as those provided by the IP Multimedia Subsystem (IMS), are all provided on top of and independent from essential network functions. Whereas previous systems allowed for different transport protocols such as ATM and SS7 between core nodes, the EPC relies entirely on IP for network layer connectivity. Furthermore a "flat," as opposed to a hierarchical, design philosophy was adopted, meaning in a very basic sense that as few nodes as possible are involved in processing user data traffic [58].
- *Heterogeneous internetworking*: Support for internetworking between modern and legacy 3GPP and non-3GPP systems. Multimode UEs can seamlessly handover between different RATs with minimal (sub-300ms for real time and sub-500ms for non-real time services) interruption time [61]. This capability s considered a guiding principle of SAE since such factors as roaming and ease of migration are so important from the network operator perspective.
- *Common security framework*: Security and privacy features are built into every aspect the EPS. Each stratum of the network, from the application layer services to the raw traffic going over the wireless channel, is secured through mutual authentication, cyphering and access control policies.
- *Common QoS, charging and policy control*: QoS over end-to-end connections (in truth only applying to traffic within the domain of the EPS) is provisioned through EPS *bearers*, as we shall see shortly. Treatment of traffic can be differentiated in the user plane through QoS Class Identifiers or QCI as well as in the control plane with Allocation and Retention Priority (ARP) parameters. Policy and Charging Control (PCC) network entities including the Policy and Charging Rules Function (PCRF) provide a centralized mechanism for doing what the name implies, enforcing QoS policies and charging on a per-subscriber and per-service basis. The specific roles of each of these entities is discussed in the following section.

With demand for mobile data services skyrocketing, the enhanced, data-centric, low-latency, high-capacity cellular services promised by LTE/SAE could not have come

at a better time. While numerous other competing standards have emerged alongside the 3GPP standards, it is clear that the Evolved Packet System has already firmly positioned itself in the 4G mobile landscape. As we go over the interfaces, protocols, procedures and network elements of the EPS in the following sections, we tailor our discussion around that which we want to model in simulation. In this first iteration of our work, we focus on implementing a bare-minimum set of features on top of existing features provided in the ns-3 framework and model library that we deem necessary for a proof-of-concept simulation of end-to-end connectivity over the RAN and core network. As a consequence we shall not delve into all the finer points of LTE/SAE but instead provide an overview of the technology and background relevant to the model detailed in Section 9.

6.3 EPC Network Architecture and Protocols

Before we can adequately describe the technical nuts and bolts of the LTE radio interface and protocol stack, we must introduce the essential functions taking place behind the scenes in the core network. Figure 6.1 shows us the logical *network entities* found in the EPC and the *interfaces* between them. We use the term *logical* here to imply independence from any physical server or device in the network. Instead, each logical entity has a specific role in the network with a well-defined functionality, and multiple entities can be co-located on the same hardware. The concept of an interface is similarly abstract and represents a logical connection and relationship between entities. In the 3GPP vernacular, interfaces are referred to as *reference points* and take the form of alphanumeric (often two- or three-character) abbreviations like S11, X2, and S6a. An interface is composed of a set of *protocols* for signaling and data transport between network entities, although the same protocol (GPRS Tunneling Protocol as a case in point) may be found on multiple interfaces. To continue with our definition of terminology, a protocol is a formal set of technical rules and procedures for communication between network entities, typically involving formatting of data and message syntax, control messaging, transmission, error handling, etc. [58].

The above figure shows the key entities and interfaces known to the E-UTRAN and EPC. While these are recognized as the essential components involved in standard network operations, any practical network is considerably more complex. Omitted from this figure are interfaces between other (non-EPS) core and access systems along with elements providing higher-layer services such as IMS and AAA (the reader is referred to [58] for elaboration). The bigger picture of a real-world network deployment, including all of the intricacies of heterogeneous networking and application layer service delivery, are beyond the scope of our work. In future iterations, we hope to continually add non-EPS and even non-3GPP network elements to our model (to simulate heterogeneous handovers, perhaps), however we seek to "keep it simple" for now. We shall refer to these components of the basic EPS topology frequently throughout this document and we shall later detail our approach to reproducing them in simulation.

6.3.1 Network Entities

In the following sections, we expand on the specific roles of each of these fundamental logical entities, with attention given to differentiating them in terms of the various control and user-plane functions they provide.

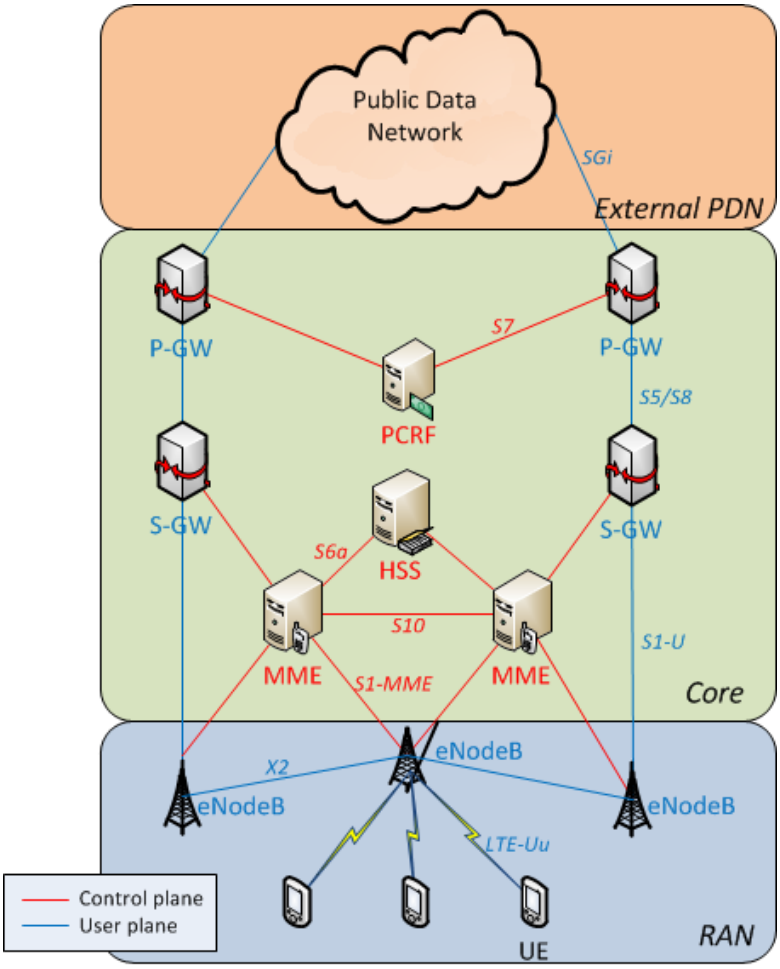


Figure 6.1: Simplified EPC network architecture

6.3.1.1 User Equipment (UE)

As the name suggests, the UE is any end-user device communicating with the base stations (eNodeBs) of the E-UTRAN using the LTE stack. In the age of mobile broadband, the archetypical cell phone is just one of many such devices that take advantage of mobile data services. Feature-rich *smartphones*, tablets, laptops, netbooks, dongles, and smart-metering devices (used by utility companies) are just a few examples of devices presently on the market that are beginning to be equipped with LTE radios. The UE communicates with the eNodeB over the E-UTRAN Uu interface [65].

6.3.1.2 eNodeB (eNB)

The eNodeBs are the base stations that make up the E-UTRAN and provide wireless connectivity for the UE. The term *base station* is used loosely and can refer to any number of cell deployment scenarios. For instance, the common three-sector site deployment covers three cell areas that comprise the same eNodeB [57]. While reminiscent in some ways of the UMTS NodeB, the Evolved NodeB has assumed many additional responsibilities, such as performing header compression and cyphering (encryption/decryption) of user plane data over the air link along with the following Radio Resource Management (RRM) tasks [66].

- *Radio Bearer Control (RBC)*: The establishment, maintenance and release of ra-

dio bearers. The eNB takes stock of available resources along with the requirements of existing sessions and attempts to fulfill the QoS requirements of new sessions when setting up radio bearers (see Section 6.3.2.1 for more on bearers).

- *Radio Admission Control (RAC)*: The eNB must determine if the QoS requirements of requested radio bearer sessions can be met and accept or reject requests accordingly.
- *Dynamic Resource Allocation (DRA) and Packet Scheduling (PS)*: As we shall see in Section **??**, wireless resources can be assigned to users and sessions over the dimensions of both time and frequency in the form of Resource Blocks (RBs). The eNB must leverage the requirements of in-progress radio bearers, CQI feedback information from UEs, buffer status and other factors to optimize scheduling of user data. DRA also involves downlink and uplink power control on a per-RB basis.
- *Inter-Cell Interference Coordination (ICIC)*: An eNB must communicate over backhaul links (X2 or S1-U interface) with eNBs in neighboring cells to perform multi-cell RRM and coordinate the use of wireless capacity.

Some other noteworthy aspects of RRM include cell selection procedures and load balancing over multiple cells. Additionally, eNodeBs coordinates handover procedures with one another (over the X2 interface) and with the MME (using the S1-MME interface). All user plane packets are forwarded between the eNodeB and S-GW over the S1-U interface.

6.3.1.3 Mobility Management Entity (MME)

The MME is the main control node in the EPC and performs a broad range of functions for session management, mobility, gateway selection, tracking and paging procedures for the UE, which are mostly considered part of the Non-Access Stratum (NAS) protocol [58, 67]. The NAS protocol handles all control-plane signaling between the MME and UE. The MME communicates with the multiple S-GW over the S11 interface for such purposes as bearer management and similarly has a one-to-many relationship with eNodeBs connected by the S1-MME interface.[21]

6.3.1.4 Serving Gateway (S-GW)

The S-GW is the termination point of the E-UTRAN into the EPC and is primarily concerned with the operations listed below [57, 58, 67].

- *User data forwarding*: All user-plane data to and from the UE is sent between the eNodeB and S-GW over the S1-U interface. The GPRS Tunneling Protocol (GTP) maps EPS bearers carrying data to and from UEs to tunnels for the purpose classifying user data. The S-GW also buffers downlink data during handovers until bearers can be re-established.
- *Mobility anchoring*: The S-GW is the mobility anchor point for the UE during inter-eNB handover and receives control messages from the MME for tearing down and setting up new bearers. It is also the connection point with S-GWs or SGSNs in other 3GPP mobile networks (or for supporting legacy GPRS in the

[21] While the role of the MME is diminished in the implementation introduced in Section 9, we feel it is still important to give an overview of its principle functionality and associated interfaces.

same network or administrative domain), which forward data traffic on behalf of roaming UEs.

6.3.1.5 PDN Gateway (P-GW)

The P-GW node is the connection point (i.e. the SGi interface) of the EPS to external Packet Data Networks like the Internet. It is connected to the S-GW by the S5/S8 interface, which can be GTP or PMIPv6-based (See Section 6.3.3.3). Its primary tasks are as follows [58].

- *Connections to external networks*: The P-GW is the point of the presence of the core network on a particular PDN. UEs connecting to multiple PDNs (different Autonomous Systems in the Internet, for instance) may so do through multiple P-GWs.
- *UE IP addressing*: IP addresses are allocated to each UE for routing external data traffic at the time of default bearer activation (see Section 6.3.2.1).
- *Packet filtering and traffic shaping*: Deep packet inspection and filtering is performed by applying Traffic Flow Templates (TFT), which supply the QoS parameters for each user session (i.e. EPS bearer).
- *Mobility anchoring (non-3GPP access)*: The P-GW provides the anchor point for UEs roaming in non-3GPP access networks. Such concepts related to heterogeneous networking are, however, beyond the scope of this document (the reader is referred to [58] for more on this subject).

6.3.1.6 Policy and Charging Rules Function (PCRF)

The PCRF is the central node responsible for managing policy (e.g. access control, QoS parameters) and flow-based charging [58] based on subscriber information from the Home Subscriber Server (HSS). While both are important entities in any practical deployment, we forgo thorough attention to the functions provided by the PCRF or HSS in this iteration of our work as these functions are not essential to our basic EPS representation in simulation.

6.3.2 EPC Procedures

Before getting into the technical aspects of the protocols employed by the various entities for network operation, we should understand some of the high-level services that the EPS as a whole must provide. Session management refers to the handling of connections between UEs and PDN hosts. As the user moves, mobility management functions take care of maintaining active sessions. The UE must associate with an eNodeB and core network entities or reassociate as a result of handover (or other events) though the attach and selection procedure. [22]

6.3.2.1 Session Management

6.3.2.1.1 EPS Bearers

User plane sessions over the RAN and packet core are realized through the bearer system [58, 68]. End-to-end bearers are known as *EPS bearers* and, as illustrated above,

[22] Security services are also an integral aspect of practical EPS networks and are provided in some form by each core entity. We omit a discussion of these aspects as they are outside the scope of this work.

are decomposed into three parts:

- *Radio bearers (RB)* carry data over the E-UTRAN-Uu interface and are controlled by the RRC protocol (see Section **??**). Uplink RBs have an associated UL-TFT (Uplink Traffic Flow Template), which include information for classifying application data and associate it with certain QoS paramters used in scheduling (by the eNodeB) and packet filtering at the P-GW.
- *S1 bearers* between the eNodeB and S-GW take the form of GTP tunnels and are identified by the TEID (Tunnel Endpoint ID) parameter in the GTP-U header.
- *E-RAB* or E-UTRAN Radio Access Bearers refer to the combined radio and S1 bearers.
- *S5/S8 bearers* are specific to the GTP-based S5/S8 interface and are also identified by the TEID field. In the downlink direction, the DL-TFT information assigned to S5/S8 bearers provides the QoS values for the session.

Default bearers are established when the UE initially connects to the PDN and remain for the lifetime of the connection. They are usually assigned the default QoS values for standard data traffic. A UE connecting to hosts in multiple external networks has one default bearer for each respective PDN. Any additional bearers are then known as *dedicated bearers* and can be established on-demand as needed by the UE. Each dedicated bearer may have special QoS settings to suit the requirements of specific application data.

6.3.2.1.2 Traffic Flow Templates (TFT) and QoS Profiles

As mentioned, the TFT typically contains such information as source and destination (public) IP addresses and transport layer port numbers along with the Type of Service (TOS) field from IP header, which is used by the UE for discriminating data belonging to different IP flows. The TFT is mapped to a QoS profile for determining MAC-layer scheduling priority at the eNodeB and is also used in packet filtering by the P-GW as data ingresses or egresses the network. The QoS profile of an EPS bearer includes the following fields:

- *QoS Class Identifier (QCI)*: The QCI value specifies the scheduling priority, packet delay budget and allowable packet loss for different types of traffic. Along with

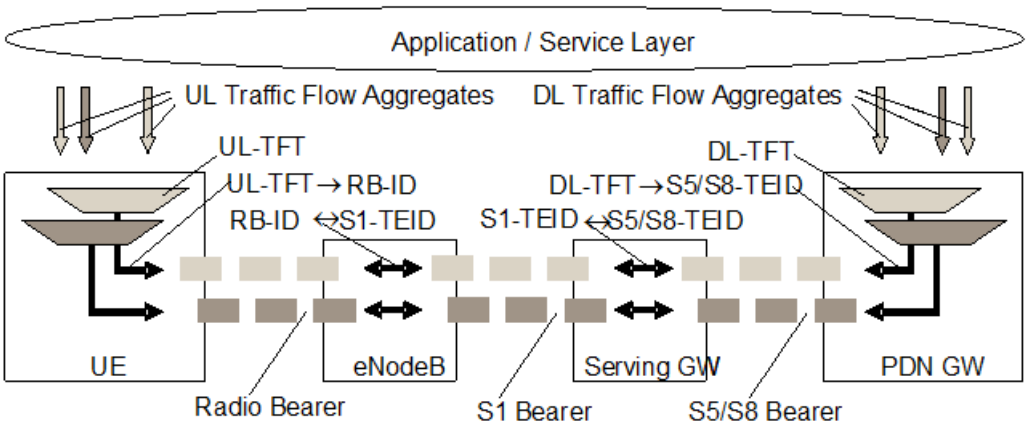


Figure 6.2: EPS bearer (GTP-based S5/S8, figure taken from [68])

being incorporated into MAC scheduling decisions, the QCI determines the RLC mode for each logical channel [56].

- *Allocation and Retention Policy (ARP))*: The ARP is a value between 1 and 15 (1 being the highest priority) specifying the priority level used in Radio Admission Control to determine if a bearer can be established that preempts an existing session.
- *Guaranteed Bit Rate (GBR)*: The expected data rate for an IP flow.
- *Maximum Bit Rate (MBR)*: The maximum allowable data rate for a flow that, which, along with the GBR, is used for traffic shaping.

6.3.2.1.3 IP Connectivity

The foremost role of the EPC is to provide transport of user application data. By application data, we mean data associated with protocol layers including the IP layer and above. Multiple connections between the UE external hosts, identified by the typical "5-tuple" parameters, may be transported over a single *PDN connection*, which is supported by a default or dedicated bearer instance. The IP connections (again we mean end-to-end connections represented by 5-tuple information) are tunneled through the underlying transport network by the user plane entities, which are entirely transparent to the UE. Furthermore, the IP domain of the PDN (including the UE's publicly routable address) is separate from that of the EPS. The UE therefore sees no intermediate hops between itself and the PDN Gateway. We see this situation illustrated in the figure below [58].

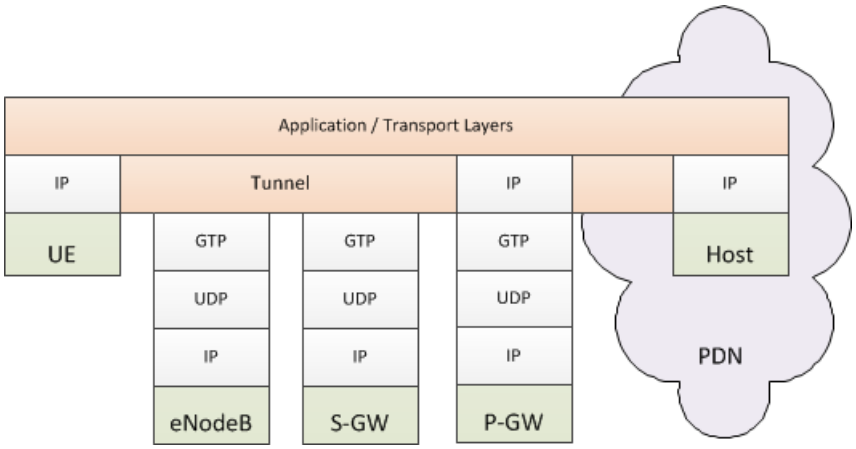


Figure 6.3: Application data tunnel over the EPS (figure based on [58] - Section 6.1.1.3)

6.3.2.1.4 Address Allocation

We mentioned how PDN connections and the core transport network exist in separate IP routing domains. Core network entities may be addressed using a private IPv4 or IPv6 scheme. The address assigned to the UE must be in a public range to be routable on the PDN, which may IPv4- or IPv6-based. IPv4 address allocation may be done during the initial attach procedure as part of the *attach accept* message from the MME.

6.3.2.2 Mobility Management

The purpose of mobility management and, specifically, the chief role of the MME is threefold: (i) Tracking the UE so it can be reached by the network at all times (in the case of incoming connections, for instance), (ii) ensuring the user the ability to initiate outbound connections and (iii) maintaining active connections as the user moves around.

6.3.2.2.1 Tracking and Paging

UEs in the *active mode* (RRC_CONNECTED) state presently have an active connection to the network. A UE can also be in the *idle mode* (RRC_IDLE) state, during which it periodically listens for paging messages from the MME. Section **??** elaborates on these RRC states. Idle mode UEs can move between cells (i.e. eNodeBs) in the same Tracking Area (TA) without notifying the MME. The MME directs paging messages to all eNodeBs in a particular TA in the event of an incoming connection. When moving to another cell outside of its present TA, it must send a Tracking Area Update to the MME so that it can be paged. The reader is referred to [58] Section 6.3.2.1 and [62] Section 7.2 for more information.

6.3.2.2.2 Active Mode Mobility

Active mode mobility involves sustaining active sessions while the UE moves between parts of the network or, in the case of inter-RAT handover (HO), between different access networks entirely. We shall direct our attention to two cases within the EPS domain known as X2 and S1 HO.

X2 HO, also known as *direct* HO, involves detaching from the UE's present serving eNB (the Source-eNB) and associating with a Target-eNB in a neighboring cell. HO events may be initiated after the source coordinates with the target NodeB and determines that the channel conditions in the target cell are significantly better for a specific UE. Coordination is done over the X2 interface, which is the backhaul transport link between eNB. In this case, the UE maintains its association with its current Serving GW and MME. Once the HO decision has been made and resources have been prepared in the target cell, the connection to the S-eNB is dropped before it is reestablished with the target (i.e. "break-before-make"). The S-eNodeB requests the MME to switch the downlink data path to the target eNB, which is then carried out by the S-GW. The source must finally forward any backlogged packets (that arrived for the UE during the process) to the target eNB [58, 62]. [23]

Without a X2 backhaul link, S1 HO (*indirect* handover) must take place using the S-GW as an intermediary. S1 HO is also performed when the source MME determines that the UE should associate with a new S-GW or MME. This process involves the additional steps of reconfiguring the tunnels between P-GW and S-GW along with the end-to-end EPS bearers serving the UE at hand.

6.3.3 Interfaces

Here we address some of the interfaces or reference points found in the EPS (defined in 3GPP TS 23.401), which characterized by standard protocol stacks that facilitate communication between network entities. A practical EPS deployment involves many

[23] A detailed diagram depicting X2 HO is given in [63].

entities and interfaces that we exclude from our discussion, as understanding them is not immediately relevant to our project goals.

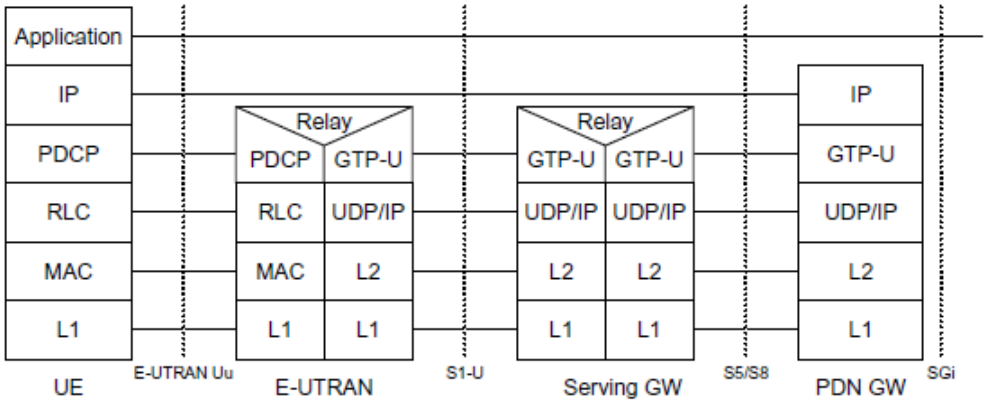


Figure 6.4: EPS user-plane interface (copied from [58]-Section 10.3.10)

6.3.3.1 LTE-Uu

The LTE-Uu (sometimes E-UTRAN-Uu) is the air interface connecting the UE to the eNodeB. In the user plane, the Uu interface is characterized by the LTE PHY, MAC, RLC and PDCP protocols, as we see in Figure 6.4. These protocols are concerned with segmentation/reassembly, cyphering, encapsulation/decapsulation, and the ultimate radio transmission and reception of user data. [24] [25]

Each of these protocols also takes part in the control plane and, along with the RRC protocol, perform control signaling with the eNodeB. The "relay" function shown in the interface diagrams in this section facilitates communication between interfaces of a node. When we later discuss the LENA LTE implementation in Section 9, we see how relaying is performed by the $LteEnbNetDevice$ class. [26]

6.3.3.2 S1-U

The S1-U forwards user data between the eNodeB and Serving GW over GTP-U tunnels (mapped to S1 bearers).

6.3.3.3 S5/S8

The S5 and S8 interfaces are referred to jointly because they both serve to interconnect the Serving GW and PDN GW. Specifically, the S5 connects the S-GW to a P-GW within the home network, while the S8 terminates at the P-GW in a visited network in a roaming scenario. Both can be GTP or Proxy Mobile IPv6 (PMIPv6)-based. In the case of a GTP-based S5/S8, GTP-U sits in the user plane and GTP-C in the control plane. For a PMIP S5/S8, Generic Routing Encapsulation (GRE) is used for IP tunneling of user data and PMIPv6 performs the control plane aspects. Throughout this work, we center our attention on the former case because it significantly simplifies our implementation.

[24] We introduce these aspects of the LTE radio in Section **??**.

[25] It should also be noted that IP-layer traffic is transparent to the eNodeB and is simply forwarded along to the core network.

[26] Relaying in this sense should not be confused with multi-hop radio communications.

6.3.3.4 SGi

SGi defines the connection point between the P-GW and the PDN. The P-GW appears to be essentially a border router to neighboring networks and performs ingress and egress filtering on traffic incident on this IP-based interface.

6.3.3.5 S1-MME

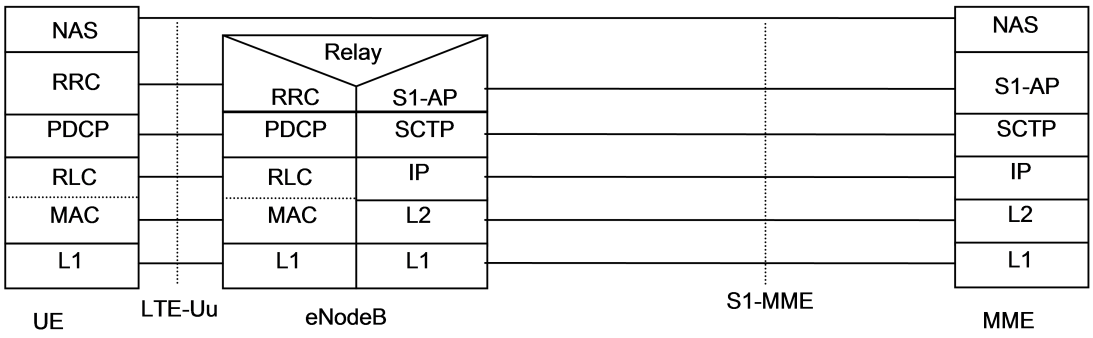


Figure 6.5: S1-MME interface (copied from [68]-Section 5.1.1.3)

The S1-AP protocol, which is responsible for all control plane signaling between the eNodeB and MME, is specific to the S1-MME interface. Stream Control Transmission Protocol (SCTP) provides transport for S1-AP messages. [27]

6.3.3.6 S11

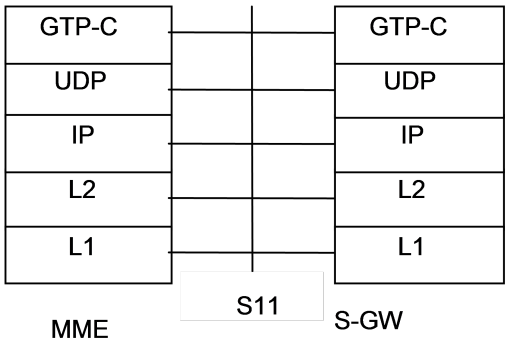


Figure 6.6: S11 interface (copied from [68]-Section 5.1.1.8)

GTP-C tunnels control plane messages between the MME and S-GW and is transported over UDP socket connections.

[27] SCTP is used for transport on many EPS interfaces and is defined in IETF RFC 2960 [71].

Part III
Design and Implementation

7 Simulation Platform Implementation

We now begin our presentation of our design methods and implementation of a parallel, real-time software emulator for LTE/SAE networks, which is composed of three parts. In this section, we discuss this core simulator platform, including the event scheduler, logging, tracing, configuration and other utility classes provided in the ns-3 framework. In Section 8, we describe our approach to building a dedicated compute cluster for running test simulations. In Section 9, we review the EPS model implemented on top of this simulation substrate.

The ns-3 project is a framework for discrete-event simulation of Internet systems [72]. It is the successor to the ns-2 simulator, a tried and tested tool that has been in use by the networking community for over a decade in the design and validation of network protocols [73]. The ns-3 library is a collection of C++ classes that include a variety of modular simulation models, which can be instantiated to build diverse simulated network scenarios. These modules are designed around a simulation *core* layer, which provides the DES *nuts and bolts*, and is typically transparent to the user and not directly manipulated when building basic simulations. Instead, *helper* classes give the user a standard API for accessing core simulator functionality. As we shall explain, it was not enough to simply adopt ns-3 "as-is" in the release version; Many core ns-3 classes required extensive modification in order to suitably match our requirements. Modules and classes from third-party developers were also integrated into the base simulator and modified as needed.

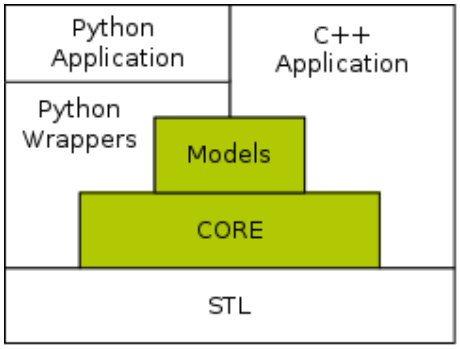


Figure 7.1: Architecture of the ns-3 framework (taken from [72])

Figure 7.1, taken from the ns-3 manual, shows the structure of a typical ns-3 application.[28] Applications must be linked to ns-3 libraries as well as the C++ Standard Template Library (STL). Simulations are generally constructed by instantiating ns-3 objects in the `int main()` function, the program entry point.[29] The C++ source file containing `main` must, at the very minimum, include the header file for one of the core simulator classes given in this section. Individual *.h* files must also be included for any additional model, helper or utility classes.

[28] Python bindings to ns-3 C++ classes are also provided so that simulation scenarios can be coded in Python, however we make no use of this feature in our work.

[29] For practical examples of how ns-3 simulations are constructed, see Section 10.

7.1 Core Simulator

The following classes implement the essential components of a DES network simulator that we described in Section 4. In the standard object-oriented C++ fashion, internal simulator entities, such as the event set, are declared private, and the API used to perform core functions like scheduling events, canceling events, starting the simulation, etc., are public methods. Each of the following classes inherits from the `SimulatorImpl` abstract class, which specifies prototypes of methods common to each core simulator implementation.[30]

As the inner workings of the core DES layer are abstracted out to the model layer, similarly the core layer operates completely independently of any network model. The role of the core layer is to efficiently schedule and process events; It has no knowledge of the nature of such events, nor does it even distinguish that it is simulating a network. Though a fundamental principle of modular software design, which is a hallmark of ns-3 design, this factor causes some difficulty when introducing parallel design patterns. As we shall see, some clever means have been employed to modify the core simulator to incorporate the necessary modifications as unobtrusively as possible.

7.1.1 Default Implementation

The `DefaultSimulatorImpl` class is the basic, single-threaded core implementation and defines only the standard methods declared in `SimulatorImpl`. Not specifying the simulator implementation type will result in this class being used. Some of its more interesting methods are described in Table C.1. Each of these methods have essentially the same behavior across the board.

7.1.2 Distributed Implementation

The full functional and implementational details of the `DistributedSimulatorImpl` class are provided by its authors in [82].[31] This core simulator type allows simulations to be run in a multi-process MPI environment. The number of ranks (i.e. processes) is specified at runtime using *mpirun* (see [86]).

Partitions are defined in the simulation source file (i.e. in `main`) by specifying the *partition ID* of each `Node` or `NodeContainer` object.[32] Each MPI rank then processes events for all nodes within a partition. Furthermore, a network topology can only be partitioned between sets of nodes joined together by point-to-point links (i.e. `PointToPointNetDevice` devices connected by `PointToPointRemoteChannel` links.) Nodes at a simulation boundary communicate with remote nodes by calling `PointToPointNetDevice::Send()`, which in turn calls `MpiInterface::SendPacket()` to perform a non-blocking send of the serialized packet object over MPI. On the remote partition, at the beginning of each iteration of the event processing loop for the respective rank, `MpiInterface::ReceiveMessages()` is called to receive all packets from remote nodes sent during the previous iteration, deserialize said packets, and schedule `PointToPointNetDevic- e::Receive()` events to receive the packets on the appropriate devices. In this way, the `DistributedSimulatorImpl` singleton works in tandem with the `MpiInterface` singleton and `PointTo-`

[30] For the sake of brevity, we only describe some useful API methods provided by each implementation since these classes are already well-documented in the ns-3 Doxygen (see [74]).

[31] Simulations specifying this core simulator type must be compiled with the *–enable-mpi* option (see [72]).

[32] Here the partition ID is equivalent to the rank ID

`PointNetDevice` objects.

The UML activity diagram in Figure B.1 shows the program flow within the `Run()` routine and the main event processing loop.[33] To summarize the procedure, after each event is executed, we must determine if the timestamp of next event in the queue is within the allowed window of time (i.e. less than the current LBTS). If it is, we go ahead and call `ProcessOneEvent()` to process it. Otherwise, we must resynchronize with neighboring partitions by sending and receiving LBTS messages containing the current minimum event TS. The blocking `MPI_Allgather()` routine is invoked to exchange LBTS messages, acting as a barrier and forcing the process to wait until all processes have performed the call. Also at this point, we receive any packets sent from nodes on remote partitions. This process repeats until the *stop* event is encountered or the event set is empty.

7.1.2.1 `MpiInterface` class

This class is used as a singleton (all members are static). Calling `MpiInterface::Enable()` should be done at the start of `main` to ensure the MPI communicator is established before the core simulator attempts to use it. Table C.3 gives some noteworthy public methods provided by the ns-3 MPI API, which in some cases are nothing more than wrapper for MPI routines. The user does not typically use these methods directly, unless one is developing a model that requires MPI. Currently the only model supporting `MpiInterface` is `PointToPointRemoteChannel`, which is not covered in this document.[34]

7.1.3 Multi-Threaded Implementation

The `MultiThreadedSimulatorImpl` class is not part of the release version of ns-3 and is the contribution of a third-party developer, as detailed in [83]. A multithreaded parallel implementation was sought to compare against the message passing-based version, `DistributedSimulatorImpl` to determine if the shared memory model offers any potential performance gain over the distributed model. [35][36] This parallel DES algorithm creates a one-to-one relationship between nodes and partitions, with assigned partition IDs being equivalent to node IDs (i.e. contexts). [37] Each node, then, has its own event set and simulation clock.[38]

This parallel DES algorithm closely follows the basic Chandy-Misra-Bryant algorithm discussed in Section 4.4.5.1 and makes use of the notion of scheduling events

[33] Not shown in the activity diagram are other ranks performing this same procedure symmetrically.

[34] See [74] for the ns-3 Doxygen.

[35] While it may intuitively seem that a shared memory algorithm should perform better than a functionally-equivalent message passing version due to reducing copy operations, a deeper understanding of the MPI implementation is needed to before this assumption can be made. OSU MVAPICH2 1.8 [88], a popular MPI incarnation, does, in fact, use a shared memory *channel* (known as the *Nemesis* channel) for intranode message passing [51]. Thus presents the need to assess an MPI-based core simulator side-by-side with a pthread-based version.

[36] The `MultiThreadedSimulatorImpl` class along with the helper class and example simulations in [83] were originally developed for ns-3 3.9. Some minor modifications were required to integrate it into v3.11, which was the release version at the time this work was conducted. See Section 13 for information on obtaining the source code for this project.

[37] The `MultiThreadingHelper` class handles installing partitions.

[38] We experimented with redesigning the algorithm to support partitioning based on clusters of nodes, in a similar fashion to the distributed algorithm. Considerable debugging was still required at the time this document was composed, so a discussion of this line of investigation is omitted.

with context. We therefore have a separation of events into *local* events, specific to a partition context, and *global* events, which apply to the simulation a whole. Unlike the distributed implementation, partitions are not specified during node creation, but instead the `MultiThreadingHelper` helper class is used to create partitions for each node. The helper also computes the lookahead for each partition by calling `GetMinDelay()` on each link connected to a net device on the respective node.[39] The API for the helper class is given in Table C.4.

The activity diagram in Figure B.2 shows the procedure in `Run()`, which performs some initializations such as creating the thread synchronization barrier with the barrier type specified by the `BarrierType` attribute.[40] [41] A number of worker threads are then spawned, which enter the `DoRunThread()` method for processing events. As shown in the diagram, each iteration of the event processing loop begins by synchronizing partitions and computing the LBTS for individual partitions, as in the distributed model. Global events are then processed by the primary thread (with *thread ID* 0). All threads must then arrive at a barrier before selecting a partition for processing. All events up to the local LBTS are processed, after which threads must resynchronize.

7.1.4 Real Time Implementation

`RealtimeSimulatorImpl` is designed to be used in combination with ns-3 *tap-bridge* feature, as we shall soon introduce, for doing real-time emulation [38]. It employs a `WallClockSynchronizer` singleton for delaying event processing by the appropriate amount of wall-clock time. The synchronizer accomplishes this by calling the `SleepWait()` routine followed by the `SpinWait()` routine. `SleepWait()` calls `pthread_cond_timedwait()` (see [53]) to put the thread to sleep for an amount of time slightly before the real-time deadline.[42] This is to ensure that the sleep operation does not overshoot the deadline due to some delay induced by the kernel scheduler if, for instance, the thread is being preempted. A more precise *wait* operation is achieved with `SpinWait()`, which continuously calls `gettimeofday()` and loops for the remainder of the delay. The only discernible reason for the initial *sleep wait* is to allow a thread to be preempted in order to use the ns-3 emulation feature on a uniprocessor system.[43] As mentioned in Section 5, we wish to control the system scheduler to provide for full utilization of processors by the simulator program. It was subsequently determined experimentally that doing away with the *sleep wait* function and relying entirely on finer-precision *spin wait* improved performance in terms of reducing real-time jitter.[44]

Some attributes are given in Table C.7. `HardLimit` parameter sets the real-time deadline, that is, the maximum allowable jitter. If this jitter is exceeded, the `SynchronizationMode` parameter defines whether the simulation should abort, log the instance, or do nothing.[45]

---

[39] While this core simulator is not completely married to `PointToPointRemoteChannel`, as is the case with `DistributedSimulatorImpl`, all channels must implement the `GetMinDelay()`.

[40] The different barrier types are covered in [83]

[41] Table C.5 enumerates the available attributes that can be set using the ns-3 configuration system.

[42] The precision of `pthread_cond_timedwait()`, like `gettimeofday()`, is actually measured in jiffies or ticks of the system timer interrupt. Most modern architectures and kernel versions support a High Resolution Timer (HRT) with nanosecond resolution. The concept of a jiffy is covered in [54].

[43] The ns-3 tap-bridge feature uses a separate thread for polling the tap device. This is also why some locking primitives are employed in the real-time core implementation for ensuring thread safety.

[44] These ancillary results are omitted for conciseness.

[45] The option *SYNC_SOFT_LIMIT* was added to output a debugging message in the case of a missed

7.1.5 Real Time-Distributed and Multithreaded Implementations

We implement real-time capability for the `DistributedSimulatorImpl` and `MultiThreadedSimulatorImpl` classes simply by incorporating the wall-clock synchronizer functionality into the `ProcessOneEvent()` method for each class. Real-time synchronization can then be enabled with the attribute `IsRealTime` (see Table C.2).

7.2 Tap-Bridge Emulation

The `TapBridge` class is used to integrate "real" network hosts with ns-3 simulations.[46] The basic architecture for using ns-3 as an emulator is shown in Figure 7.2.[47] What is effectively accomplished through this seemingly daunting configuration is a host that has a presence in both the real and simulated world.

7.2.1 Linux Container (LXC) Virtualization

Virtual Machines are used to create multiple *real* Linux hosts on the physical server hosting the simulation. We adopt the Linux Container (LXC) tools package (see [75]) for creating lightweight user-space VMs called *containers*.[48] Each container appears as an independent Linux host with a unique network stack and Ethernet interface (*eth0*, in this case). Interface *eth0* is bridged with a virtual *tap* Ethernet interface, *veth*, on the host OS, which provides an IPC socket through which data can be tunneled from the container to the host machine. The *veth* interface is then bridged with another manually-created tap device to which the corresponding `TapBridge` device in the simulation can attach. [49]

8 Compute Cluster Architecture

Our approach to developing a hardware platform for testing our software is centered around pursuing hardware components that can be procured inexpensively and easily scaled to support larger (i.e. more simulated network elements) and more computationally-intensive simulations. Due to the highly parallel nature of our proposed DES software, the computational power for individual processors is perceived as being of lesser utility than the number of these processing elements. In simpler terms, quantity is preferred over quality. With these design principles in mind, the natural solution presents itself in a *beowulf* cluster, as it is sometimes known, which is a cluster of homogeneous, inexpensive hosts or *compute nodes* networked together for running parallel applications. The original notion of beowulf clusters, as presented in [78], assumes supercomputer-grade hardware is expensive, and involves the use of personal computer hardware for each compute node and LAN technologies such Ethernet for networking. Fortunately,

---

deadline instead of raising an exception, as is the behavior when *SYNC_HARD_LIMIT* is specified as the synchronization mode.

[46] The ns-3 model library documentation (see [72]) covers the basic concepts and usage of the `TapBridge` class. [40] is a thorough tutorial on setting up ns-3 emulation with Linux Containers (LXC), covering the topics in this section in greater detail. [39] demonstrates how ns-3 can be integrated into a practical testbed.

[47] Throughout this section, we refer to the use of the `TapBridge` *UseBridge* mode. See [72] for details on `TapBridge` modes.

[48] It was found that Linux Containers are more ideally suited for our purposes than other standard virtualization products, such as VMWare and OpenVZ, because LXCs do not require separate disk images for each VM instance and require significantly less system resources and setup time.

[49] The *tunctl* package [77] is used to manage TUN/TAP devices. The *brctl* package [76] includes utilities for managing bridges.

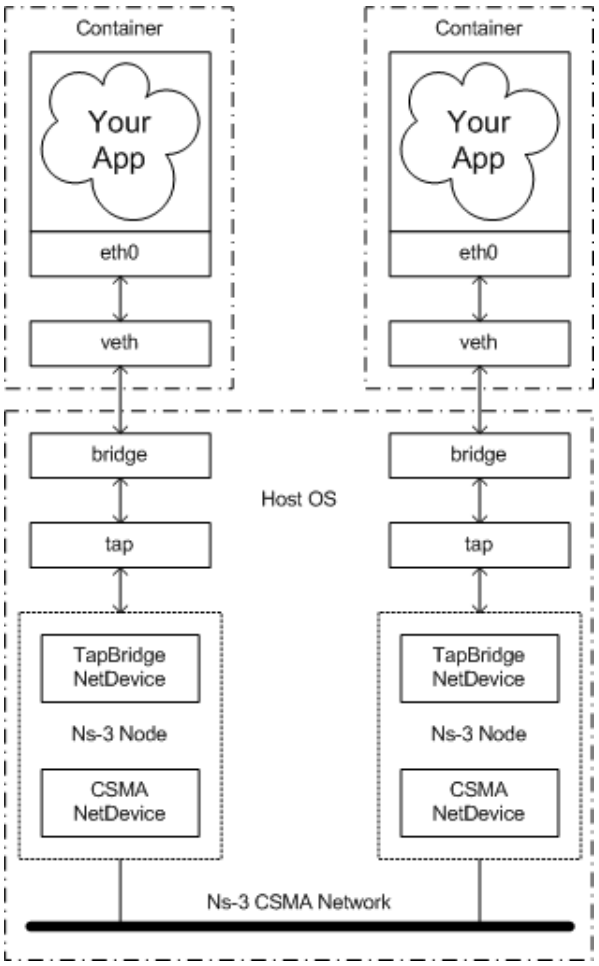


Figure 7.2: Integration between Linux Container VMs and an ns-3 simulation device using tap devices (taken from [40]).

this is no longer the case, as cheap multi-core SMPs are now widely available off-the-shelf along with the same high-speed interconnects found in many large-scale supercomputing clusters.[50]

Figure 8.1 shows the overall architecture of the two-node cluster built for this project. Each of the two identical hosts is based on AMD Opteron 6000-series SMP technology and communicate over 10Gb/s InfiniBand Host Channel Adapters (HCAs) connected to an InfiniBand *fabric* switch. Within each node, the AMD Direct Connect Architecture (DCA) (see [17]) supports two 6-core SMPs sharing a single motherboard in a hybrid UMA-NUMA configuration. Each SMP can uniformly access its own cache and shared DDR memory and has non-uniform access to the shared memory for the other SMP via HyperTransport inter-IC technology. Each SMP also has access to the PCI Express bus for communicating with the HCA.

8.1 InfiniBand Interconnect

InfiniBand is a low-latency, high-data rate network technology often found in commercial data centers and HPC applications [89]. An InfiniBand *switched fabric* takes the form of a *Fat-Tree* or mesh topology and is designed for scalability in that it allows for

[50]Each of the powerful compute nodes built for this project were procured for less than $2000.00 USD. The total price of all networking components amounted to less than $1300.00 USD

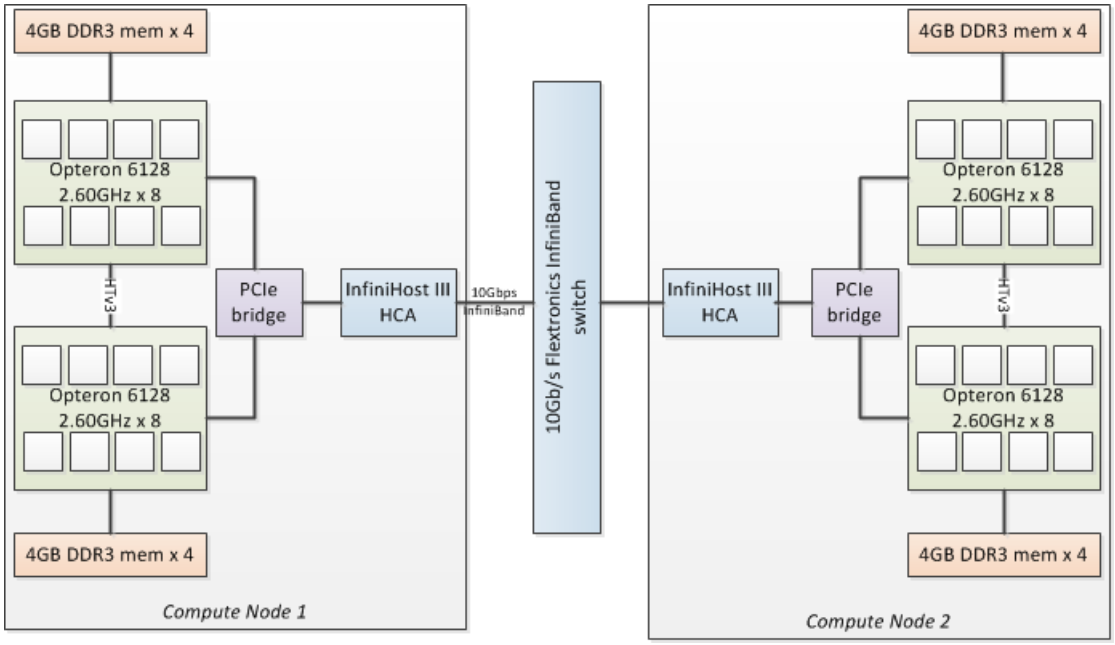


Figure 8.1: Two node Opteron (2x2x8-core) cluster with InfiniBand interconnect

compute nodes to be added to the topology without resulting in increased end-to-end latency. For this project, the fabric consists only of two bi-directional links between each compute node and switch. InfiniBand HCA and switch products offer signaling rates of up to 300Gbps and sub-microsecond end-to-end latencies.[51] [52]

9 LTE/EPS Model Implementation

The core simulator components, operating system and hardware aspects we have thus far considered are undoubtedly critical to achieving our requirements of a real time, parallel-distributed system. Nevertheless these aspects are only supplementary to our primary interest of realistic simulation of Evolved Packet System networks. We now have a solid software emulator foundation on top of which we may begin to build our model of the LTE radio interface and EPC core network entities.

9.1 ns-3 LENA Project Features

Our work is not the first effort to develop such a model, and in attempt to avoid "reinventing the wheel" as much as possible, we have again sought to adopt existing, publicly available, open source software that we may extend to meet our specifications. The ns-3 model developed as part of the LENA (LTE-EPC Network Simulator) project by Ubiquisys and the Centre Tecnològic de Telecomunicacions de Catalunya (CTTC) implements many features that we wish to incorporate into our simulation environment [79]:

- *Comprehensive channel model*: A channel realization using industry standard models provided by the ns-3 Spectrum framework [81].

[51] [90] compares the performance of various InfiniBand and other HPC networking products. With Mellanox InfiniHost III 10Gbps HCAs and an 8-port Flextronic 10Gbps (4x SDA) switch employed by our cluster, sub-$10\mu s$ latencies were demonstrated for RDMA reads and writes of 1KB or less.

[52] For tuning and optimizing MPI over InfiniBand, we employ the OSU Micro-Benchmarks package, which provides some basic example programs for testing the performance of MVAPICH2 MPI routines over Infiniband [87].

- *Basic physical uplink/downlink shared and control channel functions*: The PHY layer provides a basic OFDM scheme allowing data to be multiplexed over time and frequency. Control signaling such as DCI messages and CQI reporting is also performed. SINR is calculated for each subframe to be used in CQI feedback.
- *MAC scheduler and Femto Forum API*: Two types of scheduling algorithms are implemented for dynamic resource allocation. The functionality of the MAC scheduler is accessed through an interface based on the Femto Forum API specification [59]. The MAC layer also performs Adaptive Modulation and Coding and power control based on CQI information.
- *Radio bearer control*: Radio bearers (mapped to logical channels) are assigned a QoS Class Identifier, which prioritize traffic by influencing MAC scheduling decisions.

9.2 Shortcomings of LENA

We give a detailed technical description of the LENA classes in subsequent sections. These classes not only offer many desirable features but comprise a good architectural basis for our EPS model as well. Still these features only make up a subset of the capabilities that we wish to realize in this and future iterations of the project. From taking stock of the components that can be effectively gleaned from the LENA source code, we identify the components that are left to be implemented in order to fulfill the requirements outlined in Section 3.

- *Error model*: While the LENA PHY layer classes do calculate the interference experienced by each received Resource Block, there is no mechanism to determine if blocks were received in error (if, for instance, the MCS of a RB did not correctly match the channel conditions) based on the SINR value, the MCS and Transmission Bandwidth Configuration (i.e. the total bandwidth for a particular RB).
- *Hybrid ARQ*: HARQ with soft combining is fundamental to LTE radio performance. The HARQ model must be effectively tied into the error model as we shall soon explain.
- *Basic RLC modes*: Only the MAC and PHY layers of the LTE radio interface are partially implemented in the LENA model. At the very minimum, the basic functionality of the RLC layer, including segmentation, reassembly and reordering of RLC PDUs, must be realized.
- *Basic PDCP protocol*: At the bare minimum, PDCP sequence numbering should be implemented to be used in combination with RLC Acknowledged Mode, which would be required for simulating any kind of handover scenario.
- *GTP protocol*: Since no components of the core network are offered by the LENA model, we must "start from scratch" when implementing the EPC protocols and network entities. Implementing the GTP-Uv1 protocol is a necessity as is responsible for all user plane data forwarding. Also, for simulating many interesting scenarios such as handovers, the GTP-Cv2 protocol and the S5/S8 interface may also be needed in some form. In many cases we choose to approximate the behavior of such control signaling as we shall see.
- *S1-AP and NAS protocols*: The control-plane functionality of the MME is necessary for coordinating handovers among other processes we are interested in simulating.

9.3 Comments on Implementation Complexity

Production hardware and software systems found in cellular networks are enormously complex, with every aspect fine-tuned for optimal performance. For the purposes of realistic simulation, it is not always necessary or practical to precisely recreate each and every such aspect in the model in great detail. Such would not only be an unnecessary burden from the viewpoint of a programmer but also a waste of computational resources, potentially slowing down execution time of experiments drastically. Throughout the implementation process, we attempt to represent each EPS protocol, process and operation in the simplest possible way that still results accurate "black box" imitation of the behavior of the real system. As an example, for the sake of simplicity much of the control-plane functionality in the LENA model as well as in our modifications and extensions is done over an "ideal" (i.e. unimpaired) channel and is simulated by direct invocation of class methods (often the corresponding protocol on a remote entity). The implementational details of such control signals are introduced layer in this section. Such design decisions that improve the efficiency of simulations while not sacrificing the validity of results are embraced whenever possible. We shall make note of these and other important design considerations when they come up.

9.4 Architecture of EPS Network Elements

The protocol stack representation for each EPS network entities is shown in Figure B.5. Individual classes are discussed in the following sections.

9.4.1 UE Protocol Stack Representation

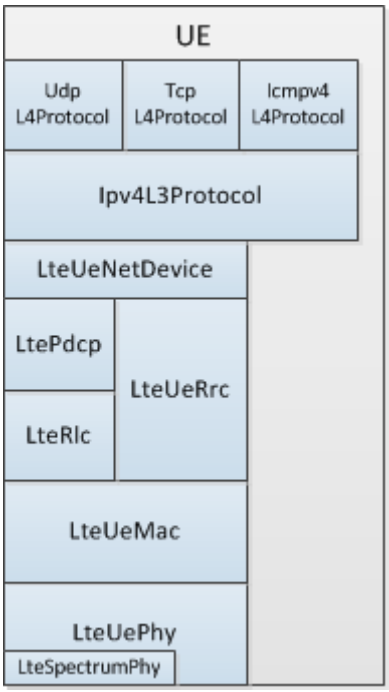


Figure 9.1: ns-3 representations of UE protocols

Figure 9.1 shows the protocol stack for UE nodes. The `LteUeNetDevice` provides the interface to LTE protocols to higher-layer protocols. Classes belonging to the LENA model are detailed in [80] and [79].

9.4.2 eNodeB Protocol Stack Representation

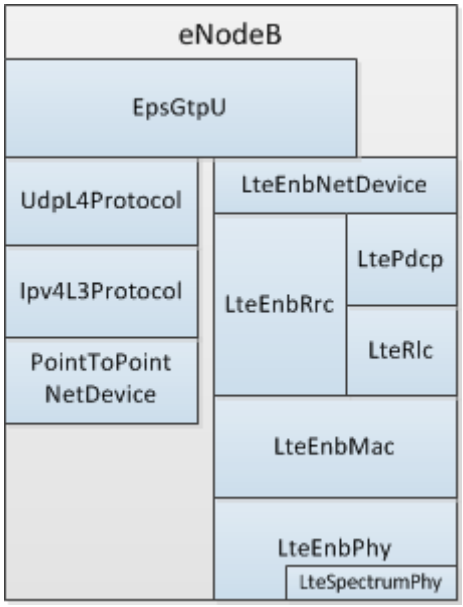


Figure 9.2: ns-3 representations of eNodeB protocols

Here we see the major protocols and their ns-3 class representations for eNodeB network nodes. In the user plane, the eNodeB has the dual roles of providing the air interface (i.e. the E-UTRAN-Uu reference point) for the UE as well as forwarding user data over the S1-U interface. The `EpsGtpU` class bridges the LTE device with the TCP/IP-based PPP device on the S1-U interface.

9.4.3 Gateway Protocol Stack Representation

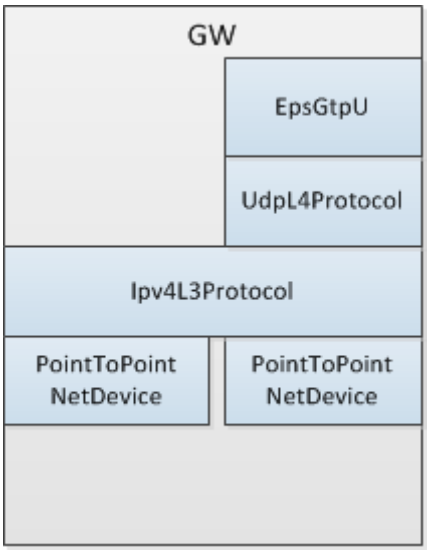


Figure 9.3: ns-3 representations of gateway protocols

We have previously explained the roles of the S-GW and P-GW in the EPS topology. Initially we choose to combine all gateway functionality into a single network entity. This representation does not deviate all that much from reality, as these nodes are often

co-located. A gateway node may have multiple PPP devices on the S1-U interface as well as the SGi interface for networking with external hosts or PDN border routers.

9.5 PHY Layer Model

The LENA PHY model and *Spectrum* channel model are detailed in [79] and [81] respectively. We give descriptions for PHY-layer classes in Table C.10.

9.5.1 Subframe Triggering

The process for building downlink subframes and triggering the UE to send UL subframes is shown in the sequence diagram in Figure B.9.

9.5.2 DCI Messages

The MAC scheduler initiates building of Data Control Indication messages, which communicate DL and UL resource block allocations to the UE. This process is shown in Figure B.10.

9.5.3 CQI Reporting

The procedure for CQI feedback messaging over the ideal control channel is given in Figure B.11.

9.6 MAC Layer Model

The LENA MAC representation is detailed in [79].

9.6.1 MAC Scheduler and Femto Forum MAC Interface

The Femto Forum is an initiative to encourage the standardization of LTE femtocell technology and has released the non-proprietary Femto Application Platform Interface to promote interoperability between different vendors' LTE radio components and software. The FAPI MAC scheduler interface specifies an API for upper-layer protocols to access the scheduler, allowing the functions provided by the scheduler to be abstracted from the scheduling algorithm itself. A set of primitives, the fundamental information elements passed between components, are also defined. Two Service Access Points (SAPs) are provided: the CMAC (Control plane MAC) SAP and the MAC (user plane MAC) SAP. The CMAC SAP is the interface between the RRC layer and MAC control plane function at the eNodeB. It communicates with the CSCHED SAP, the scheduler control interface, by sending _REQ (request) primitives and receiving _IND (indication) and _CNF (configuration) primitives, which are periodically pushed from upper layers. The MAC SAP facilitates communication between RLC instances (at both the eNodeB and UE) and the SCHED SAP in the user plane. Additionally, as we see in Figure B.8, the subframe block is responsible for triggering the scheduler each TTI and receiving the results of scheduling requests [59].

9.7 RRC Layer Model

The details of the LENA RRC model are given in [79]. A description of the relevant classes are given in Table C.12.

9.8 RLC Layer Model

We implement RLC Unacknowledged Mode to perform segmentation and reassembly of RLC PDUs in accordance with the 3GPP specifications in [69]. A description of the relevant classes are given in Table C.12.

9.9 GTP-U Model

See Table C.12 for a description of the relevant classes.

10 Testing and Validation

10.1 Core Simulator Implementation Performance Evaluation

In Section 7, we discussed the types of core simulator implementations at our disposal. To recap, the `MultiThreadedSimulatorImpl` class provides a shared memory, multi-threaded simulation scheme along with real-time synchronization of event processing.[53] In this scheme, there is one-to-one mapping between network nodes and logical processes (i.e. partitions) by default. A specified number of threads can independently process events for separate LPs occurring before the local lower-bound timestamp (LBTS), which is determined after each iteration of event processing. Alternatively, the parallel-distributed scheme used by the `DistributedSimulatorImpl` class involves the mapping of one or more nodes to partitions, which, in turn, are statically assigned to an MPI rank.[54] Each rank processes events for all nodes in its partition up until the LBTS. After each event processing iteration, each LP exchanges null messages over the simulation boundaries, informing neighboring partition of the local LBTS.

Each of these schemes is distinct in its approach to the problems of partitioning and parallel synchronization, however their intent is the same: Speedup. As they both have unique advantages and disadvantages, it is yet to be determined which will offer the best performance or if some measure of speedup is even achievable in certain simulation scenarios given the limitations of conservative synchronization algorithms.[55] In this section, we have devised several experiments to compare the performance of these two schemes and determine the speedup they offer over the default, single-threaded core ns-3 implementation.

10.1.1 Non-Real-Time Performance Comparison

The key metric throughout our evaluation in this and the following sections is time. In the following two basic tests, we measure the time consumed by each simulation using the GNU Linux *time* utility. This command outputs the following values at the end of program execution [84].

- *Real* time, the total wall clock time elapsed during execution.
- *User* CPU time is how long the program spends in user mode. This is the summation of time elapsed by all forked threads and processes while actually running the program and does not include time spent while blocked.
- *System* CPU time is the total time spent in kernel mode while executing system calls on behalf of the program. This is also summed over all execution paths, since multiple threads or processes may be accumulating CPU time waiting for system calls to return.

Using this simple and useful tool, we can not only determine the runtime of a simulation program but also get a rough idea of how long the program spends being blocked while waiting at a barrier or performing some other system call.

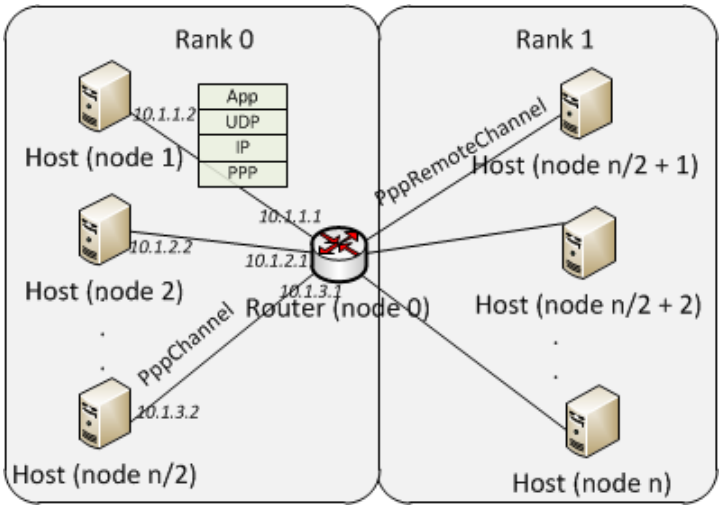


Figure 10.1: Distributed star topology test case with simple traffic generation

10.1.1.1 Distributed Implementation - Bottleneck Test

This experiment is designed to test a very practical situation where there is an uneven distribution of events that must be processed by certain logical processes. In real world networks, some sections (partitions) of the topology may infrequently communicate with each other or may even be effectively disjoint in that sense. As we shall examine in the later simulation scenarios, such topologies lend well to parallelization. Here we examine the opposite case where there is no clear-cut segregating of traffic. Furthermore, as is often the case in hierarchical and partial mesh topologies, core routers require greater throughput capacity than edge nodes in order to avoid creating a bottleneck for network traffic. With our simple techniques for LP partitioning, which can only be done with the granularity of a node with the aforementioned implementations, we cannot allocate more than one thread or process to handle events for an LP or node. Therefore we cannot increase throughput for event processing for some LPs over others. We propose several solutions to this fundamental problem in parallel network simulation in Section 12. For now, within the capabilities of our partitioning regime, we would like to demonstrate the speedup (or lack thereof) possible when one node poses an "event bottleneck" in the simulation.

Figure 10.1 shows the star topology in our ns-3 program, which consists of a hub router and n spoke nodes. Each spoke is a host running an instance of `UdpEchoClient`, which periodically sends UDP data to the hub node. The hub, running an instance of `UdpEchoServer`, echos back the reply to the sending client. We use these basic classes to generate dummy traffic.[56]

10.1.1.1.1 Simulation Configuration

Our simulation program source file, *rt-distributed.cc*, contains the `int main()` definition that is the program entry point from which all simulation objects are instantiated.

[53] See Section 7.1.3.

[54] See Section 7.1.5.

[55] See Section 4.4 for the relevant background.

[56] It should be noted that our methods for traffic generation do not conform to any standard traffic models and, as in this scenario, are designed to induce high overall load on the network.

To better acquaint the reader with how simulations are configured and initialized, we dissect each of the components as follows. For the sake of conciseness, we shall refrain from going into so much detail for subsequent experiments. The first code we find in the `main` routine is the following.

```
uint32_t nSpokes = 2;
double stopTime = 10.0;
bool isRealtime = false;

CommandLine cmd;
cmd.AddValue ("nSpokes", "Number of nodes in the star", nSpokes);
cmd.AddValue ("stopTime", "Simulation duration", stopTime);
cmd.AddValue ("isRealtime", "Enable real time syncing", isRealtime);
cmd.Parse (argc, argv);
```

Here we parse some parameters passed in from the command line, which are fairly self-explanatory. We can turn real-time synchronization on or off with the *IsRealTime* attribute. Next we set the simulator implementation type.

```
MpiInterface::Enable (&argc, &argv);
GlobalValue::Bind ("SimulatorImplementationType",
                                StringValue ("ns3::DistributedSimulatorImpl"));
Config::SetDefault ("ns3::RtDistributedSimulatorImpl::IsRealTime",
                                BooleanValue (isRealtime));
uint32_t systemId = MpiInterface::GetSystemId ();
uint32_t systemCount = MpiInterface::GetSize ();
```

Here, also, we call `MpiInterface::Enable ()` to initialize the MPI communicator. The call to `GetSystemId ()` returns the rank ID, while `GetSize ()` returns the communicator size determined at runtime.

```
NodeContainer nodes;
for (uint32_t i = 0; i <= nSpokes; i++)
        {
                 uint32_t myRank = i % systemCount;
                Ptr<Node> p = CreateObject<Node> (myRank);
                nodes.Add(p);
        }
```

We create $nSpokes$ spoke nodes plus one hub node and evenly distribute them over the ranks. The hub node (node 0) is assigned to rank 0. Next use `PointToPointHelper` and `PointToPointStarHelper` to aggregate one `PointToPointNetDevice` to each spoke and $nSpokes$ devices to the hub and link them together. Nodes assigned to rank 0 are connected to the hub with a `PointToPointChannel`, while those on remote ranks use `PointToPointRemoteChannel`.

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("100Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("1ms"));
PointToPointStarHelper star (nodes, pointToPoint);
```

The *DataRate* and *Delay* channel parameters ultimately determine the delay between scheduling TX and RX events. We then use the star helper class to aggregate the TCP/IP stack to each node and assign IPv4 addresses.

```
InternetStackHelper internet;
star.InstallStack (internet);
star.AssignIpv4Addresses (Ipv4AddressHelper ("10.1.1.0", "255.255.255.0"));
```

Addresses are assigned in as depicted in Figure 10.1. We then create an instance of `UdpEchoServer` (listening on port 9) on the hub along with one `UdpEchoClient` instance for each spoke. Client instances are set to start 10 ms apart and send 1KB-sized packets every 100 ms to the hub in order to create a somewhat varied traffic pattern.

```
uint16_t port = 9;
UdpEchoServerHelper server (port);
ApplicationContainer apps = server.Install (star.GetHub ());
apps.Start (Seconds (0.01));
apps.Stop (Seconds (60.0));

for (uint32_t i = 0; i < star.SpokeCount (); ++i)
        {
                uint32_t packetSize = 1024;
                uint32_t maxPacketCount = 100;
                Time interPacketInterval = MilliSeconds (100);
                UdpEchoClientHelper client (star.GetHubIpv4Address (0), port);
                client.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
                client.SetAttribute ("Interval", TimeValue (interPacketInterval));
                client.SetAttribute ("PacketSize", UintegerValue (packetSize));
                apps = client.Install (star.GetSpokeNode (i));
                apps.Start (Seconds (1.0 + (.01*i)));
                apps.Stop (Seconds (60.0));
        }
```

Finally, we run the simulation for the given time.

```
Simulator::Stop(Seconds(stopTime));
Simulator::Run ();
stream->WriteToFile ();
Simulator::Destroy ();
MpiInterface::Disable ();
```

After the simulation completes and *Run ()* returns, we dump the contents of the trace stream to the disk and clean a few things up. This program is compiled using the *waf* build environment.

./waf -d optimized configure –enable-sudo –enable-examples –enable-mpi

The important options here are *–enable-mpi*, which sets the *NS3_MPI* directive, and *-d optimized*, which enables full compiler optimizations.[57]

10.1.1.1.2 Simulation Execution

Before running the simulation, we use *./waf shell* to set the requisite ns-3 environmental variables.[58] We can now execute the program using the *mpirun* utility.

/usr/bin/time -p mpirun -np 2 -H ltesim1 ./rt-distributed –nSpokes=32 –stopTime=10 –isRealtime=0

The first token is the *time* command. The *-p* option sets the output format to display time in seconds. The *mpirun* utility is used to execute parallel jobs with MPI. The *-np* option specifies the number of processes (ranks) and *-H* is used in place of a host file to indicate the list of hosts on which to run them.[59] We measure elapsed real, user-space

[57] Equivalent to *g++ -O3* [85]

[58] See [72].

[59] The *mpirun* utility is included with OpenMPI 1.4.3 compiled on CentOS 5.6. See [86] for usage.

and kernel-space time for simulations of size $nSpokes = \{128, 256, 1024\}$ nodes with rank assignments evenly distributed to $np = \{1, 2, 4, 8, 16\}$ processes. That is, for the combination of $nSpokes = 128$ and $np = 8$, each rank has 16 nodes (LPs) to tend to. We initially limit our experiment to the intranode case, where all MPI ranks are confined to a single SMP system. As discussed in Section 8, our test cluster employs dual 8-core SMP compute nodes (16-cores per cluster node), so we should be safe from excessive context switching and process movement as long as we maintain a one-to-one affinity between processors and ranks.[60]

10.1.1.1.3 Analysis of Debugger Output

In order to confirm that our simulation behaves in the expected way, we enable the following log messages to debug certain components of interest.[61]

```
1 LogComponentEnable("MpiInterface", LOG_LEVEL_ALL);
2 LogComponentEnable("Ipv4L3Protocol", LOG_LEVEL_ALL);
3 LogComponentEnable("UdpEchoClient", LOG_LEVEL_ALL);
4 LogComponentEnable("UdpEchoServer", LOG_LEVEL_ALL);
```

The `MpiInterface` class wraps all calls to MPI routines, so by enabling full debug messages we can see whenever data packets and null messages are being sent over the simulation boundary (i.e. between LPs, individual nodes in this case, on different ranks). Logging `Ipv4L3Protocol` calls allows us to see the IP header information whenever packets are TXed or RXed at the IP layer, and finally we can enable logging when packets arrive at the application with the latter two statements. Below is a sample of the output from running *rt-distributed* in debug mode with $nSpokes = 2$ and $np = 2$. Lines 1-20 show the function call chain at node 1 from the application layer client down to the link layer interface and then to the channel itself. We see the IP forwarding table lookup in lines 4-5 and the IP header built in line 7. Then the PPP header is added and the link layer frame is sent to node 0 by calling `MpiInterface::SendPacket()`, which in turn calls the non-blocking *MPI_Isend* routine. Lines 22-30 log the reception of the packet at each layer on node 0. Though these messages only feature a small subset of simulation components, it is enough to demonstrate the correct sequence of events.

```
1  UdpEchoClientApplication:ScheduleTransmit()
2  UdpEchoClientApplication:Send()
3  Ipv4L3Protocol:Send(0x1f22ed0, 0x1f2df50, 10.1.1.2, 10.1.1.1, 17, 0x1f2df30)
4  Testing address 127.0.0.1 with mask 255.0.0.0
5  Testing address 10.1.1.2 with mask 255.255.255.0
6  Ipv4L3Protocol::Send case 3:  passed in with route
7  Ipv4L3Protocol:BuildHeader(0x1f22ed0, 10.1.1.2, 10.1.1.1, 17, 1032, 64, 1)
8  Ipv4L3Protocol:SendRealOut(0x1f22ed0, 0x1f2dfc0, 0x7fffff61c1d0)
9  Send via NetDevice ifIndex 0 ipv4InterfaceIndex 1
10 PointToPointNetDevice:GetMtu()
11 Send to destination 10.1.1.1
12 PointToPointNetDevice:Send()
13 p=0x1f2dfc0, dest=ff:ff:ff:ff:ff:ff
14 UID is 4294967296
15 PointToPointNetDevice:AddHeader()
16 PointToPointNetDevice:TransmitStart(0x1f1d7a0, 0x1f2dfc0)
17 UID is 4294967296)
18 Schedule TransmitCompleteEvent in 0.0016864sec
19 MpiInterface::SendPacket(): Node 1 on rank 1 sending packet to 0 at +1003686400.0ns
```

[60] We discuss CPU affinity in Section 8.

[61] The ns-3 logging system is only available when programs are compiled in debugging mode (i.e. *waf -d debug configure*).

```
Sent 1024 bytes to 10.1.1.1
PointToPointNetDevice:TransmitComplete()
PointToPointNetDevice:Receive(0x26d2590, 0x26e3b00)
Ipv4L3Protocol:Receive(0x26d4070, 0x7fff259688a0, 0x26e3b00, 2048,
                                                  02-06-00:00:00:00:00:02)
Packet from 02-06-00:00:00:00:00:02 received on node 0 of size 1052
For me (destination 10.1.1.1 match)
Ipv4L3Protocol:LocalDeliver(0x26d4070, 0x26e41c0, 0x7fff25968540)
Received 1024 bytes from 10.1.1.2
```

10.1.1.2 Analysis of Simulation Runtime Results

For each combination of number of spoke nodes ($nSpokes$) and number of ranks (np), we average the three time metrics from the *time* utility over two trials. The results from these trials are tabulated in Table C.13 and graphed per each value of np in Figures B.12 through B.14. The total user and kernel mode times are shown along with the average (normalized) per-process times. We observe that for the 128-node case, real time decreased only slightly from 1.11s in the uniprocess trial to 1.03s in the 2-process trial. While average per-process user time decreased by 80%, with the total user time decreasing almost threefold from the 1-process case, the average per-process system time increased by 71%.[62] The increase in system time makes sense as it can be attributed to two factors that, as we shall find out, are particularly debilitating from the perspective of efficient network simulation. There is an unavoidable, though so small as to be negligible, delay incurred by the non-blocking MPI send and receive for interprocess communication. This is not of concern. The real issue is with synchronization. After each event processing iteration, the MPI rank issues a call to `MPI_Allgather` to exchange LBTS messages with other ranks. As a blocking routine, this call is, in effect, a barrier. Thus the first process to arrive must wait for all $np - 1$ other processes to finish their iteration before resynchronization can occur.[63] While we have not devised a test to directly measure the amount of time spent waiting at this barrier, the only conclusion that can be drawn from the data is that this is a significant bottleneck in performance. It is a well-known issue in discrete event network simulation that such synchronization is the foremost limiting factor in parallel simulation. When the delays (ultimately, the lookahead) between neighboring nodes on separate logical processes is small, the speedup and the degree to which actual work can be parallelized may be overshadowed by the increase in communication and synchronization time (i.e. time spent waiting at barriers).[64]

[62] Quite anomalously, the total user time elapsed by both processes decreased from 0.9s to .35s, which would indicate a computational speedup of over 2 (an impossibility, in principle, with only two processes). It is apparent that this cannot be the case, however it is unknown what factors influence the user mode time in this way and further investigation is required. At present, due to the complex interactions between simulator objects during user-mode execution along with the frequent use of system calls, such as memory allocations, throughout all code paths, it would be difficult to directly analyze exactly how much time is spent in user mode versus kernel mode for certain sections of code by, for instance, inserting calls to the glibc `clock` function. This may also reflect our lack of understanding of how *time* computes these metrics.

[63] See Section 7.1.2 for a discussion of the distributed synchronization algorithm.

[64] In our simulation, we set the propagation delay of all PPP connections to 1ms. Just doing some back-of-the-napkin calculation, we know that it would take at take around 80μs to transmit a full 1024-byte packet over 100Mbps connection. Other factors also impact the total packet delay which are not so trivial to model such as propagation delay, processing of checksums and forwarding and queuing time for routers. With the incorporation of all these factors, 1ms seems a reasonable estimate for our purposes here. Our hand-waving of some these aspects could result in inaccurate and unrealistic results, but our

We see this detrimental effect to an even greater degree as we increase the number of processes. For the 128-node test, we have a near doubling of the total real time and, while the normalized user time decreases from the 1-process case, the system time shoots up dramatically. We continue to see this trend for cases with additional processes, with an 50% increase in total runtime between the 1 and 16-process case. This is, of course, the opposite effect of what we had hoped to achieve and, while slightly disappointing, it is not at all surprising. This test is designed to show that it does not make sense to parallelize over nodes that are highly interdependent in terms of communication. For the 256 and 1024-node trials, we see even less overall speedup between the 1 and 2-process case and, again, an increase in runtime for additional processes. While these results suggest that there may be a fundamental shortcoming with our synchronization algorithm, it should be noted that this rough benchmark does not directly tell anything about the execution time for simulation events (the measure we are trying to improve). In subsequent tests, we will look closely at traces of event timestamps to give us a better understanding of the impact of each core simulator implementation on real-time event processing performance.

10.1.1.3 Distributed Implementation - Embarrassingly Parallel Test

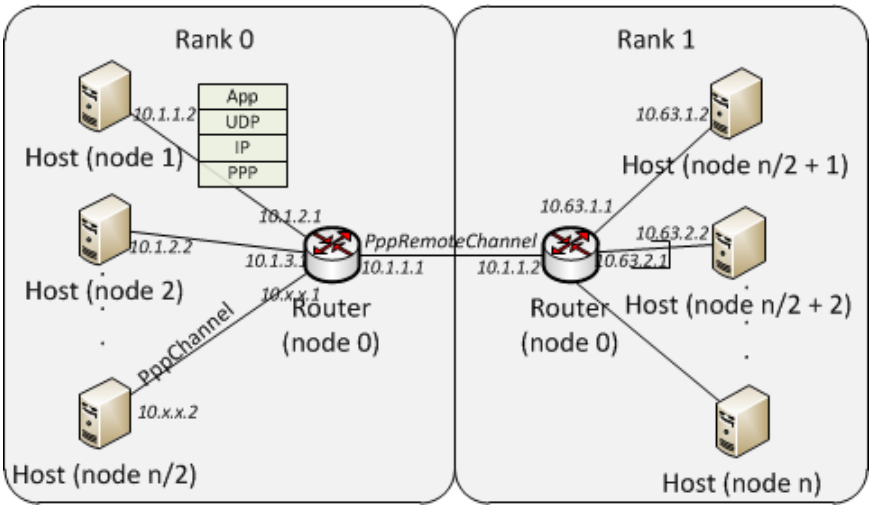


Figure 10.2: Distributed dumbbell topology test case simple traffic generation

This test is contrived to minimize communication between logical processes in attempt to demonstrate a best-case situation where events scheduled on two neighboring LPs can be processed as independent from one another as possible. This clear-cut apportioning of work to tasks with minimal communication between the tasks is known as an *embarrassingly parallel* algorithm. In this situation, some degree of communication and synchronization is still required, since even disjoint LPs must exchange LBTS messages to synchronize. In the figure, we see a dumbbell topology of two point-to-point star topologies with hub routers connected by an additional PPP link. As in the previous experiment, spoke nodes only communicate with the hub node for their local star. No data packets are exchanged over the router link at all.

estimate is acceptable for the purpose in this series of trials of testing the limits our algorithm.

10.1.1.3.1 Simulation Configuration

Our simulation configuration is simply a mirroring of the single star topology, with the addition of a `PointToPointRemoteChannel` connecting the two stars. The number of spoke nodes given at the command line is divided evenly between the two stars. We separate nodes into two `NodeContainers`, `leftNodes` and `rightNodes`, and create the link as follows.

```
1 NetDeviceContainer routerDevs = pointToPoint.Install(rightNodes.Get (0),
2                                                      leftNodes.Get(0));
```

The node with index 0 from each container is the hub node for that star. Left and right nodes are statically assigned to MPI rank 0 and 1, respectively. Addresses are assigned to each `routerDevs` as shown in Figure 10.2.

10.1.1.3.2 Simulation Execution

Our distributed dumbbell network is represented in the *rt-distributed2* program. As before, we execute this program with the time utility for $nSpokes = \{128, 256, 1024\}$ for 1 and 2 processes. We average the output from time over two trials.

/usr/bin/time -p mpirun -np 2 -H ltesim1 ./rt-distributed2 –nSpokes=128 –stopTime=10 –isRealtime=0

10.1.1.3.3 Analysis of Debugger Output

We again run the program in debug mode to confirm the correct behavior. We observe that no MPI messages are logged beside synchronization null messages.

10.1.1.3.4 Analysis of Simulation Runtime Results

As in the previous experiment, we take the average of two trials for each measure of time for simulations of size 128, 256 and 1024 nodes. The numeric results from these trials are tabulated in Table C.14 and plotted in Figure B.15, B.16 and B.17. The results expose once for all the degree to which synchronization cost is impairing the ability to effectively parallelize the simulation. For the 128-node case, we see an 172% increase in total runtime. Although user-mode time for all processes was nearly halved, the system time increased 7-fold. In the 256-node case, we see a 74% increase in runtime, with a similar trend in user and system time. We do see a slight improvement in runtime for the large-scale, 1024-node test, with a speedup of just over a 2%. This suggests that there is a point in scaling up the number of nodes per each partition at which the gain from parallelizing computation exceeds the added cost of synchronization.[65] Again, we attribute this dramatic increase in system time to time waiting for the `MPI_Allgather` call to return. Even though there are no events being scheduled across simulation boundaries, and each LP could safely process events without any need for synchronization at all (theoretically resulting in a speedup approaching 2), the simplicity of the algorithm does not take this factor into consideration. Therefore,

[65] We do not include results for larger size simulations because, as we shall see in later experiments, the real-time jitter starts to become excessive with partitions of 128 nodes or greater, despite any potential speedup from parallelization.

null messages must still be exchanged periodically across the boundary, and that means waiting at the blocking MPI call.

These results confirm the conclusion of the authors of the `DistributedSimulatorImpl` class in [82] that, when lookahead becomes small, standard conservative techniques can no longer function effectively without some additional optimizations such as event bundling and other methods discussed in Section 12.[66]. While the results are discouraging, they have brought us closer to understanding the real challenges of DES for practical networks and helped to identify the bottlenecks in our algorithm.

10.1.1.4 Multithreaded Implementation - Bottleneck Test

We now move on to applying the same simple approach of runtime analysis in order to do some initial benchmarking of the `MultiThreadedSimulatorImpl` POSIX thread-based simulator implementation. While this algorithm only presents the possibility of intranode parallelism in its present form, we have previously discussed how it could be combined with the distributed model implementation to create a hybrid parallel programming model.[67] For now, we evaluate its intranode performance with the *time* utility for the star topology depicted in Figure 10.1. This is the same bottleneck scenario as in Section 10.1.1.1, although the behavior of the algorithm is fundamentally different.

10.1.1.4.1 Simulation Configuration

The simulation program code is similar to that of the distributed bottleneck test with several key differences. The parameters for initializing the core simulator object are passed in as follows.

```
1  CommandLine cmd;
2  cmd.AddValue ("nThreads", "", nThreads);
3  cmd.AddValue ("stopTime", "", stopTime);
4  cmd.AddValue ("isRealtime", "", isRealtime);
5  cmd.Parse (argc, argv);
6
7  Config::SetDefault ("ns3::MultiThreadedSimulatorImpl::ThreadsCount",
8                                          UintegerValue (nThreads));
9  Config::SetDefault ("ns3::MultiThreadedSimulatorImpl::IsRealTime",
10                                         BooleanValue (isRealtime));
11
12 MultiThreadingHelper multiThreadingHelper;
13 multiThreadingHelper.Enable ();
```

With these command line arguments we can enable or disable real time synchronization (we disable it for this test), specify the number of threads along with the simulation duration. We then create a `MultiThreadingHelper` object and initialize it by calling `Enable ()`. The remaining code is identical to *rt-distributed* in Section 10.1.1.1 up until the these statements.

```
1  multiThreadingHelper.Install ();
2  Simulator::Stop(Seconds(stopTime));
3  Simulator::Run ();
4  Simulator::Destroy ();
```

[66] We refer to the concept of *horizontal parallelization in Section 4*

[67] See Section 4.4.3.

We must call `MultiThreadingHelper::Install ()` after node and device creation. This assigns nodes to partitions (a one-to-one association) and sets the lookahead per each partition to the minimum channel delay for all links incident on the node, which is uniformly 1ms for each PPP link in the star topology.

10.1.1.4.2 Simulation Execution

We execute the multi-threaded program for combinations of $nSpokes = \{128, 256, 1024\}$ and -$nThreads = \{1, 2, 4, 8, 16\}$ and observe the output from the *time* utility.

/usr/bin/time -p ./mt-star –nSpokes=2 –stopTime=10.0

10.1.1.4.3 Analysis of Debugger Output

We temporarily enable debugging of the `MultiThreadedSimulator` object with this logging statement at the beginning of the `main` routine.

```
1 LogComponentEnable("MultiThreadedSimulatorImpl", LOG_LEVEL_DEBUG);
```

This allows us to see the dynamic selection of partitions by threads.

```
1 Partition 3 NextPartitionEventTs 0 minMessageTs 0
2 Partition 3 being processed by thread 0
3 Processing events at partition 3 going from 0 until 2000000
4 Partition 2 NextPartitionEventTs 0 minMessageTs 0
5 Partition 2 being processed by thread 1
6 Processing events at partition 2 going from 0 until 2000000
7 Thread 0 processed 4 events at current iteration.
8 Thread 1 processed 4 events at current iteration.
```

This snippet of log output shows the parallel execution of events for partitions 3 and 2 by threads 0 and 1, respectively, confirming the expected behavior of the algorithm. We also observe that the LBTS for both partitions for the shown iteration is 2 ms.

10.1.1.4.4 Analysis of Simulation Runtime Results

We measure the real, user and system time for combinations of 128, 256, and 1024-nodes with 1, 2, 4, 8 and 16 worker threads and average the results of two trials. The data collected from these trials are given in Table C.15 and displayed in Figures B.18 through B.20. We observe the total runtime for the 128 and 256 node case increases by a small degree for each number of threads. For these trials, both the user and system time increase significantly with the number of threads for all cases except for when $nThreads = 2$, for which the system time actually decreases from 0.05 s to 0.02 s. The increasing system time can again be blamed on the pthread barrier `wait ()` that all threads must reach after each event processing iteration. As observed in [83], the cost of this operation increases with the number of threads, which is to be expected. A possible explanation for the rise in user-mode time is the added complexity of the algorithm compared to the MPI-based implementation. Once more, for a large number of nodes, we find that our algorithm does produce a small speedup, with a decrease in runtime from 11.86 to 11.16 between the single-threaded and 8-thread tests. This minute improvement reflects the results found in [83] for a topology of 1024 nodes.

The results demonstrate that this algorithm is also deficient in providing the desired speedup, synchronization once again being the unavoidable crippling factor. There are some obvious design flaws that present themselves, the most obvious of which is the inability to assign more than one node to a partition, resulting in a large number of null messages that must exchanged and frequent resynchronization. We suggest some proposed amendments to the algorithm in Section 12.

10.1.2 Real-Time, Distributed Implementation Test

In the non-real-time tests, we made use of the *time* utility to give us an rough idea of the performance of our parallel algorithms. We have so far disabled real-time event synchronization, leaving the simulator to process events as fast as possible. For the remaining experiments discussed in this in the following sections, we approach measuring algorithm performance when real-time mode is enabled in terms of the *jitter*, the difference between an event's scheduled simulation time and the wall-clock time at which it completes.

$$jitter = |t_{wall} - t_{sim}| \tag{10.1}$$

Logging of jitter is performed using the ns-3 trace system.[68] [69] The particular *trace source* that we make use of is `PointToPointNetDevice` ***MacRx*** source that is invoked at the end of a `PointToPointNetDevice::Receive ()` event. This gives us the exact time (in nanoseconds) elapsed during event processing for the `Receive ()` event, at the conclusion of which the PPP packet is decapsulated and passed up to the IP layer.

10.1.2.0.5 Simulation Configuration

For this experiment, we repeat essentially same procedure used in the non-real-time test. At the end of our simulation source file *rt-distributed.cc* but before `Simulator::Run ()` is called, we include the following source.

```
std::string filename;
if(systemId == 0)
        {
                filename = "rt-distributed-0.tr";
        }
else
        {
                filename = "rt-distributed-all.tr";
        }
AsciiTraceHelper ascii;
Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream (filename);
pointToPoint.EnableAsciiAll (stream);
```

In order for us to more easily distinguish between intra- and inter-rank events, we separate trace output into two files as shown. We then run the simulation for the given time. When `Run ()` returns, we do the following.

```
stream->WriteToFile ();
```

`OutputStreamWrapper::WriteToFile ()` dumps the trace output to the files given above.

[68] See [72]

[69] Modifications to the `OutputStreamWrapper` class were made so that traces remain in memory and are not written to the disk until the simulation completes. Frequent costly disk I/O operations would have a detrimental effect on achieving real-time performance, so this is seen as a simple way avoid them.

10.1.2.0.6 Simulation Execution

In addition to observing the results of distributing the simulation over more processes, we wish to see what added latency is induced in the inter-node case as compared to the intra-node case. We run the simulation for combinations of $nSpokes = 16, 32, 128$ and $np = 1, 2, 14, 16, 32$ processes (with real-time mode enabled) using this command.[70]

mpirun -np 2 -H ltesim1,ltesim2 –nSpokes=2 –realtime=1 –distributed=1 –stopTime=10.0

In this example, we have two processes, one assigned to host *ltesim1* and the other to *ltesim2*. We run the with all processes confined to a single compute node by specifying only *ltesim1* in the host list.

10.1.2.0.7 Analysis of Real-Time Jitter Results

Table C.16, C.18 and C.19 give the maximum, minimum, and mean and standard deviation for the jitter over two trials of each intra-node test case for 16, 32 and 128-node topologies. For comparison, Table C.17 gives the inter-node results for 16 nodes.[71] We separate the data for the hub node (*node 0*) and spoke nodes to observe the bottlenecking effect at its source.

For the 16-node, uni-process control case, we observe an average jitter of $1.47\mu s$, with the maximum recorded jitter of $7.10\mu s$ for the hub node. As we shall discuss in Section IV, this is a very good basis as it is well beneath our real-time event deadline of 1 ms. As we previously saw in the total runtime for the non-real time distributed bottleneck test, we see a slight improvement in performance over the uni-process case in the 2-process test, with the maximum jitter being reduced by several microseconds. We find that the added inter-node latency does not have a significant impact on average jitter for the 2-process test, as only a few microseconds are added to the average. The maximum jitter, however, increases more than 10-fold to $68.6\mu s$, still well within the desired 1 ms. We see for increased numbers of processes a gradual increase in the average and maximum jitter, which agrees with the data from the non-real-time tests.

We see that with a 32-node topology, the average and maximum jitter for the hub node is actually less than it is for the 16-node tests. While the difference is minute (only a few hundred nanoseconds), it is admittedly counterintuitive. For now, the only explanation that presents itself is that the altered traffic pattern is actually more efficiently processed by the algorithm, even though there is a larger volume of events. We also observe that the 2-process performs worse than the uni-process case, unlike the 16-node topology.

10.2 LTE Model Test Cases

We now begin our evaluation of the LTE/SAE network model previously introduced. Three scenarios are contrived in order to produce data that is meaningful in analyzing the performance of the ns-3 model in terms of real-time latencies or jitter. We begin by ”stress testing” a simulated E-UTRAN network, consisting of a single eNodeB and multiple UEs, as we collect statistics on the real-time jitter. We then show

[70] We choose these values for $nSpokes$ somewhat arbitrarily but with the intention of providing data that shows the achievable performance for a small number of nodes as well as a moderately-sized topology that still produces an acceptable amount of jitter of less than 1 ms.

[71] We also give the difference in max, min, mean and std between each of two identical trial.

combine the simulated E-UTRAN with a simple core network consisting of a gateway, which tunnels traffic between the eNodeB and a host in the external PDN. The same `UdpEchoServer` traffic generation scheme is used for this experiment. Finally, we demonstrate a scenario involving an emulated PDN host and mobile, with real HTTP traffic transmitted over a simplified EPS network.

10.2.1 RAN Saturation Test Case

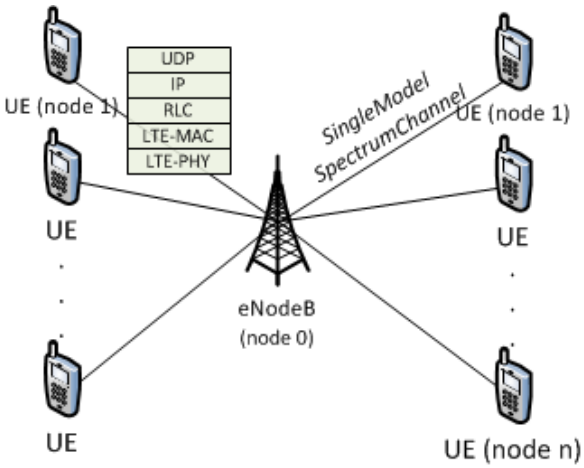


Figure 10.3: LTE radio network saturation test case (DL data transmitted in all resource elements)

This scenario is devised to observe the resulting jitter statistics when a simple LTE E-UTRAN model is under maximum load, which is effectively achieved by transmitting dummy data in every available Resource Element (that is, across all subcarriers for each subframe).[72] We measure the wall-clock time elapsed between when `LteSpectrumPhy::EndRx ()` events are scheduled and when the decapsulated RLC SDU is passed up to the IP layer.[73] The *RxPDU* trace is invoked at the end of the `LteRlcUm::DoReceivePdu ()` method, which records the desired jitter.

As of yet, neither of the parallel DES algorithms we introduced earlier allow partitioning of wireless networks.[74] For this test case, we therefore evaluate the single-threaded performance of our simulated E-UTRAN. We later combine this RAN model with our core network model, which can be partitioned across PPP links.

10.2.1.1 Simulation Configuration

The file *lte-network-saturation.cc* contains the `int main ()` function that defines our simulation scenario. In `main ()`, we first parse in values from the command line for the number of mobiles and the simulation duration.

```
1 uint32_t numUe = 2;
2 double stopTime = 10.0;  // seconds
```

[72] RLC *saturation mode* (RLC_SM, see Section 9) is configured with the *RlcMode* attribute of the RRC protocol (`LteEnbRrc`) at the eNodeB. For each UE, the default RLC_UM mode (set with the *RlcMode* attribute for `LteUeRrc`) is used to receive downlink PDUs.

[73] Recall that the PDCP protocol is not yet implemented, so we pass packets directly between the RLC and IP layer.

[74] We shall address why designing a parallel simulator for wireless networks is problematic in Section IV.

```
3
4   CommandLine cmd;
5   cmd.AddValue ("numUe", "", numUe);
6   cmd.AddValue ("stopTime", "", stopTime);
7   cmd.Parse (argc, argv
```

We set the simulation type to the `RealtimeSimulatorImpl` class and enable the calculation of checksums for packets.[75]

```
1   GlobalValue::Bind ("SimulatorImplementationType", StringValue ("ns3::RealtimeSimulatorImpl"));
2   GlobalValue::Bind ("ChecksumEnabled", BooleanValue (true));
```

We then create one eNodeB and a number of UEs equal to $numUe$.

```
1   NodeContainer enb;
2   NodeContainer ue;
3   enb.Add (CreateObject<Node> (0));
4   for(uint32_t i = 0; i < numUe; ++i)
5           {
6                   ue.Add (CreateObject<Node> (0));
7           }
```

Next we install the `ConstantPositionMobilityModel` on both node containers.[76]

```
1   MobilityHelper mobility;
2   mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
3   mobility.Install (enb);
4   mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
5   mobility.Install (ue);
```

We then instantiate a `LenaHelper` to configure LTE model parameters. We set the MAC scheduler type and RLC modes for the eNodeB and UEs. We then install LTE network devices on all nodes and attach each UE to the cell.

```
1   Ptr<LenaHelper> lena = CreateObject<LenaHelper> ();
2   lena->SetSchedulerType ("ns3::PfFfMacScheduler");
3   lena->SetAttribute("EnbRlcMode", EnumValue(LteRlc::RLC_SM));
4   lena->SetAttribute("UeRlcMode", EnumValue(LteRlc::RLC_UM));
5
6   NetDeviceContainer enbUtranDev = lena->InstallEnbDevice (enb.Get(0));
7   NetDeviceContainer ueDev = lena->InstallUeDevice (ue);
8   lena->Attach (ueDev, enbUtranDev.Get (0));
```

Before calling `Run ()`, we set the trace output filename. After the simulation completes, we call `LenaHelper::WriteTraces ()` to dump the trace output to the disk.

```
1   lena->EnableDlRlcTraces ("dl-rlc-trace.csv");
2
3   Simulator::Stop(Seconds(stopTime));
4   Simulator::Run ();
5
6   lena->WriteTraces ();
```

[75] Checksums pass or fail based on classes derived from `ErrorModel` for different packet types (see [74].)

[76] We use the origin as the default position for all nodes, resulting in zero propagation delay calculated from the mobility model. Of course this is not a practical scenario, however we are only concerned with measuring the real-time jitter in these experiments, which is unrelated to he mobility model. It should be noted that we are not in any way trying to validate the PHY and MAC-layer components of the LTE model, since their correctness and non-real-time performance has already been evaluated in [80]. See Section IV for further discussion.

10.2.1.2 Simulation Execution

We run the simulation for $numUe = 1, 2, 4, 8, 16, 32, 64$ mobiles for 5 seconds and average the jitter statistics from two trials.

```
./lte-saturation-test --numUe=1 --stopTime=5.0
```

10.2.1.3 Analysis of Real-Time Jitter Results

We skip the procedure for checking log messages it is sufficient to observe the trace output in order to verify the correct behavior of the model. Table C.20 gives the jitter statistics for this test case. The RLC *RxPDU* jitter is plotted over time in Figure B.21 and B.22. We see that for one and two UEs, we never exceed 1ms of real-time latency, with an average jitter for the 2-UE case of $71\mu s$ and a maximum of $268\mu s$. Note that this average jitter is over ten times what is was for the `MacRx` event for `PointToPointNetDevice`, which is owed to the overall complexity of the LTE model. For $numUe = 8$, we observe an average jitter of $178\mu s$, but, as we can see in Figure B.21, there are consistent spikes in latency in excess of 1ms, with a maximum recorded jitter of $3.4ms$ over two trials. We observe the same pattern for $numUe = 16$, with the jitter not exceeding $2.46ms$. For the first time, we observe that an unsustainable amount of computation that is evidently required 32 and 64 mobiles. In Figure B.22, the jitter seems to increase linearly with time and quickly exceeds 1ms. It appears that, after a certain point, event execution can no longer keep up with real-time at all, and so latencies become additive. We should keep in mind that we are simulating the theoretical worst-case traffic pattern for a single cell. The eNodeB is the bottleneck here, and it is interesting that the latencies are dependent on the number of mobiles since we are still transmitting the same volume of data in the 1-UE case as in the 64-UE case.

10.2.2 Simple Traffic Generator Test Case

We now hope to demonstrate more practical traffic pattern that still creates a scenario of high load but does not completely saturate the simulated network, thereby allowing the algorithm more of a window of opportunity to resynchronize with real-time after each event and, in turn, allowing a larger number of UEs to be supported.

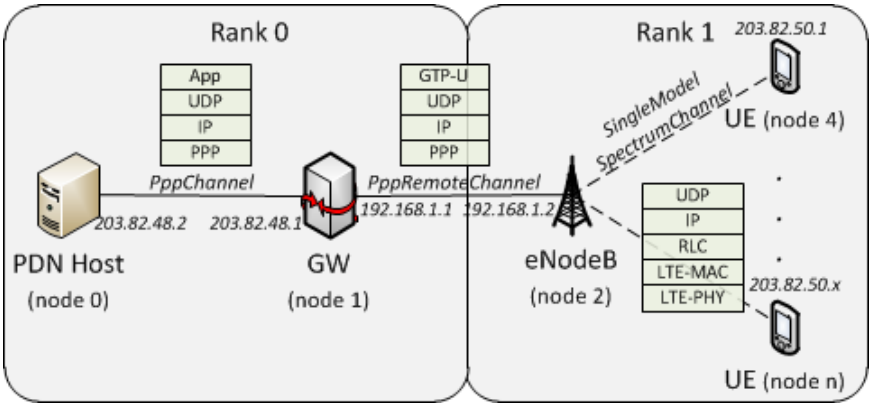


Figure 10.4: LTE network test case with simple traffic generation

Here we constructed a simplified EPS network with one gateway (representing a

co-located P-GW/S-GW), which filters traffic from an external host and transmits IP packets a GTP-U tunnel to the eNodeB. A *UdpEchoClient* installed on the PDN host to send a KB-sized packet every 100ms to each UE. As we increase the number of UEs, we increase the load on the core network links and E-UTRAN and observe the jitter statistics for both `PointToPointNetDevice::Receive ()` (the output of the *MacRx* trace) and `LteSpectrumPhy::EndRx ()` events (from the `LteUeRlc` *RxPDU* trace).

10.2.2.1 Simulation Configuration

The file *lte-network.cc* contains the `int main ()` function that defines our simulation scenario. We first parse command line arguments as in the last example code. We then create four `NodeContainer` objects: *pdn*, *gw*, *enb* and *ue*. We install the mobility model on the eNodeB and UEs and set up the RAN network using a `LenaHelper` as before, but using *RLC_UM* now instead of *RLC_SM* for the eNodeB. We also create the aforementioned PPP links.

```
1  PointToPointHelper ppp;
2  ppp.SetDeviceAttribute ("DataRate", StringValue ("1000Mbps"));
3  ppp.SetChannelAttribute ("Delay", StringValue ("1ms"));
4
5  NetDeviceContainer sgiDev = ppp.Install (pdn.Get(0), gw.Get(0));
6  NetDeviceContainer s1uDev = ppp.Install (gw.Get(0), enb.Get(0));
```

In the saturation test case, no TCP/IP protocol stack was installed on our nodes. As we are now dealing with application-layer traffic and not dummy PDUs, we use `InternetStackHelper` to allocate the stack and `Ipv4AddressHelper` to assign IP addresses to PPP and LTE devices.

```
1  Ipv4ListRoutingHelper list;
2  Ipv4GlobalRoutingHelper globalRouting;
3  Ipv4StaticRoutingHelper staticRouting;
4  list.Add (staticRouting, 10);
5  list.Add (globalRouting, 0);
6
7  InternetStackHelper epsStack;
8  epsStack.SetRoutingHelper (list);
9  epsStack.Install (pdn.Get(0));
10 epsStack.Install (gw.Get(0));
11 epsStack.Install (enb.Get(0));
12
13 InternetStackHelper utranStack;
14 utranStack.SetRoutingHelper(staticRouting);
15 utranStack.Install (ue);
16
17 Ipv4AddressHelper sgiIpAddr;
18 sgiIpAddr.SetBase ("203.82.48.0", "255.255.255.0");
19 Ipv4InterfaceContainer sgiIpIf = sgiIpAddr.Assign (sgiDev);
20
21 Ipv4AddressHelper s1uIpAddr;
22 s1uIpAddr.SetBase ("192.168.1.0", "255.255.255.0");
23 Ipv4InterfaceContainer s1uIpIf = s1uIpAddr.Assign (s1uDev);
24
25 Ipv4AddressHelper ueIpAddr;
26 ueIpAddr.SetBase ("203.82.50.0", "255.255.255.0");
27 Ipv4InterfaceContainer ueIpIf = ueIpAddr.Assign (ueDev);
```

We use the private address range 192.168.1.x for S1-U devices (between the GW and eNodeB) and the public ranges of 203.82.48.x and 203.82.50.x for the PDN and mobiles, respectively. We must also do some magic with the routing configuration.

```
globalRouting.PopulateRoutingTables ();

Ptr<Ipv4> pdnIpv4 = pdn.Get(0)->GetObject<Ipv4> ();
Ptr<Ipv4StaticRouting> pdnRoutes = staticRouting.GetStaticRouting(pdnIpv4);
pdnRoutes->AddNetworkRouteTo("203.82.50.0", "255.255.255.0",
                        pdnIpv4->GetInterfaceForDevice(sgiDev.Get(0)));

Ptr<Ipv4> enbIpv4 = enb.Get(0)->GetObject<Ipv4> ();
Ptr<Ipv4StaticRouting> enbRoutes = staticRouting.GetStaticRouting(enbIpv4);
enbRoutes->AddNetworkRouteTo("203.82.50.0", "255.255.255.0",
                        enbIpv4->GetInterfaceForDevice(enbUtranDev.Get(0)));

for(uint32_t i = 0; i < ue.GetN (); ++i)
        {
                Ptr<Ipv4> ueIpv4 = ue.Get(i)->GetObject<Ipv4> ();
                Ptr<Ipv4StaticRouting> ueRoutes =
                                                staticRouting.GetStaticRouting(ueIpv4);
                ueRoutes->AddNetworkRouteTo("255.255.255.255", "0.0.0.0",
                                ueIpv4->GetInterfaceForDevice(ueDev.Get(i)));
        }
```

Since the global routing helper is not able to dynamically resolve routes between the gateway and eNodeB, we must establish static routes between the PDN and mobiles.

```
Ptr<EpcGtpU> gwGtpU = CreateObject<EpcGtpU> ();
gwGtpU->Install(EPC_PGW, s1uDev.Get(0));

Ptr<EpcGtpU> enbGtpU = enbUtranDev.Get(0)->GetObject<LteEnbNetDevice> ()->GetGtp ();
enbGtpU->Install (EPC_ENODEB, s1uDev.Get(1));

std::vector<Ipv4Address> ueIpAddrs;
Ipv4Address ueIpAddrBase = Ipv4Address("203.82.50.1");
for(uint32_t i = 0; i < ue.GetN (); ++i)
{
         Ipv4Address nextUeIpAddr = Ipv4Address (ueIpAddrBase.Get () + i);
         ueIpAddrs.push_back (nextUeIpAddr);
}

Ptr<LteEnbRrc> enbRrc = enbUtranDev.Get(0)->GetObject<LteEnbNetDevice> ()->GetRrc ();
gwGtpU->Attach(s1uDev.Get(0), s1uDev.Get(1), enbRrc, ueIpAddrs);
```

We also must configure the GTP-U protocol on the S1-U interface and add an entry for each UE (including IP addresses) to a data structure used by the gateway for mapping of data traffic to tunnels.

10.2.2.2 Simulation Execution

We run the simulation for $numUe = \{1, 2, 4, 8, 16, 32, 64\}$ mobiles for 5 seconds and average the jitter statistics from two trials.

```
./lte-saturation-test --numUe=1 --stopTime=5.0
```

10.2.2.3 Analysis of Real-Time Jitter Results

Table C.21 gives the jitter statistics logged by the RLC *RxPDU* and PPP *MacRx* traces. The jitter for these events is plotted over time in Figures B.23 through B.26. The average and maximum-recorded RLC jitter is actually higher than it is in the RAN saturation scenario for less than 8 mobiles. This can be accounted for by the additional PPP

events in the event set. For $numUe = 8$, we see the jitter is slightly less than it is in RLC_SM at $102\mu s$, with the maximum jitter dropping to $229\mu s$. We see similar latencies for PPP events on the order of hundreds of microseconds. In Figure B.24 and B.26, we see an interesting behavior for $numUe = 16$. While most events experience a latency of a couple hundred μs, intermittent spikes of over 1ms are accompanied by more infrequent spikes of multiple milliseconds. As this same behavior occurs for other trials as we shall see, it would seem that event processing becomes backlogged at certain identifiable points. In our future work, we hope to investigate the source of this bottlenecking effect further.

10.2.3 Distributed Network Test Case

We repeat the previous experiment parallelized using the `DistributedSimulatorImpl` core simulator with the topology in Figure 10.4 partitioned into the core and RAN across the S1-U interface (between the eNodeB and GW).

10.2.3.1 Simulation Configuration

We configure the simulator to use the `DistributedSimulatorImpl` algorithm, as previously demonstrated in Section 10.1.1.1. We then assign the *enb* and *ue* node containers to rank 0 and the *pdn* and *gw* node containers to rank 1. All other code remains the same.

10.2.3.1.1 Simulation Execution

We execute the *lte-network-dist* program using the *mpirun* utility. For the internode case, we include the hostname *ltesim2* in the host list (delimited by *-H*). We run the simulation for $numUe = \{1, 2, 4, 8, 16, 32, 64\}$ mobiles for 5 seconds and average the jitter statistics from two trials.

mpirun -np 2 -H ltesim1 ./lte-network-dist –numUe=2 –stopTime=5.0 –isRealtime=1

10.2.3.2 Analysis of Simulation Real-Time Jitter Results

Results from this test case are tabulated in Table C.22 (the intranode case) and Table C.23 (for the internode case). The jitter is plotted over time in Figures B.23 through B.26. For the distributed version of the simple traffic generator experiment, we see a notable improvement over the single-process case PPP events, with jitter experienced at the *gw* and *pdn* host nodes dropping from $150\mu s$ to $22\mu s$ in the 8-UE case. A slight improvement is seen for RLC events at UE nodes and PPP events at the eNodeB. The average and maximum jitter generally increased in the internode case, although only by tens of microseconds.

10.2.4 Real Traffic Test Case

At last, we have arrived at our proof-of-concept testbed simulation, which is the product of our implementation efforts up to this point. Using the tap-bridge emulation feature of ns-3, an HTTP connection is established over our simulated EPS topology between a VM representing a LTE User Equipment and another VM representing a web server in the external packet data network, as shown in Figure 10.5. With the *wget* utility, we transfer a large file hosted by the HTTP server to the UE. As before, we observe the real-time jitter for PPP and RLC events for both the default real-time

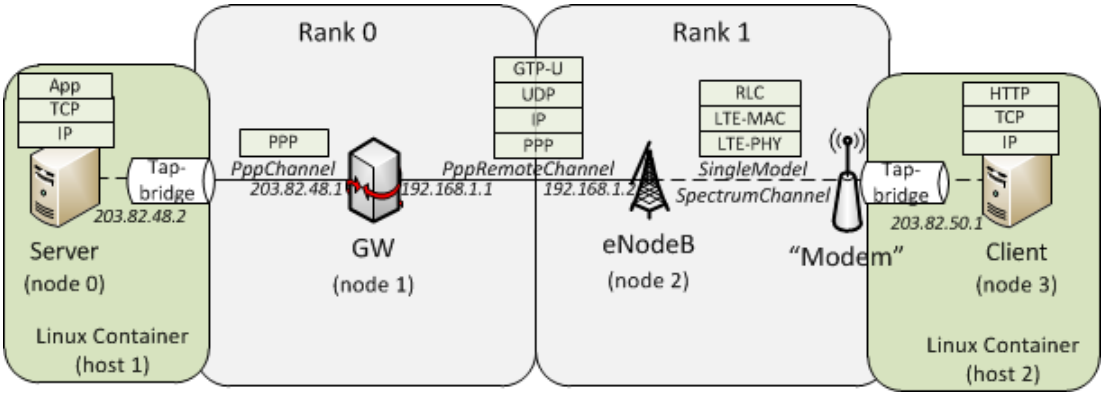


Figure 10.5: LTE network test case with real traffic (software-in-the-loop)

(`RealtimeSimulatorImpl`) and distributed (`DistributedSimulatorImpl`) core simulators.

10.2.4.1 Simulation Configuration

Our simulation source file, *lte-network-tap.cc*, *Gw* and *enb* nodes are configured as in the preceding sections. For the *pdn* and *ue* nodes, we do not install the ns-3 TCP/IP stack using `InternetStackHelper` or assign addresses, as this is handled by the native stack in each VM.

We first parse in the *isDistributed* argument from the command line. We use this to enable or disable distributed mode in the `TapBridge` settings.

```
bool isDistributed = false;

CommandLine cmd;
cmd.AddValue ("isDistributed", "", isDistributed);
```

```
TapBridgeHelper tapBridge;
tapBridge.SetAttribute ("Mode", StringValue ("UseBridge"));
tapBridge.SetAttribute ("IsDistributed", BooleanValue(isDistributed));
tapBridge.SetAttribute ("Rank", UintegerValue(pdnRank));
tapBridge.SetAttribute ("DeviceName", StringValue ("tap-left"));
tapBridge.Install (pdn.Get (0), sgiDev.Get (0));

tapBridge.SetAttribute ("Rank", UintegerValue(0));
tapBridge.SetAttribute ("DeviceName", StringValue ("tap-right"));
tapBridge.Install (ue.Get (0), ueDev.Get (0));
```

Using a `TapBridgeHelper`, we install a tap-bridge device on each of the two "real" nodes, setting the device to *UseBridge* mode.[77] We assign these devices to the TUN/TAP interfaces *tap-left* and *tap-right*, corresponding to the virtual interfaces for the HTTP server and UE VMs, respectively. We have extended the `TapBridge` class to work with the `DistributedSimulatorImpl` simulator by statically assigning each device to a rank, as shown above.[78]

We must now set up our bridge and virtual TUN/TAP interfaces with the *brctl* and *tunctl* utilities. We follow the procedure in [40] to create the interfaces *br-left*, *br-right*, *tap-left*, and *tap-right* on the host OS.

[77] See [40] for a detailed tutorial on ns-3 emulation.

[78] This feature does not yet support internode MPI operation. Processes spawned with *mpirun* must belong to a single host.

We then configure and start our Linux Container VM environments. Two simple config files, *lxc-left.conf* and *lxc-right.conf*, contain parameters for initializing each LXC. The contents of *lxc-left.conf* are as follows.

```
lxc.utsname = left
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br-left
lxc.network.ipv4 = 203.82.48.2/24
```

The container is assigned the hostname *left*. A virtual interface with the prefix *veth* is automatically created when the container is started, and is bridged with the *tap-left* device by *br-left*. Within the container, *veth* appears as *eth0* with the IP address shown. The *lxc-right.conf* file for the UE container has the same format, with *br-left* replaced by *br-right* and the address 203.82.50.1 assigned to its virtual *eth0*.

10.2.4.1.1 Simulation Execution

Before running the simulation, we create and then start each container with the following commands.

lxc-create -n lxc-left lxc-start -n left /bin/bash

We repeat these commands for the *right* container in a separate terminal window. After entering the second command on the host OS, we are presented with a root *bash* shell for the LXC environment.[79]

For the two-process case, we run the *lte-network-tap* program using *mpirun* from the shell of the host OS.

mpirun -np 2 -H ltesim1 ./lte-network-tap –isDistributed=1 –stopTime=120.0

For the uniprocess case, we simply omit the *mpirun* component in the above command. We also specify the stop time as 120 seconds. Once the simulation has started, we enter the *right* container (i.e. the virtual UE) and, using the *wget* command, request the file from the HTTP server.

wget http://203.82.48.2/largefile.bin

10.2.4.1.2 Analysis of Simulation Results

Table C.24 gives the RLC *RxPDU* and PPP *MacRx* event jitter statistics for each test. The jitter is also plotted over time in Figures B.35 and B.36.[80] Table C.25 gives statistics on the HTTP transfer as reported by *wget*. The difference in average, maximum and minimum jitter between the one and two-processes tests is negligible. From Figure B.35, we see that only once does the latency exceed $200\mu s$ in the reception of RLC

[79] Root privileges are required to create, start and destroy LXC containers.

[80] The jitter plot for each trial appears slightly shifted in time due to the HTTP transfer starting at different times.

PDUs at the UE, spiking to about $1.4ms$ for just a single event. While t average jitter is only $12\mu s$ for PPP events, Figure B.36 shows frequent spikes above 1 and 2ms. The occurrence of these spikes seems to be reduced in the two-process, distributed test, though the average is roughly the same when compared to the uni-process case.

Part IV
Conclusion

11 Realization of Goals

Significant progress has been made towards laying the groundwork for a software platform that should prove to be an invaluable tool for researching next-generation cellular wireless systems. The proof-of-concept *software-in-the-loop* experiment in the preceding section demonstrates the culmination of our efforts toward the realization of the requirements defined in Sections 2 and 3 for this phase of the project. Undertaking to achieve these initial goals, which are quite broad in scope, requires technical knowledge from a variety of disciplines. The background given in the first half of this thesis draws from such a wide array of topics as discrete-event simulation, parallel programming, the GNU Linux kernel, high-performance computing, telecommunications networks and digital wireless communications systems. Bringing together all of the requisite information represents a substantial achievement by itself, and strides were made in the design and implementation of the software systems and compute cluster platform, as well. Nonetheless, there is still much room left for improvement. We shall now return to each of the high-level goals outlined in Section 2 and analyze our progress toward achieving them. In Section 12, we propose a plan for advancing each of these areas in future iterations of the project.

11.1 Model Validity

Only the most minimal set of features required to demonstrate end-to-end IP connectivity over an LTE/SAE network have thus far been implemented. In addition to the PHY and MAC-layer functions of the LTE radio offered by the LENA project classes [80], a simplified model of RLC Unacknowledged Mode and GTP-U tunneling were incorporated into the overall model. Validation of the channel model (i.e. the *Spectrum* model for ns-3), LTE MAC scheduler, Adaptive Modulation and Coding scheme, and other components of the LENA model is detailed in [81] and [80]. The implementation of the RLC protocol is also correctly follows the segmentation and reassembly procedure prescribed in 3GPP TS 36.322 [69], with the format of RLC headers and PDUs accurately reflecting the specifications bit-for-bit. Similarly, the GTP-Uv1 model provides an accurate bit-level representation of the GTP-U header and includes all relevant data elements (see 3GPP TS 29.281 [70]) needed to properly encapsulate basic IP packets and tunnel them between the external network and user equipment over the core network. More advanced traffic filtering and QoS aspects of GTP are as yet unimplemented. The same can be asserted for all relevant protocols in the TCP/IP stack that are provided in the ns-3 model library. The relevant specifications and RFCs for the IP (`Ipv4L3Protocol`), TCP (`TcpL4Protocol`), UDP (`UdpL4Protocol`) and other classes are cited in [74].

11.2 Real-Time Performance

Considerable time and effort was devoted to analyzing the real-time performance of core DES algorithms and LTE/SAE model components. For the very basic LTE test simulations in Section 10, it was shown that for a small number (less than 16) of mobiles, real-time jitter averaged to no more than several hundred microseconds for individual events, with infrequent, intermittent latency spikes of over 1 millisecond. In the real traffic test case involving one "real" PPP host and one LTE mobile, we see an average of around $60\mu s$, with the jitter never exceeding 1.5ms. The degree to which this added delay in event processing affects simulation accuracy and validity depends on the scenario being tested and the nature of the desired results. For testing protocols purely over the air interface, a jitter of over 1ms may be unacceptable. Since we are mostly interested in testing the behavior of protocols and their effects on higher-layer protocols and applications, the impact of an additional 1ms delay may be imperceptible. For instance, if we wish to observe the subframe error rate of a new coding scheme and the resulting effects on streaming video over Real-Time Streaming Protocol and TCP, the only deviation from true emulation of such a system would be the delay experienced at the IP layer and above. In this example, an added delay of 1ms is unlikely to have any compromising effect on the behavior of any applicable protocol and will not apparent in the playback of the video.

11.3 Parallel DES Algorithms

The one area in which our product clearly comes up short of the intended project goals is efficient parallel simulation. The pthread and MPI-based core simulator implementations we tested both offered only a small fraction of performance gain over the default implementation when two concurrent execution paths were employed. For additional threads or processes, both simulators performed worse than the default. Their failure is a result of the simplicity of the algorithms, which are not optimized for the kinds of practical, real-world cellular networks that we are dealing with. In [82], it was shown that a linear performance gain (proportional to the number of processes) is possible with the `DistributedSimulatorImpl` simulator for a network with 200ms of lookahead. This 200ms delay is not common to any practical technology. Instead, the 1ms lookahead for PPP links in our simulation scenarios resulted in much less of a window of opportunity for parallelization of events. Any gain from parallelization was dominated by the cost of frequent synchronization and communication of null messages between logical processes.

Furthermore, neither of these core simulators support partitioning of wireless networks, or any type of broadcast network for that matter. Only network partitions connected by point-to-point links are supported. This feature will be incorporated during the next iteration of the project by allowing a many-to-many communications scheme for scheduling remote events and exchanging null-messages.

12 Future Work

12.1 Improved Parallel Simulation Algorithms

It is apparent that a more advanced conservative synchronization algorithm must be employed to achieve a superior speedup. As discussed, real-time simulations are, in principle, limited to conservative techniques, and optimistic algorithms that have found success in simulators such as Georgia Tech Time Warp [28] would not be effective. [21] and [27] evaluate several horizontal partitioning algorithms that use the technique of

event bundling to reduce the frequency of synchronization. [24], [26] and [25] suggest several methods for increasing lookahead for wireless networks by exploiting properties of protocols. The concepts of *protocol lookahead* and *event lookahead* are introduced in [27] as a way to augment the standard transmission delay-based lookahead model.

Furthermore, we propose two novel approaches to synchronization, the first of which is based on our observation that synchronization is the true performance bottleneck in such simulations in contrast to the communication cost of exchanging null messages, which is relatively cheap. If, in the process of scheduling an event, all neighboring LPs could be asynchronously updated with the timestamp of the new event, there would be no need to wait at a barrier after each event processing iteration in order to resynchronize. LPs could then continuously process events without running the risk of becoming inconsistent. The number of null messages would increase considerably, however, as each partition would need to maintain the minimum timestamp of the event set for neighboring partitions. This would also increase the overall memory requirements of the simulation.

The second proposed solution requires increased integration between the network model and the core simulator algorithm. If each event could be given some tag that lists other partitions that are effected by an event, only those events would need to be considered in the calculation of Lower Bound Timestamps, which would, in some cases, result in a larger parallelization window. Both of these modifications will be evaluated in our future work.

12.2 VM-Synchronized Emulation

SliceTime [29] is a system that enables non-real-time emulation by integrating ns-3 event processing with the system scheduler in the Xen hypervisor. By effectively slowing down the execution of instructions for a Xen-based virtual machine to match the execution time of ns-3 events, real applications can be integrated with a simulation that requires more computation than can be supported in real-time. As a result, simulations can be scaled up indefinitely, with non-real-time results being theoretically identical to those of a real-time emulation.

12.3 Alternative Parallel and Distributed Solutions

While still widely in use today, the pthread, MPI and OpenMP frameworks we have so far investigated are no longer the only option for developers of parallel applications. The CUDA [93] and OpenCL [92] frameworks enable programs to harness multiple shared-memory CPUs as well as Graphics Processing Units (GPUs), which can have hundreds of parallel processing units per chip.[81] We hope to investigate how these frameworks can be integrated with the ns-3 simulator. General-purpose Computing on Graphics Processing Units (GPGPU) may enable computationally-expensive operations such as error calculations to be sped up significantly.

13 Source Code Availability

Our code is still in need of extensive testing and debugging before it will be made available through the ns-3 code repository at *http://code.nsnam.org/ns-3-dev*. Presently, the code can be obtained at our SourceForge repository at *http://chic.svn.sourceforge.net*. One can also visit the website for the NYU Poly Center for Advanced Technology in Telecommunications at *http://catt.poly.edu*.

[81] [91] presents some initial work with integrating CUDA support into ns3.

References

[1] Cisco Systems, Inc., “Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2010-2015.”

[2] National Instruments Corp., “NI LTE Measurement Suite,” http://www.ni.com

[3] S. Kumar, V. S. Raghavan, and J. Deng, ”Medium Access Control protocols for ad hoc wireless networks: A survey,” IEEE Wireless Comm., vol. 12, no. 2, pp. 19 - 29, Apr. 2005.

[4] V. Brik, E. Rozner, S. Banerjee, and V. Bahl, “DSAP: A Protocol for Coordinated Spectrum Access,” in Proc. Sixth IEEE DySPAN, Nov. 2005.

[5] M. Buddhikot, P. Kolodzy, S. Miller, K. Ryan, and J. Evans, “DIMSUMnet: new directions in wireless networking using coordinated dynamic spectrum,” in Proc. Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM’05), Jun. 2005.

[6] S. Balasubramaniam and J. Indulska, “Vertical handover supporting pervasive computing in future wireless networks,” Computer Communications, vol. 27, no. 8, pp. 708-719, May 2004.

[7] H. Schulzrinne and E. Wedlund, “Application Layer Mobility Using SIP,” in ACM SIGMOBILE Mobile Computing and Communications, vol. 4, no. 3, July 2000.

[8] K.J. Ma., R. Bartos, S. Bhatia, R. Nair, “Mobile video delivery with HTTP,” IEEE Communications Magazine, vol. 49, issue 4, April 2011.

[9] Aiglent Technologies, “Agilent Technologies’ New LTE Base-Station Emulator Speeds Development and Verification of LTE User Equipment,” http://www.agilent.com/about/newsroom/presrel/2010/01sep-em10115.html .

[10] Polaris Networks, “LTE Network Equipment Emulators,” http://www.polarisnetworks.net/lte_emulators.html

[11] M.A.L. Sarker, M.H. Lee, “FPGA-based MIMO testbed for LTE applications,” in Proc. of the Eighth International Conference on Wireless and Optical Communications Networks (WOCN), May, 2011.

[12] D. Bates, S. Henriksen, B. Ninness, S.R. Weller, “A 4×4 FPGA-based wireless testbed for LTE applications,” in Proc. of the IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications, Sept. 2008.

[13] M. Guizani, A. Rayes, B. Khan, A. Al-Fuquah, *Network Modeling and Simulation*, Wiley, 2010.

[14] R. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley, 2000.

[15] J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++*, Wiley, Dec. 2010.

[16] R.M. Fujimoto, K. Perumalla, A. Park, H. Wu, M.H. Ammar, G.F. Riley, "Large-scale network simulation: how big? how fast?," in Proc. of the 11th IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2003.

[17] Advanced Micro Devices, Inc., "Direct Connect Architecture," http://www.amd.com/us/products/technologies/direct-connect-architecture/Pages/direct-connect-architecture.aspx .

[18] T. Rauber, G. Ränger, *Parallel Programming For Multicore and Cluster Systems*, Springer, 2010.

[19] D.M. Nicol, "Scalability, locality, partitioning and synchronization in PDES," in Proceedings. Twelfth Workshop on Parallel and Distributed Simulation, 1998.

[20] K. Yocum, E. Eade, J. Degesys, D. Becker, J. Chase and A. Vahdat, "Toward Scaling Network Emulation using Topology Partitioning," in Proc. of the 11th IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2003.

[21] G. Kunz, O. Landsiedel1, S. Götz, K. Wehrle, J. Gross, F. Naghibi, "Expanding the Event Horizon in Parallelized Network Simulations," in Proc. of the 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2010.

[22] K.M. Chandy, J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," IEEE Transactions on Software Engineering, Volume SE5, Issue 5, pp. 440452, 1979.

[23] R.E. Bryant, "Simulation of packet communication architecture computer systems," Technical report, Massachusetts Institute of Technology Cambridge, MA, USA, 1977.

[24] J. Liu, D.M. Nicol, "Lookahead Revisited in Wireless Network Simulations," in Proc. of the 16th Workshop on Parallel and Distributed Simulation, 2002.

[25] R. A. Meyer, R. L. Bagrodia, "Improving Lookahead in Parallel Wireless Network Simulation," in Proc. of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998.

[26] X. Liu, D.M. Nicol, "Improvements in conservative parallel simulation of large-scale models," Doctoral Dissertation, Dartmouth College, 2003.

[27] P. Peschlow, A. Voss, P. Martini, "Good news for parallel wireless network simulations," in Proc. of the 12th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems, 2009.

[28] S. Das, R. Fujimoto, K. Panesar, D. Allison, M. Hybinette, "GTW: a time warp system for shared memory multiprocessors," in Proceedings of the 26th conference on Winter simulation, Dec. 1994

[29] E. Weingärtner, F. Schmidt, H. vom Lehn, T. Heer, K. Wehrle, "SliceTime: A platform for scalable and accurate network ," in Proc. of the 8th USENIX conference on Networked systems design and implementation, 2011.

[30] T.H. Cormen , C.E. Leiserson, R.L. Rivest, C.Stein, *Introduction to Algorithms*, The MIT Press, third edition, July 31, 2009.

[31] D. Brown, "Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem," Communications of the ACM CACM, Volume 31, Issue 10, Oct. 1988.

[32] K. L. Tan, L. Thng, "SNOOPy Calendar Queue," in Proc. of the 32nd conference on Winter simulation, 2000.

[33] P.M.Papazoglou, D.A.Karras and R.C.Papademetriou, "An Efficient Distributed Event Scheduling Algorithm for Large Scale Wireless Communications Simulation Modelling," in Proc. of IWSSIP 2009. 16th International Conference on Systems, Signals and Image Processing, 2009.

[34] B. Barney, "Introduction to Parallel Computing," Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/ .

[35] B. Barney, "POSIX Threads Programming," Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/ .

[36] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, Oct. 2010.

[37] B. Barney, "Message Passing Interface (MPI)," Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/mpi/

[38] D. Mahrenholz, S. Ivanov, "Real-Time Network Emulation with ns-2," in Proc. of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications, Oct. 2004.

[39] A. Alvarez, R. Orea, S. Cabrero, X. G. Pañeda, R. García, D. Melendi, "Limitations of Network Emulation with Single-Machine and Distributed ns-3," in Proc. of the 3rd International ICST Conference on Simulation Tools and Techniques, 2010

[40] C. Dowell, "HOWTO Use Linux Containers to set up virtual networks," ns-3 Wiki, Feb. 2010, http://www.nsnam.org/wiki

[41] M. Tim Jones, "Inside the Linux scheduler," http://www.ibm.com/developerworks/linux/library/l-scheduler/

[42] L. DiMaggio, "Understanding your (Red Hat Enterprise Linux) daemons," Red Hat Magazine, March, 2009, http://magazine.redhat.com/2007/03/09/understanding-your-red-hat-enterprise-linux-daemons/

[43] "Configuring and Managing a Red Hat Cluster," Red Hat, Inc., http://www.centos.org/docs/5/html/5.1/Cluster_Administration/

[44] ”Linux Setting processor affinity for a certain task or process,” http://www.cyberciti.biz/tips/setting-processor-affinity-certain-task-or-process.html

[45] “Cpuset Management Utility,” https://rt.wiki.kernel.org/articles/c/p/u/Cpuset_Management_Utility_89c8.html

[46] “Open MPI: Open Source High Performance Computing - Frequently Asked Questions,” http://www.open-mpi.org/faq/

[47] “Real-Time Linux Wiki,” https://rt.wiki.kernel.org

[48] ”chrt(1) - Linux man page,” http://linux.die.net/man/1/chrt

[49] G. Jost, H. Jin, D.A. Mey, F.F. Hatay, “Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster,” NAS Technical Report NAS-03-019, Nov. 2003.

[50] E. Lusk, A. Chan, ”Early Experiments with the OpenMP/MPI Hybrid Programming Model,” Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago, 2006.

[51] M. Luo, S. P., P. Lai, E.P. Mancini, H. Subramoni, K. Kandalla, S.S. Dhabaleswar K. Panda, “High Performance Design and Implementation of Nemesis Communication Layer for Two-sided and One-Sided MPI Semantics in MVAPICH2,” in Proc. of the 2010 39th International Conference on Parallel Processing Workshops, 2010.

[52] ”gettimeofday(2) - Linux man page,” http://linux.die.net/man/2/gettimeofday

[53] ”pthread_cond_timedwait, pthread_cond_wait - wait on a condition,” The Open Group Base Specifications Issue 6, http://pubs.opengroup.org/onlinepubs/000095399/functions/pthread_cond_wait.html

[54] “time - overview of time and timers,” http://linux.die.net/man/7/time

[55] M. Rumney, Aiglent Technologies, *LTE and the Evolution to 4G Wireless: Design and Measurement Challenges*, Wiley, 2009.

[56] S. Sesia, I. Toufik, M. Baker, *LTE The UMTS Long Term Evolution: From Theory to Practice*, John Wiley & Sons, 2009.

[57] E. Dahlman, S. Parkvall, J. Sköld, P. Beming, *LTE/LTE Advanced for Mobile Broadband*, Academic Press, 2011.

[58] M. Olsson, S. Sultana, S. Rommer, L. Frid, C. Mulligan, *SAE and the Evolved Packet Core*, Academic Press, 2009.

[59] FemtoForum, “LTE MAC Scheduler Interface Specification v1.11,” http://www.femtoforum.org/femto/technical.php, Oct. 2010.

[60] 3GPP, “UTRA-UTRAN Long Term Evolution (LTE) and 3GPP System Architecture Evolution (SAE),” http://www.3gpp.org/article/lte, 3GPP technical paper, Oct. 2006.

[61] Rohde & Schwartz, “UMTS Long Term Evolution (LTE) Technology Introduction,” Rohde & Schwartz whitepaper, Sept. 2008.

[62] H. Holma, A. Toskala, *LTE for UMTS: Evolution to LTE-Advanced*, John Wiley & Sons, 2011.

[63] NMC Consulting Group, “LTE X2 Handover,” http://www.nmcgroups.com/en/expertise/lte/x2handover.asp .

[64] Santosh Dornal, “Wired n Wireless,” http://wired-n-wireless.blogspot.com .

[65] 3GPP, Technical Specification Group Radio Access Network, “Evolved Universal Terrestrial Radio Access (E-UTRA); User Equipment (UE) radio transmission and reception (Release 10),” 3GPP TS 36.213, Sept. 2011.

[66] 3GPP, Technical Specification Group Radio Access Network, “Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2 (Release 10),” 3GPP TS 36.213, Sept. 2011.

[67] 3GPP, Technical Specification Group Services and System Aspects, “Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2 (Release 10),” 3GPP TS 23.002, Jan. 2011.

[68] 3GPP, Technical Specification Group Services and System Aspects, “General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access (Release 10),” 3GPP TS 23.401, Jan. 2011.

[69] 3GPP, Technical Specification Group Radio Access Network, “Radio Link Control (RLC) protocol specification (Release 10),” 3GPP TS 36.322, Dec. 2010.

[70] 3GPP, Technical Specification Group Core Network and Terminals, “General Packet Radio System (GPRS) Tunnelling Protocol User Plane (GTPv1-U) (Release 10),” 3GPP TS 29.281, March 2011.

[71] IETF RFC 2960, “Stream Control Transmission Protocol,” http://tools.ietf.org/html/rfc2960

[72] “ns-3 Manual,” http://www.nsnam.org/ns-3-12/documentation/

[73] “The Network Simulator - ns-2,” http://isi.edu/nsnam/ns/

[74] “ns-3 Documentation,” http://www.nsnam.org/docs/release/3.12/doxygen/index.html

[75] “lxc Linux Containers,” http://lxc.sourceforge.net/

[76] “bridge,” the Linux Foundation, Nov. 2009, http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge

[77] “Universal TUN/TAP driver - Frequently Asked Questions,” http://vtun.sourceforge.net/tun/faq.html

[78] J. Radajewski, D. Eadline, “Beowulf HOWTO,” Nov. 1998, http://www.ibiblio.org/pub/linux/docs/HOWTO/archive/Beowulf-HOWTO.html

[79] “LTE Simulator Documentation Release M1,” Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), June 2011.

[80] N. Baldo, M. Miozzo, M. Requena-Esteso, J. Nin-Guerrero, “An Open Source Product-Oriented LTE Network Simulator based on ns-3,” in Proc. of the 14th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems, April 2011.

[81] N. Baldo “Spectrum-aware Channel and PHY layer modeling for ns3,” in Proc. of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools, April 2011.

[82] J. Pelkey, G.Riley, “Distributed Simulation with MPI in ns-3,” 2011 Workshop on ns3, March 25, 2011.

[83] G. Seguin, “Multi-core parallelism for ns-3 simulator,” INRIA Sophia-Antipolis, Technical Report, Aug. 2009.

[84] ”time: time a simple command,” Commands & Utilities Reference, The Single UNIX Specification, Issue 7 from The Open Group, http://pubs.opengroup.org/onlinepubs/9699919799/utilities/time.html .

[85] B. Gough, “An Introduction to GCC - for the GNU compilers gcc and g++,” Network Theory Ltd., March 2004.

[86] ”Open MPI Documentation,” http://www.open-mpi.org/doc/ .

[87] OSU Micro Benchmarks 3.5, Ohio State University, Network-Based Computing Laboratory, http://mvapich.cse.ohio-state.edu/benchmarks/ .

[88] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, Ohio State University, Network-Based Computing Laboratory, http://mvapich.cse.ohio-state.edu .

[89] O. Pentakalos, ”An Introduction to the InfiniBand Architecture,” Feb. 2002, http://www.oreillynet.com/pub/a/network/2002/02/04/windows.html

[90] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, P. Wyckoff, D. K. Panda, “Micro-Benchmark Performance Comparison of High-Speed Cluster Interconnects,” IEEE Micro, January/February, 2004.

[91] D. Strippgen, K. Nagel, “Using common graphics hardware for multi-agent traffic simulation with CUDA,” in Proceedings of the 2nd International Conference on Simulation Tools and Techniques, March 2009.

[92] ”OpenCL - The open standard for parallel programming of heterogeneous systems,” http://www.khronos.org/opencl/

[93] “What is CUDA,” http://developer.nvidia.com/what-cuda

A Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
CMAC	Control-plane Media Access Control
CQI	Channel Quality Indication
DCI	Data Control Indication
DES	Discrete Event Simulation
DSP	Digital Signal Processing
eNB	eNodeB
EPC	Evolved Packet Core
EPS	Evolved Packet System
E-UTRAN	Evolved UMTS Terrestrial Radio Access Network
FPGA	Field-Programmable Gate Array
GBR	Guaranteed Bit Rate
GPRS	General Packet Radio System
GTP	GPRS Tunneling Protocol
HO	Handover
HPC	High-Performance Computing
HSPA	High Speed Packet Access
HSS	Home Subscriber Server
ICIC	Inter-Cell Interference Coordination
ICMP	Internet Control Message Protocol
IE	Information Element
IMS	IP Multimedia System
IP	Internet Protocol
LBTS	Lower Bound Timestamp
LCID	Logical Channel Identifier
LP	Logical Process
LTE	Long Term Evolution
LXC	Linux Container
MAC	Media Access Control (layer)
MBR	Maximum Bit Rate
MCS	Modulation and Coding Scheme
MME	Mobility Management Entity
MPI	Message Passing Interface
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
PCC	Policy and Charging Control
PCRF	Policy and Charging Rules Function
PDCCH	Physical Downlink Control CHannel
PDCP	Packet Data Convergence Protocol
PDN	Packet Data Network
PDSCH	Physical Downlink Shared CHannel
PDU	Protocol Data Unit
PF	Proportional Fair (scheduler)
P-GW	PDN Gateway
PHY	Physical (layer)
PMIP	Proxy Mobile IP
PRACH	Physical Random Access CHannel
PSD	Power Spectral Density
PSTN	Public Switched Telephone Network
PUCCH	Physical Uplink Control CHannel
PUSCH	Physical Uplink Shared CHannel
QoS	Quality of Service
QCI	QoS Class Identifier
RAN	Radio Access Network
RAT	Radio Access Technology
RB	Radio Bearer
RB	Resource Block
RBC	Radio Bearer Control
RBG	Resource Block Group
RE	Resource Element
RLC	Radio Link Control (layer)
RNC	Radio Network Controller
RR	Round Robin (scheduler)
RRC	Radio Resource Control (layer)
RRM	Radio Resource Management
RT	Real Time
RTT	Round-Trip Time
SAE	System Architecture Evolution

SAP	Service Access Point
SC-FDMA	Single Carrier Frequency Division Multiple Access
SDU	Service Data Unit
S-GW	Serving Gateway
SINR	Signal to Interference-Noise Ratio
SMP	Symmetric Multiprocessor
SNR	Signal to Noise Ratio
TA	Tracking Area
TB	Transport Block
TCP	Transmission Control Protocol
TEID	Tunnel Endpoint Identifier
TFT	Traffic Flow Template
TS	Time Stamp
TTI	Transmission Time Interval
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
VM	Virtual Machine

B Figures

B.1 LTE/SAE Supplementary Figures

B.2 Core Simulation Implementation Figures

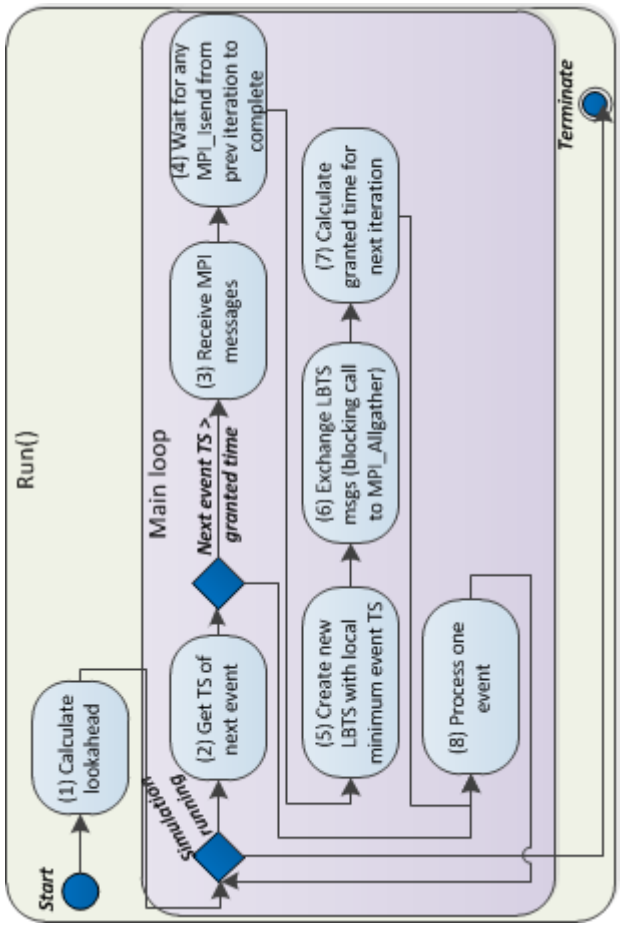


Figure B.1: Distributed simulator implementation: Top-level activity diagram

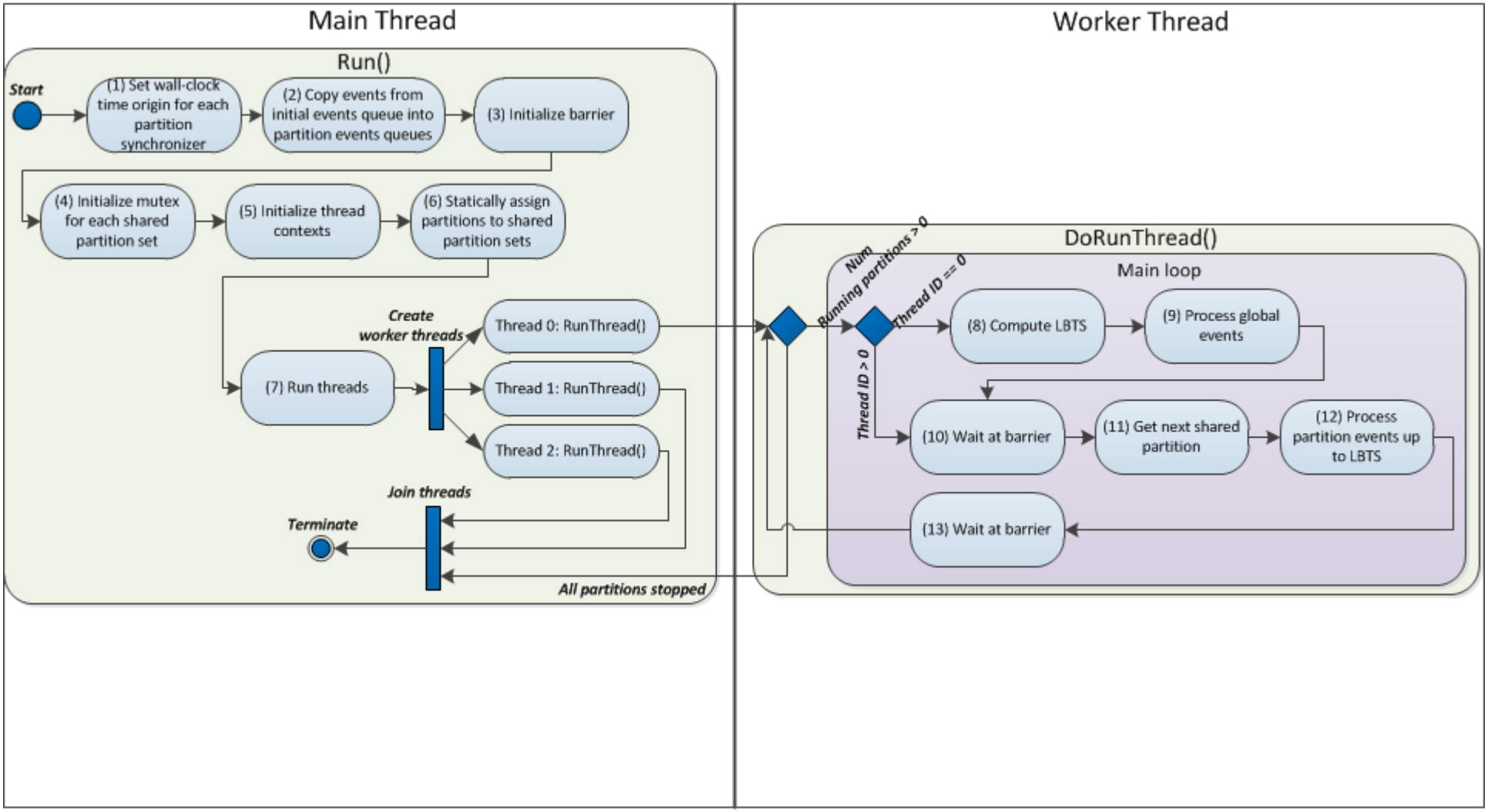


Figure B.2: Multithreaded simulator implementation: Top-level activity diagram

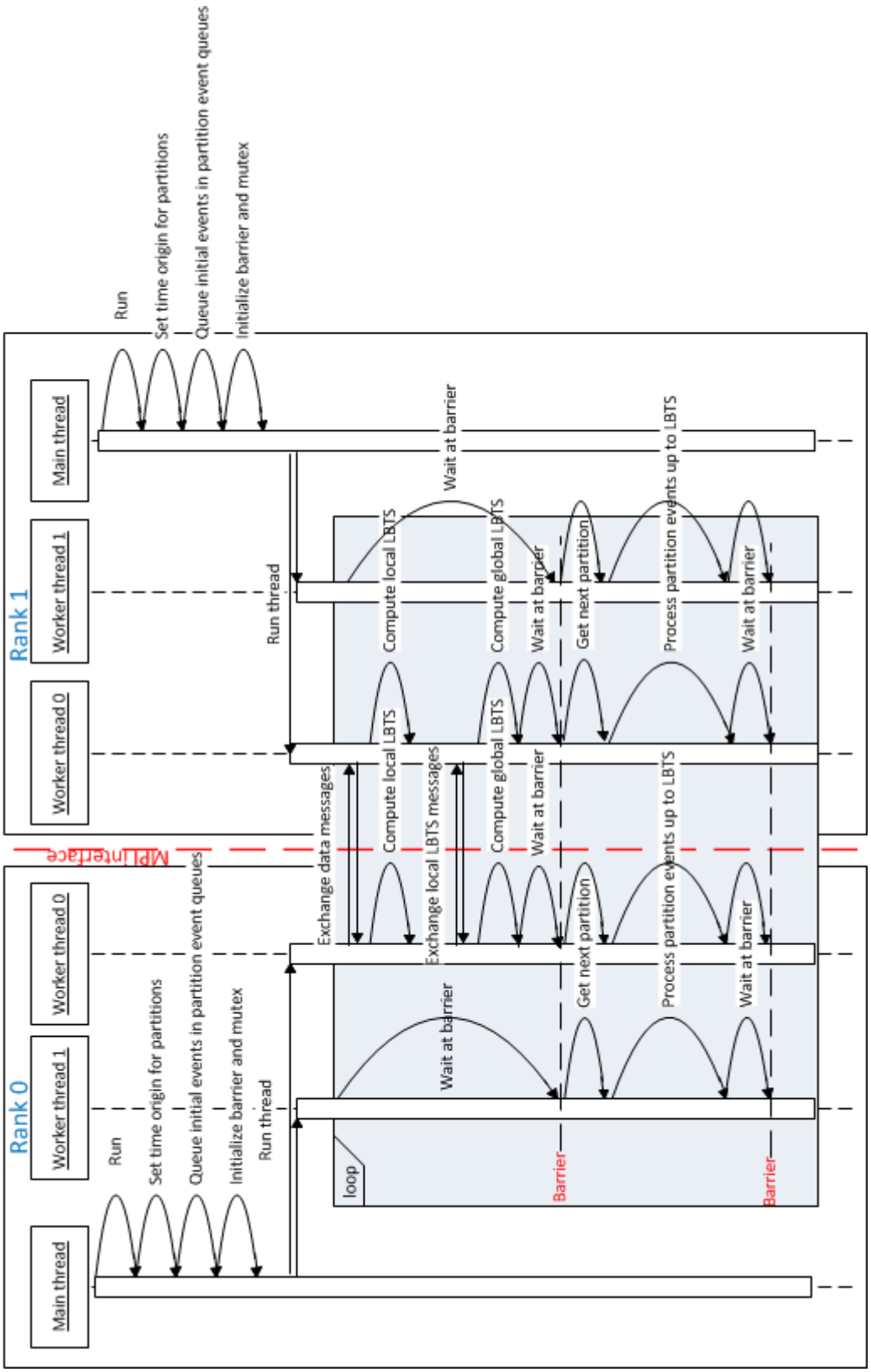


Figure B.3: Real-time hybrid parallel simulator implementation: Top-level sequence diagram

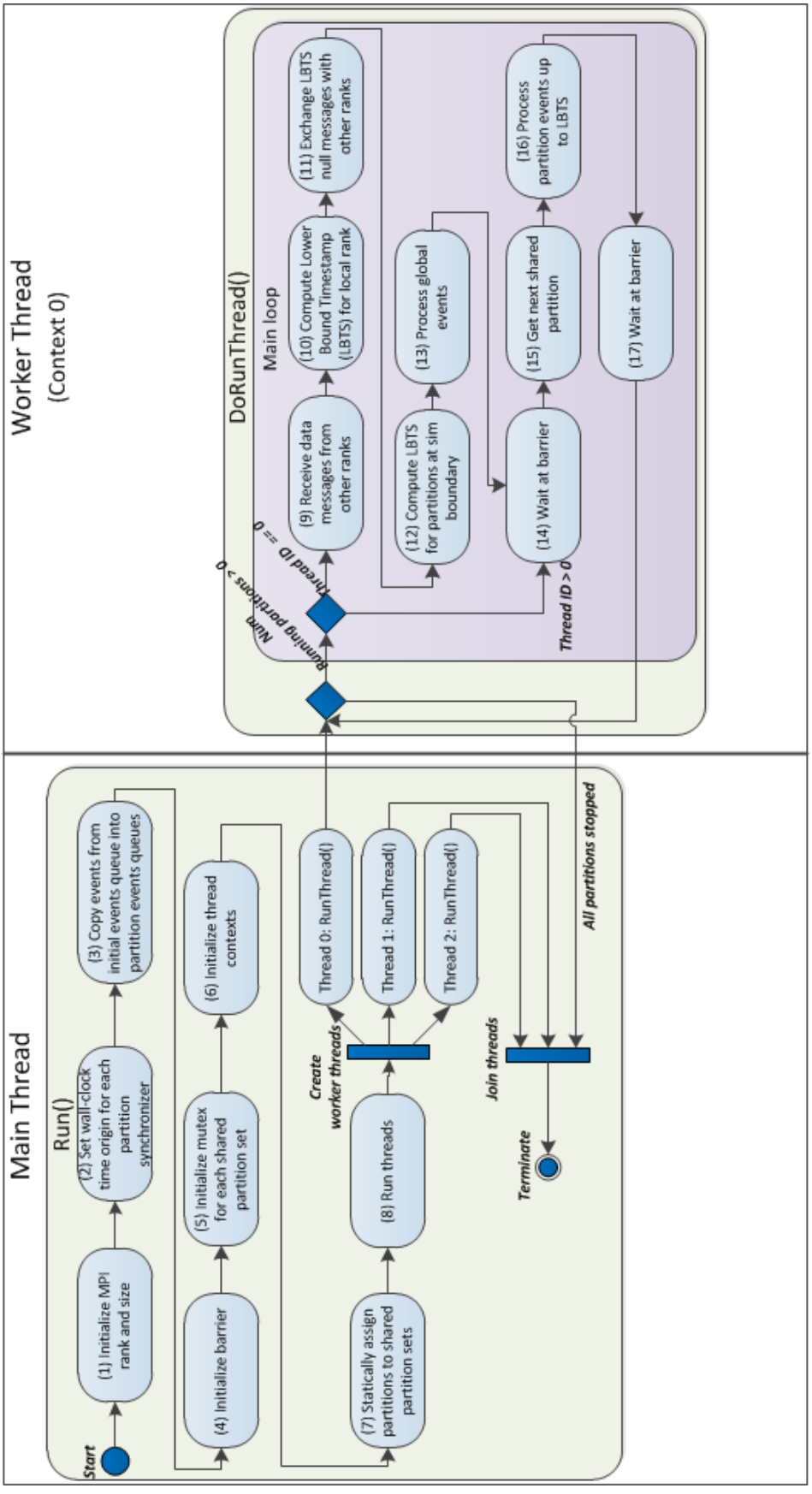


Figure B.4: Real-time hybrid parallel simulator implementation: Top-level activity diagram

B.3 LTE Model Implementation

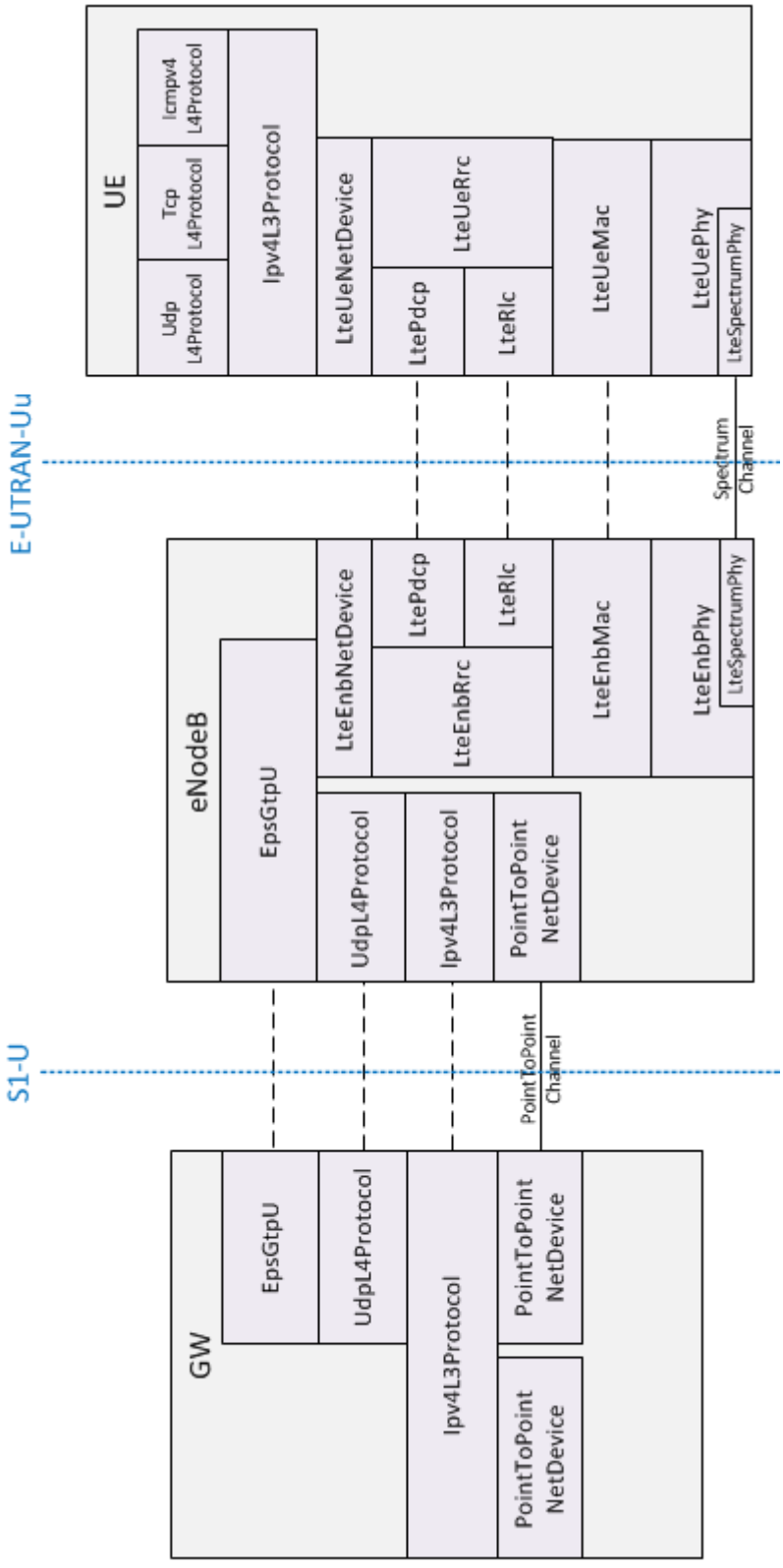


Figure B.5: ns-3 representations of network entities and protocols

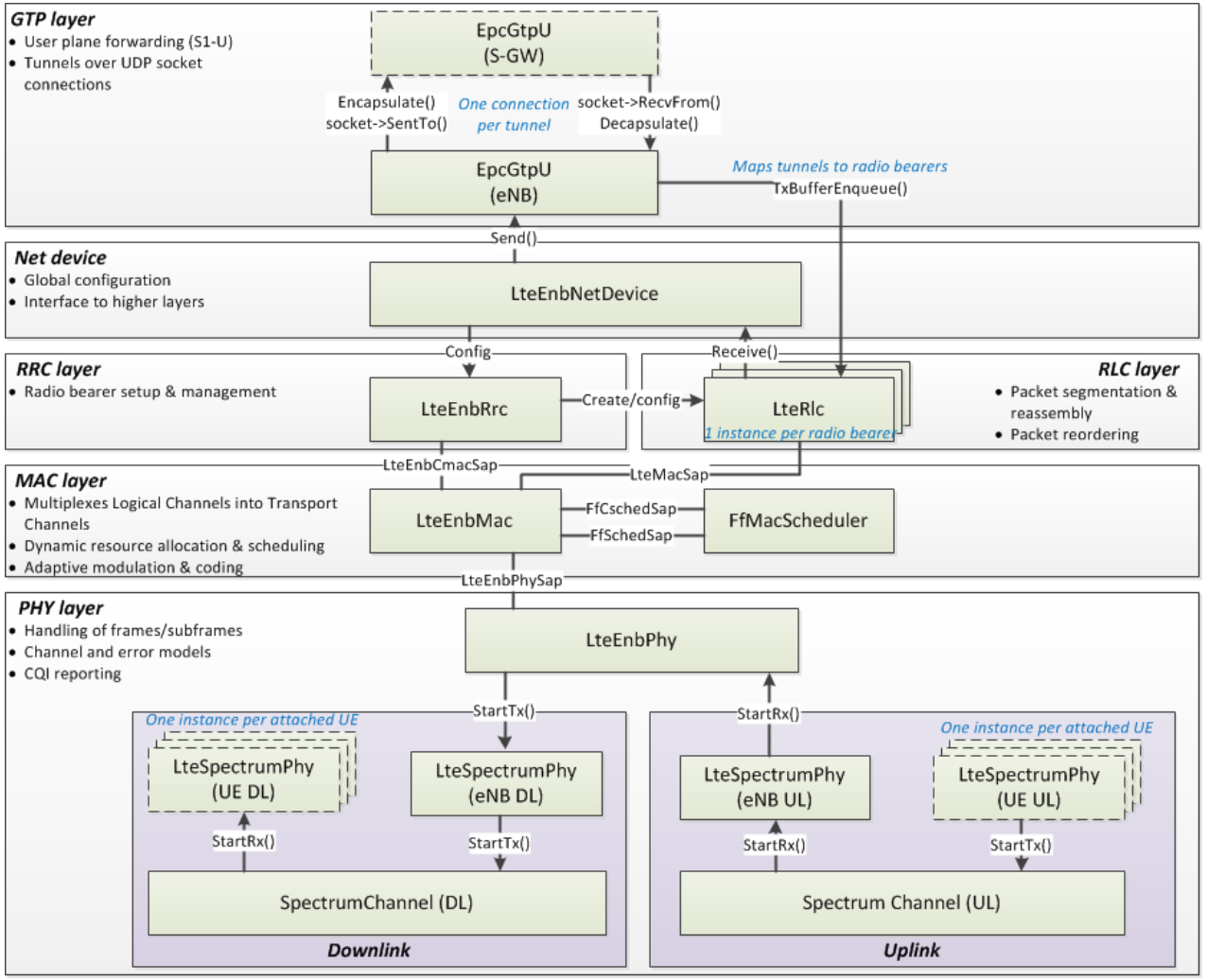


Figure B.6: eNodeB module components (based on the CTTC LENA project [79]).

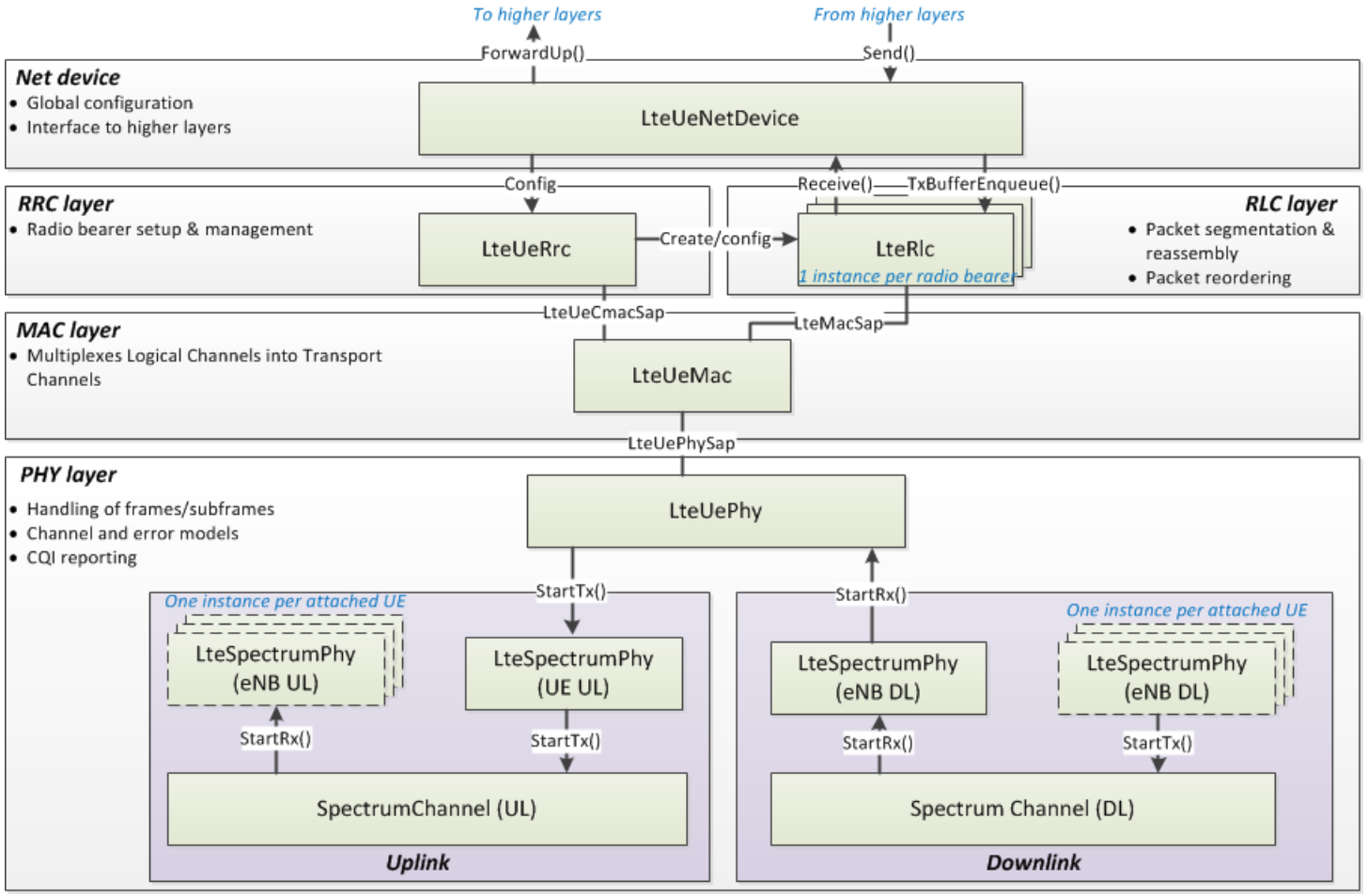


Figure B.7: UE module components (based on the CTTC LENA project [79]).

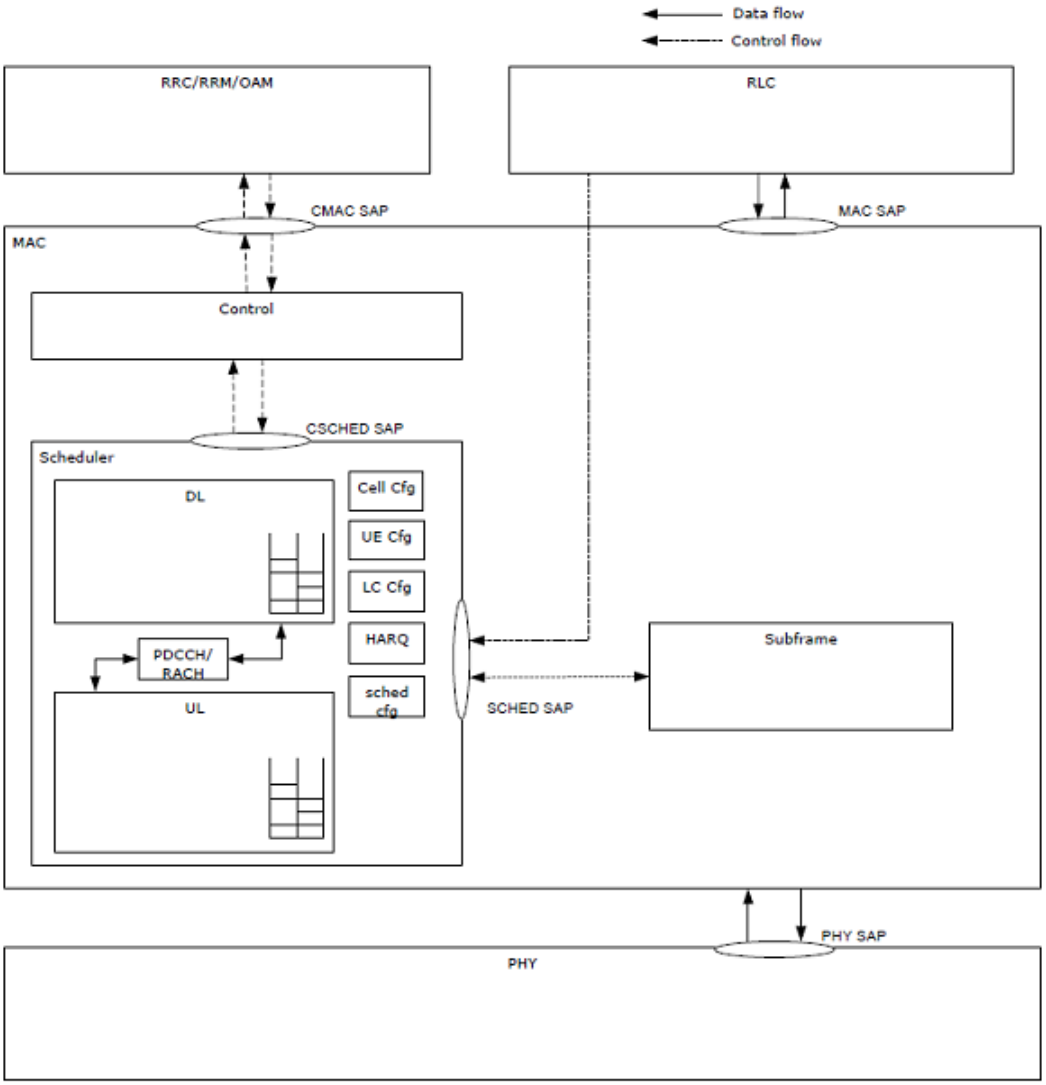


Figure B.8: Femto Forum MAC Scheduler Interface (taken from [59]).

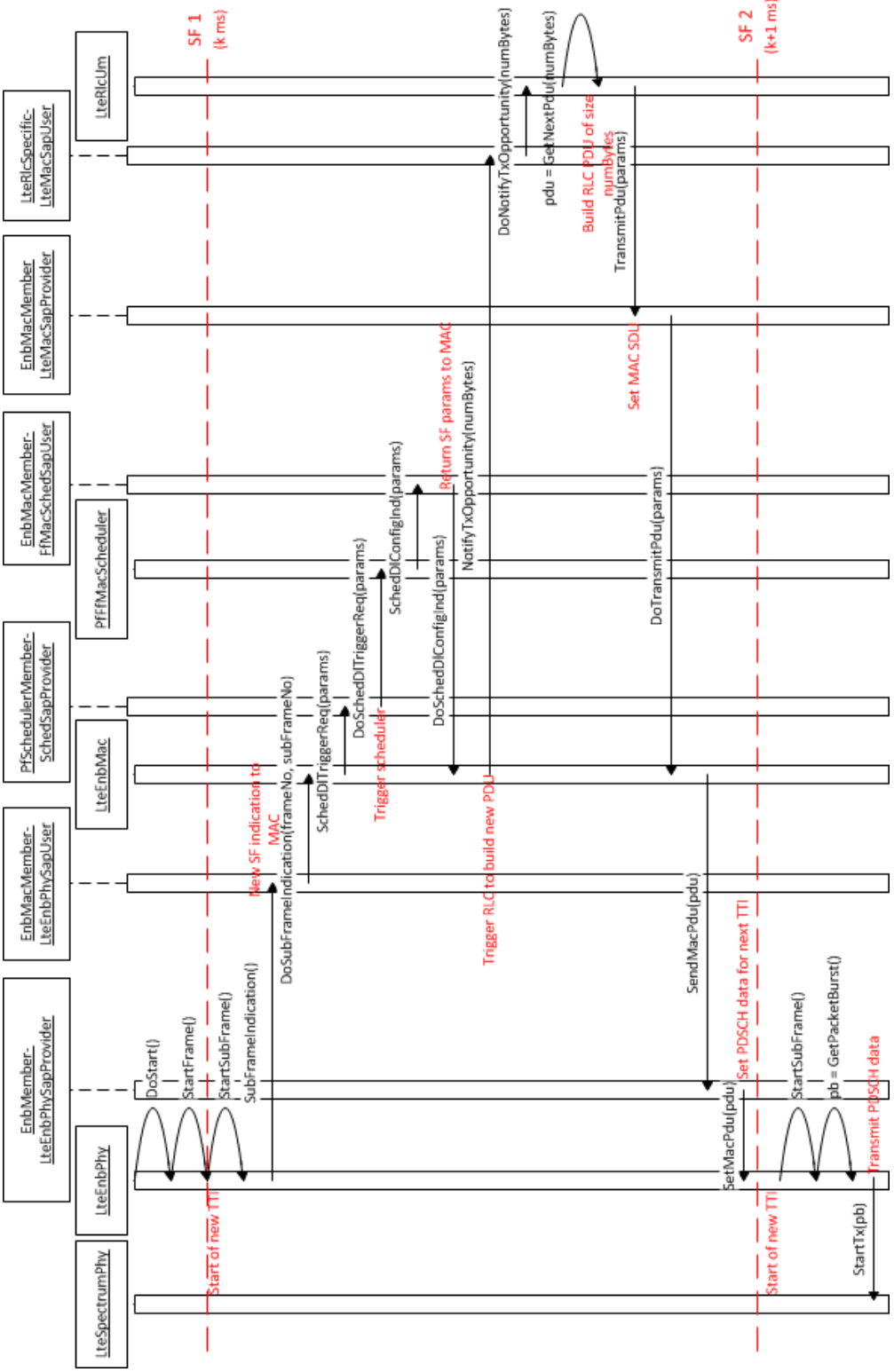


Figure B.9: Downlink subframe triggering

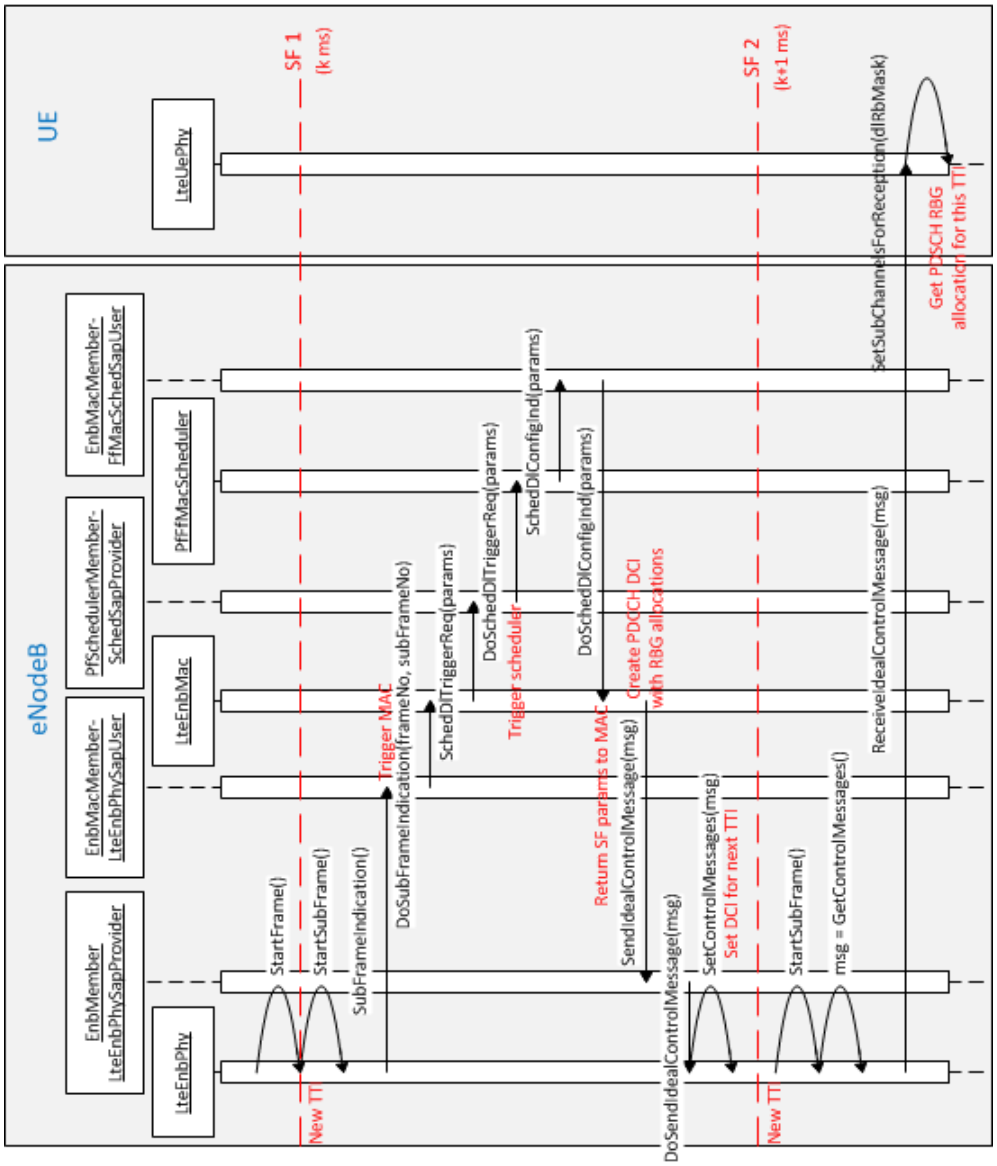


Figure B.10: Downlink Data Control Indication sent from eNodeB to UE

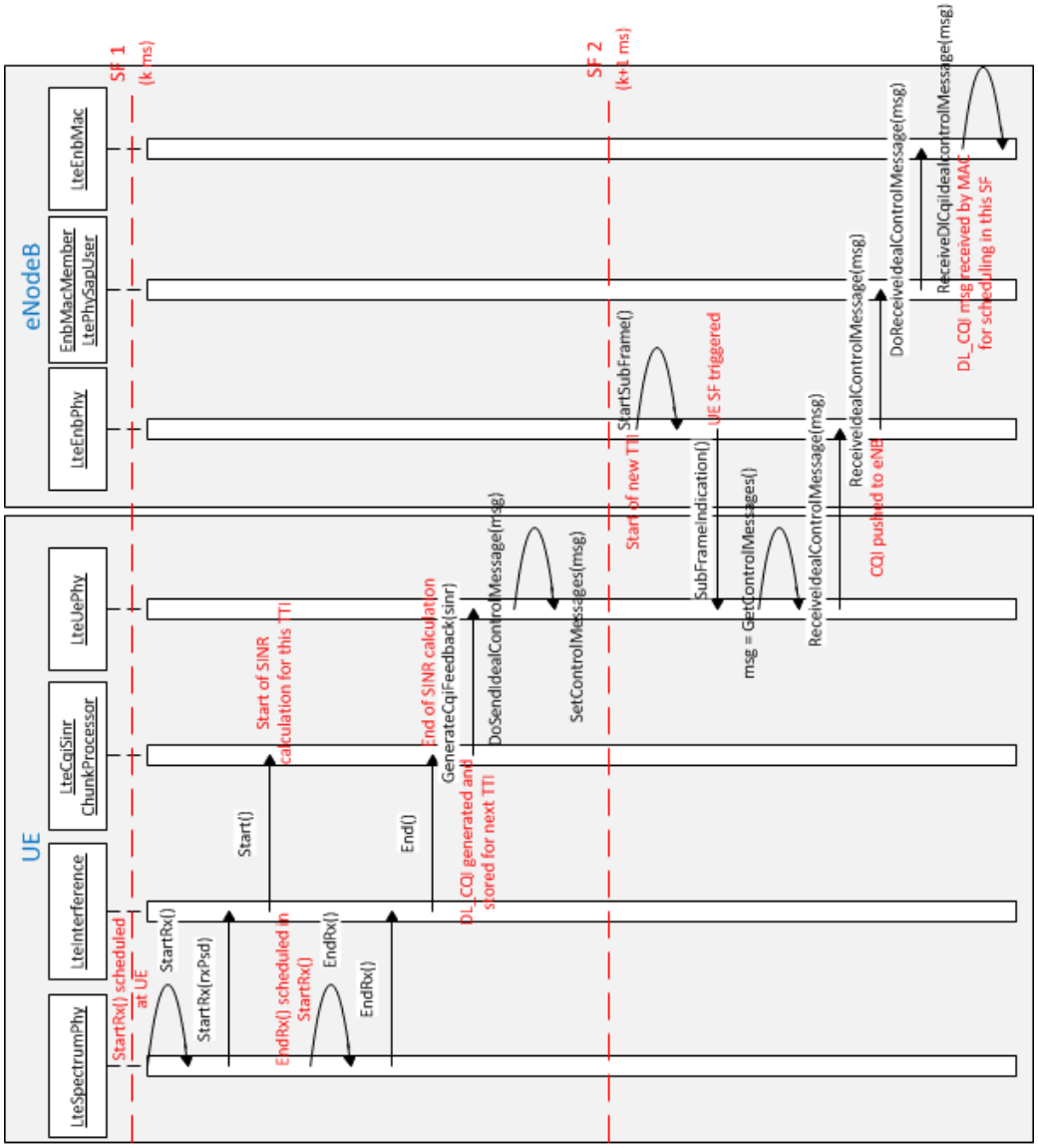


Figure B.11: Downlink CQI feedback reporting from UE to eNodeB

B.4 Results from Core Simulator Implementation Performance Tests

Figure B.12: Non-real time, distributed bottleneck test (num nodes=128)

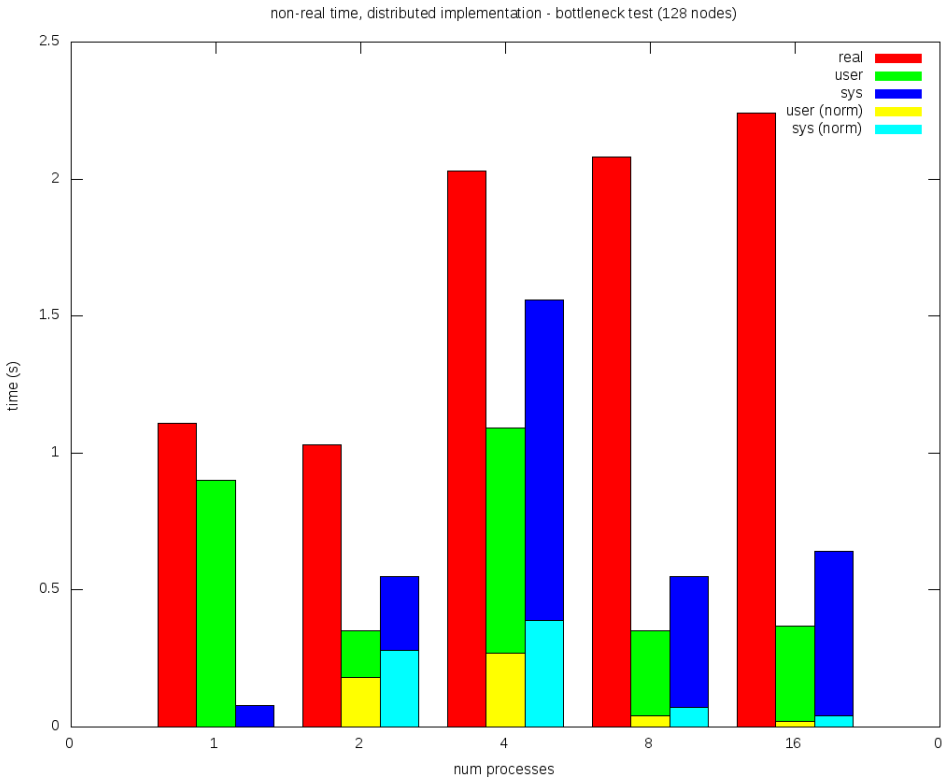


Figure B.13: Non-real time, distributed bottleneck test (num nodes=256)

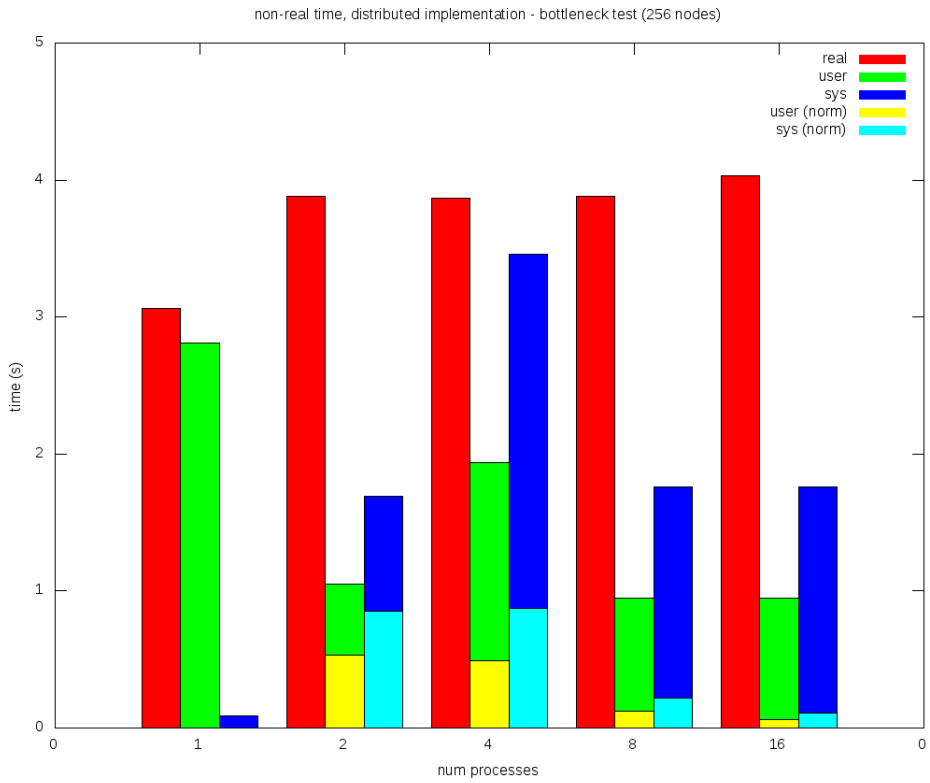


Figure B.14: Non-real time, distributed bottleneck test (num nodes=1024)

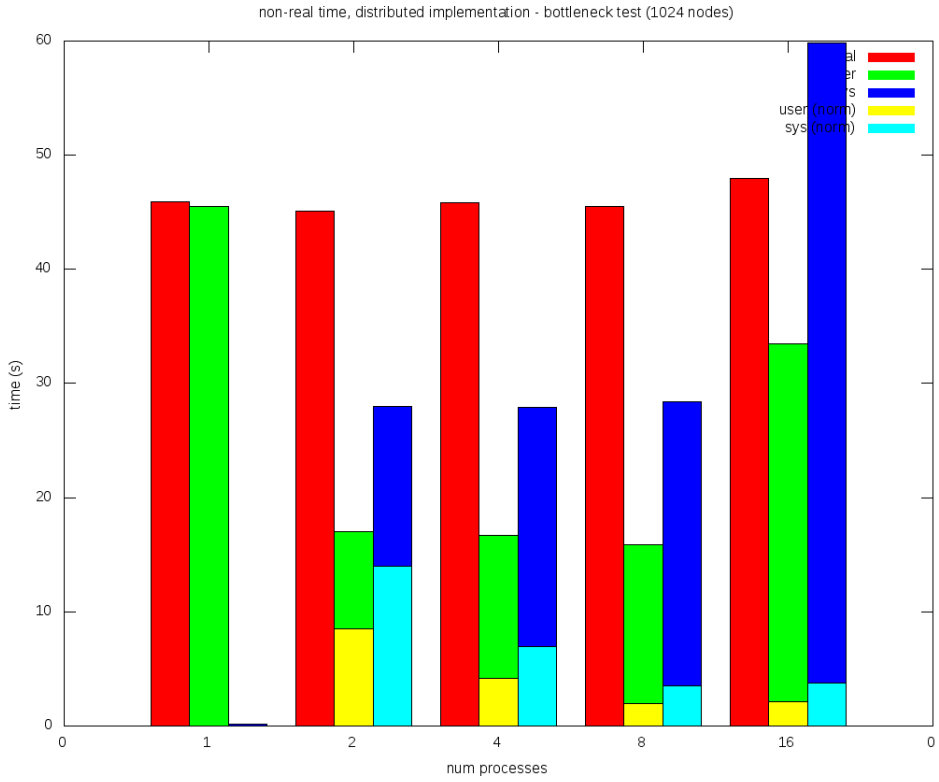


Figure B.15: Non-real time, distributed embarrassingly parallel test (num nodes=128)

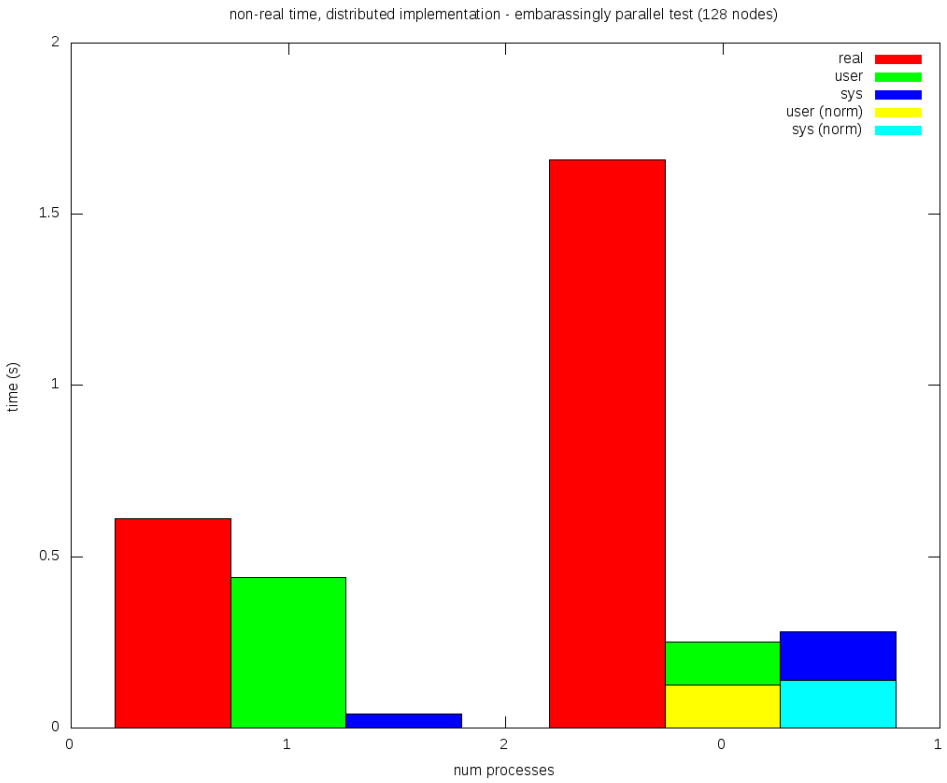


Figure B.16: Non-real time, distributed embarrassingly parallel test (num nodes=256)

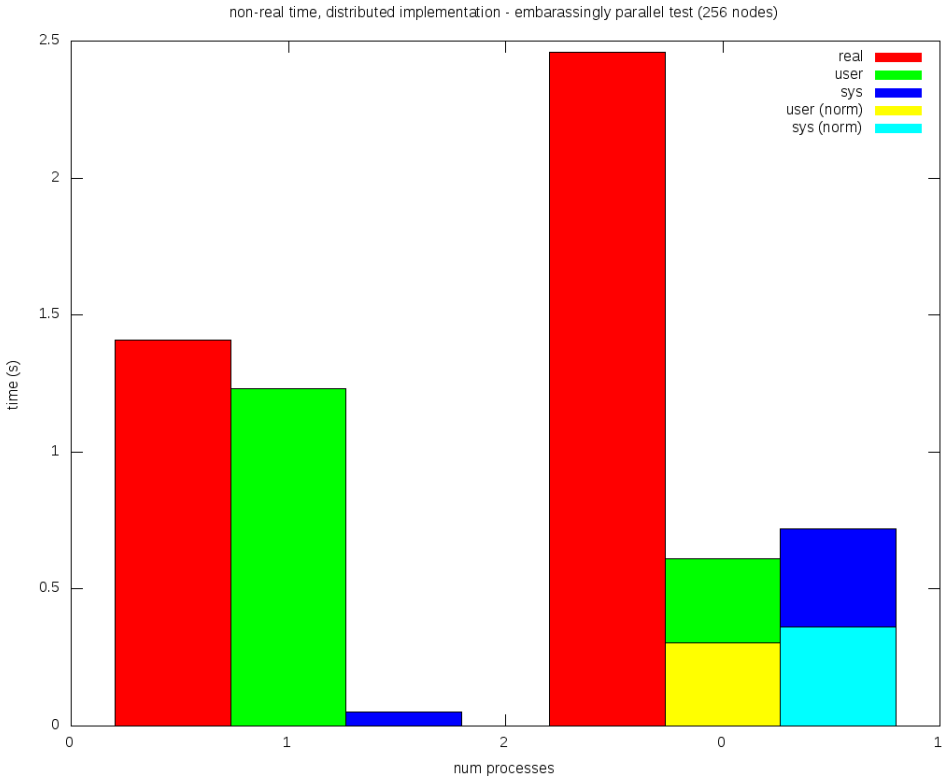


Figure B.17: Non-real time, distributed embarrassingly parallel test (num nodes=1024)

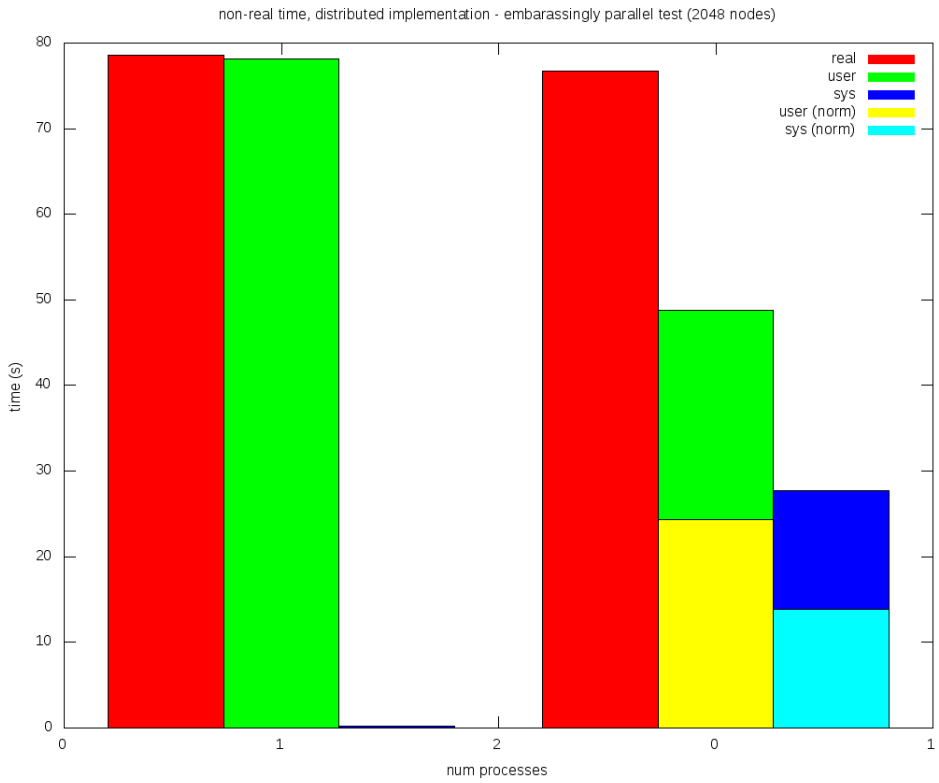


Figure B.18: Non-real time, multithreaded bottleneck test (num nodes=128)

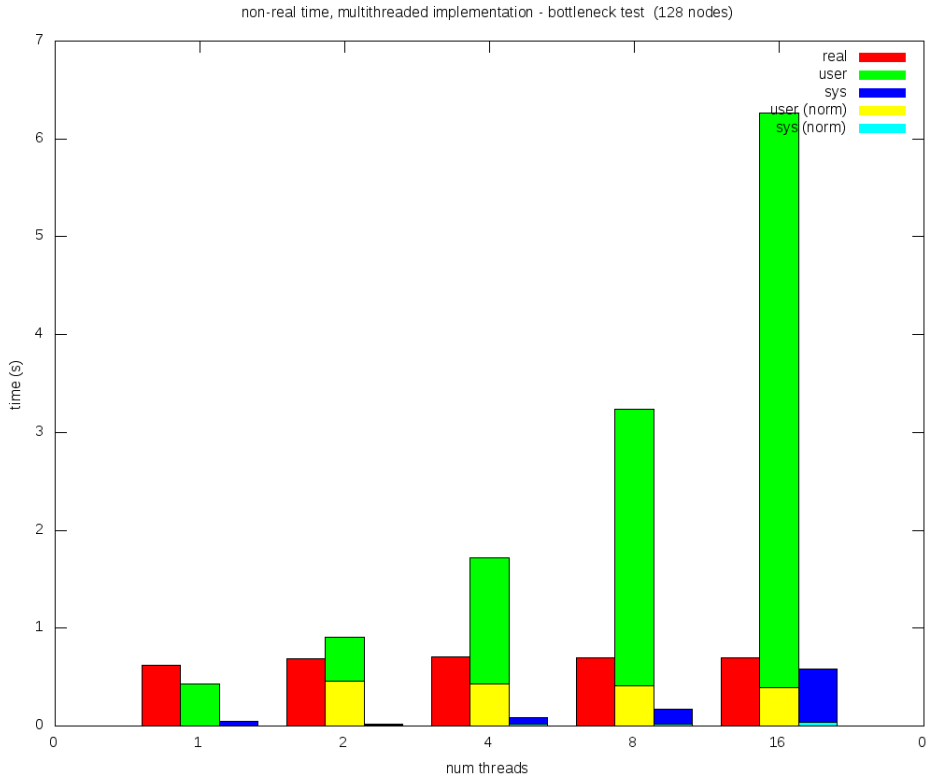


Figure B.19: Non-real time, multithreaded bottleneck test (num nodes=256)

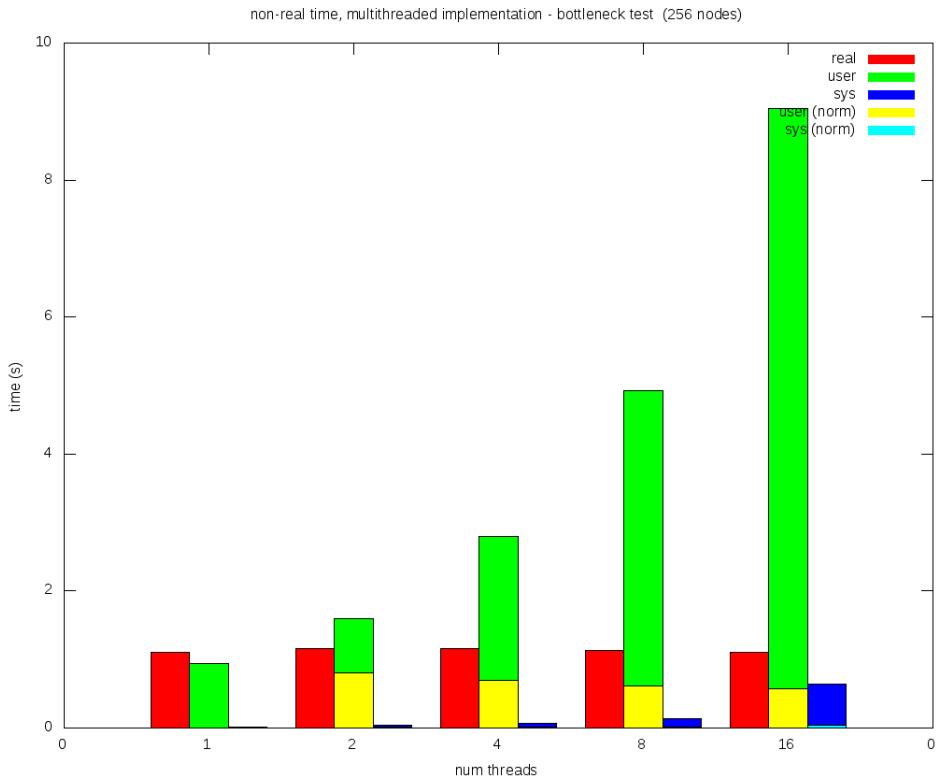
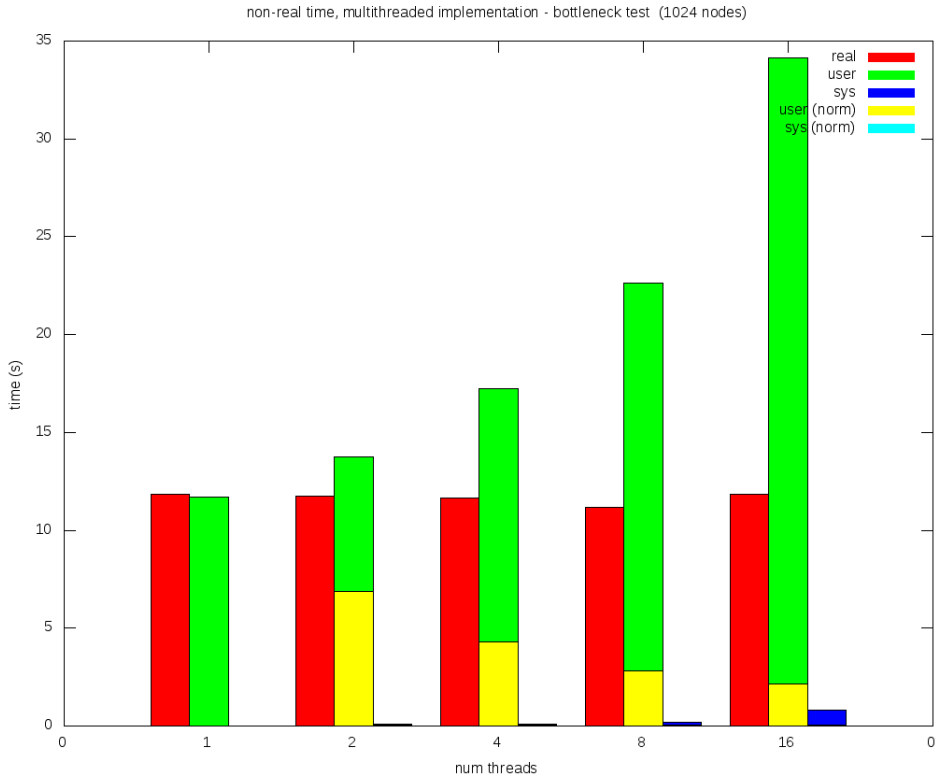


Figure B.20: Non-real time, multithreaded bottleneck test (num nodes=1024)



B.5 Results from LTE Simulation Test Cases

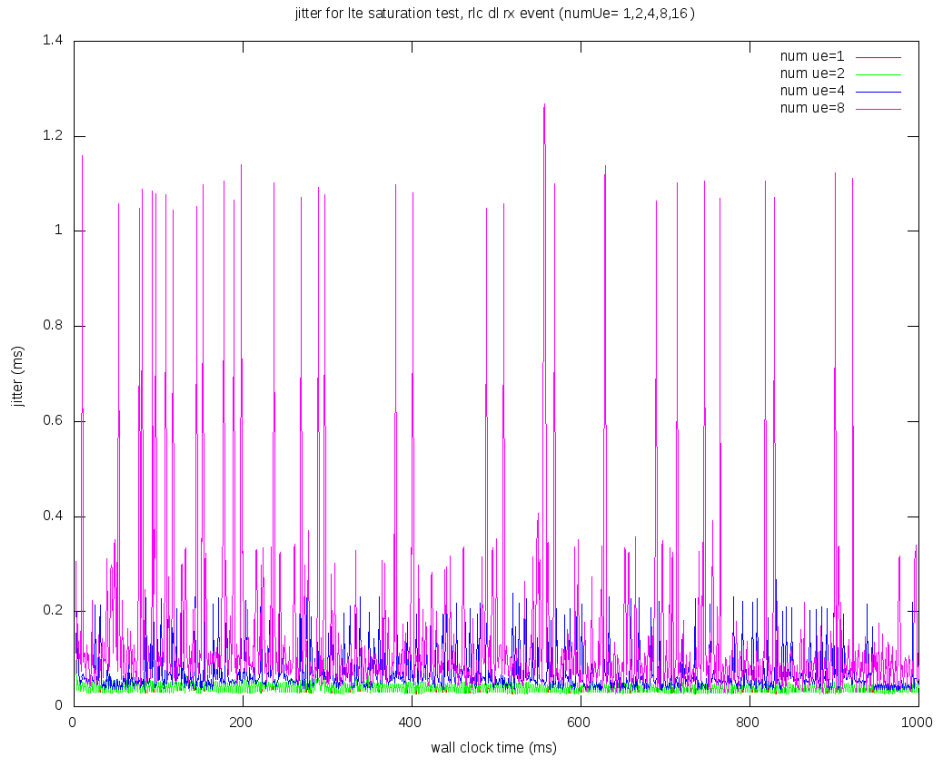


Figure B.21: LTE saturation test case: Timing jitter for DL RLC RX events at UE (numUe={1,2,4,8})

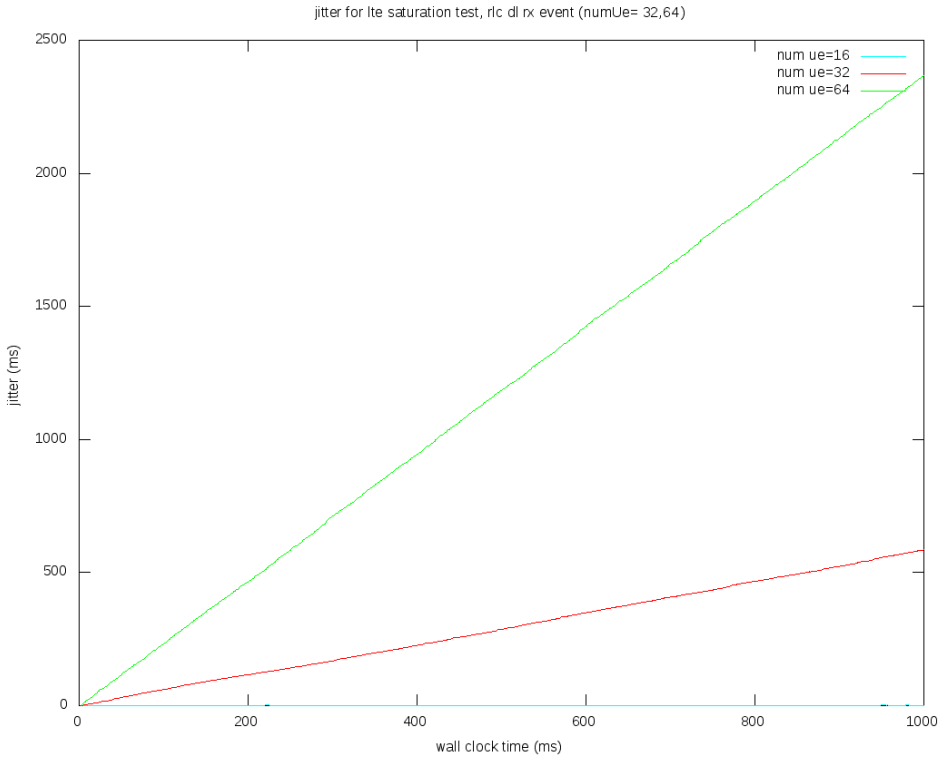


Figure B.22: LTE saturation test case: Timing jitter for DL RLC RX events at UE (numUe={16,32,64})

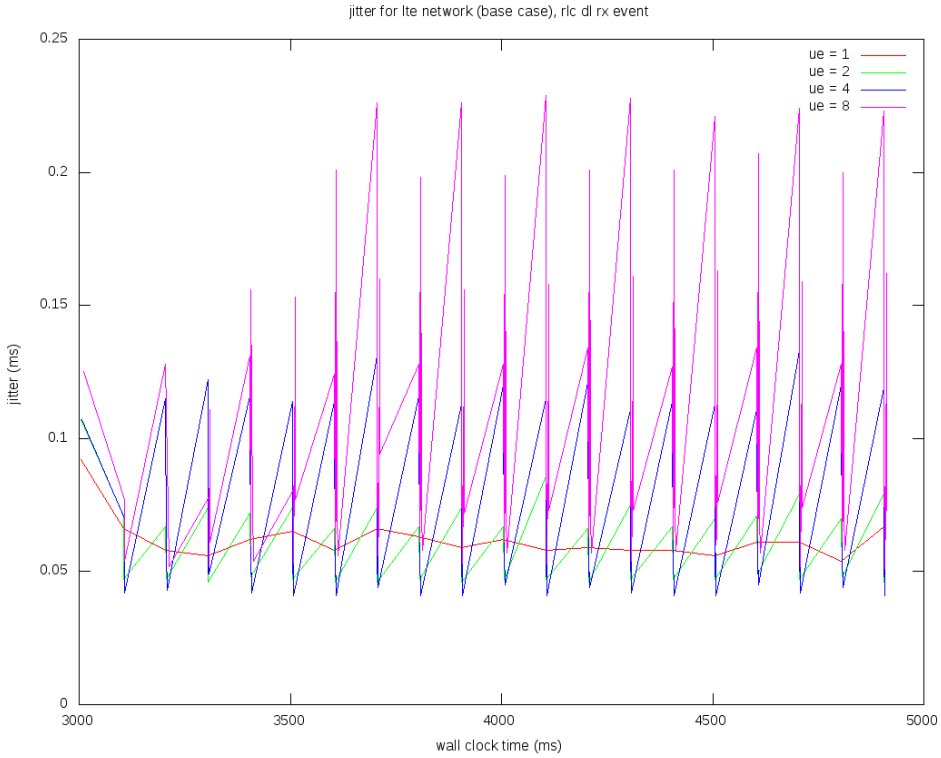


Figure B.23: LTE simple traffic generator test case: Timing jitter for DL RLC RX events at UE (numUe={1,2,4,8})

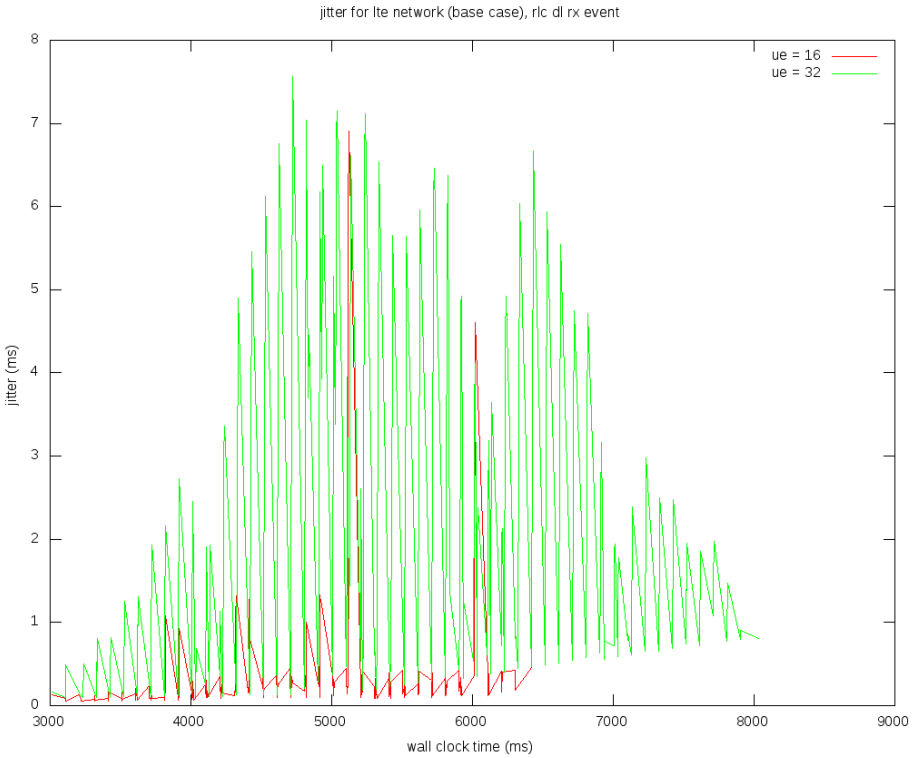


Figure B.24: LTE simple traffic generator test case: Timing jitter for DL RLC RX events at UE (numUe={16,32})

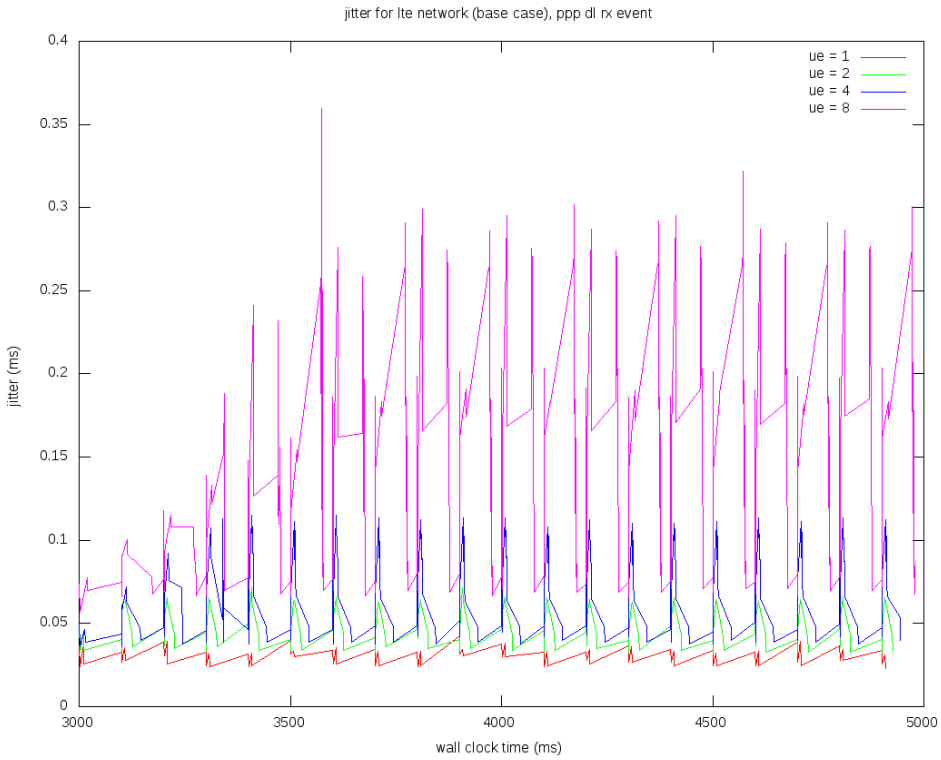


Figure B.25: LTE simple traffic generator test case: Timing jitter for DL PPP RX events at nodes 0 and 1 (server and GW)

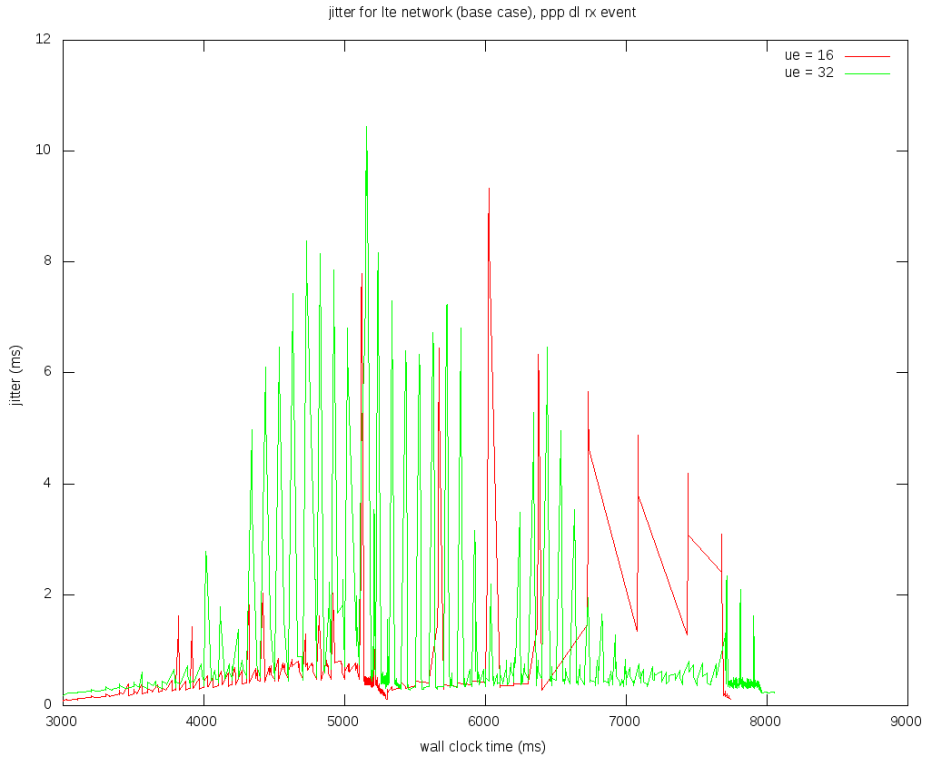


Figure B.26: LTE simple traffic generator test case: Timing jitter for DL PPP RX events at node 2 (eNodeB)

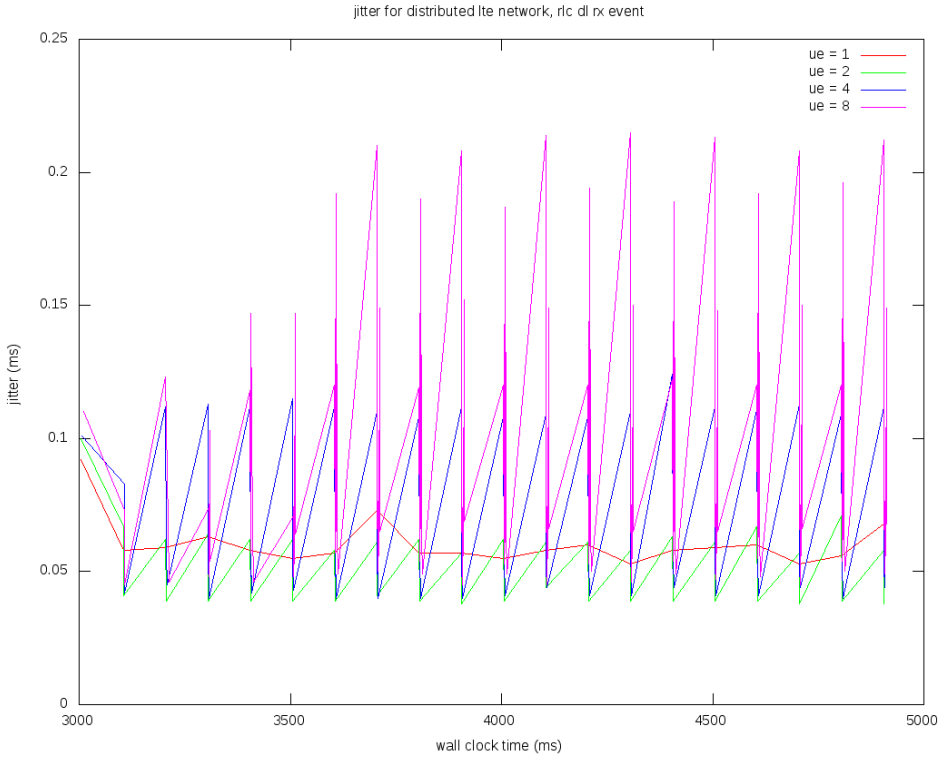


Figure B.27: LTE distributed network test case (intranode): Timing jitter for DL RLC RX events at UE (numUe={1,2,4,8})

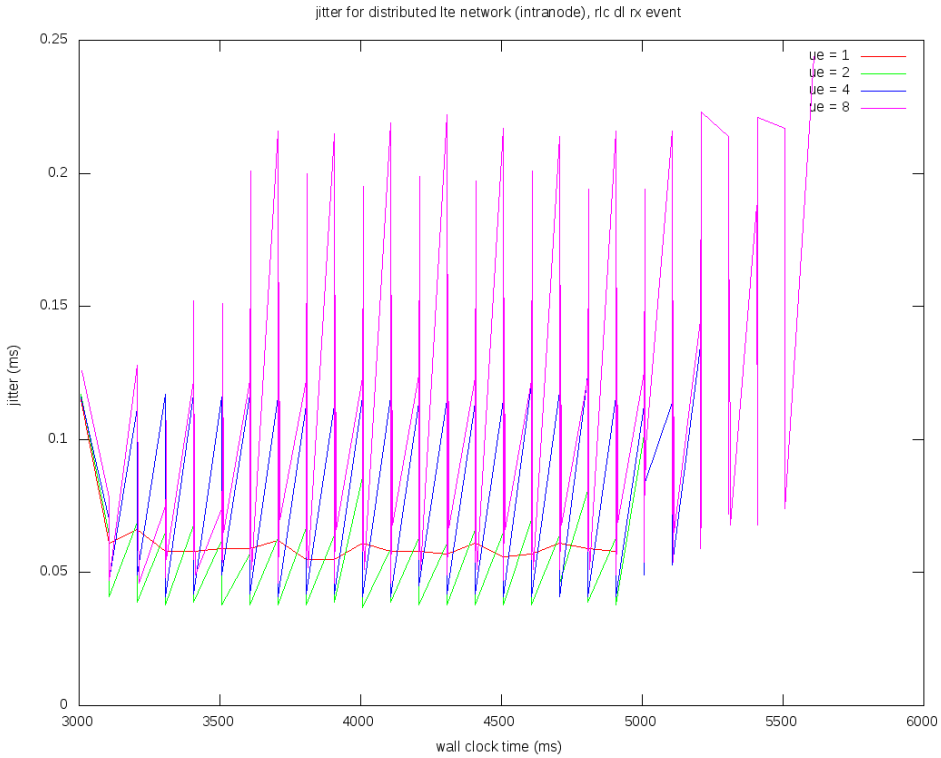


Figure B.28: LTE distributed network test case (internode): Timing jitter for DL RLC RX events at UE (numUe={1,2,4,8})

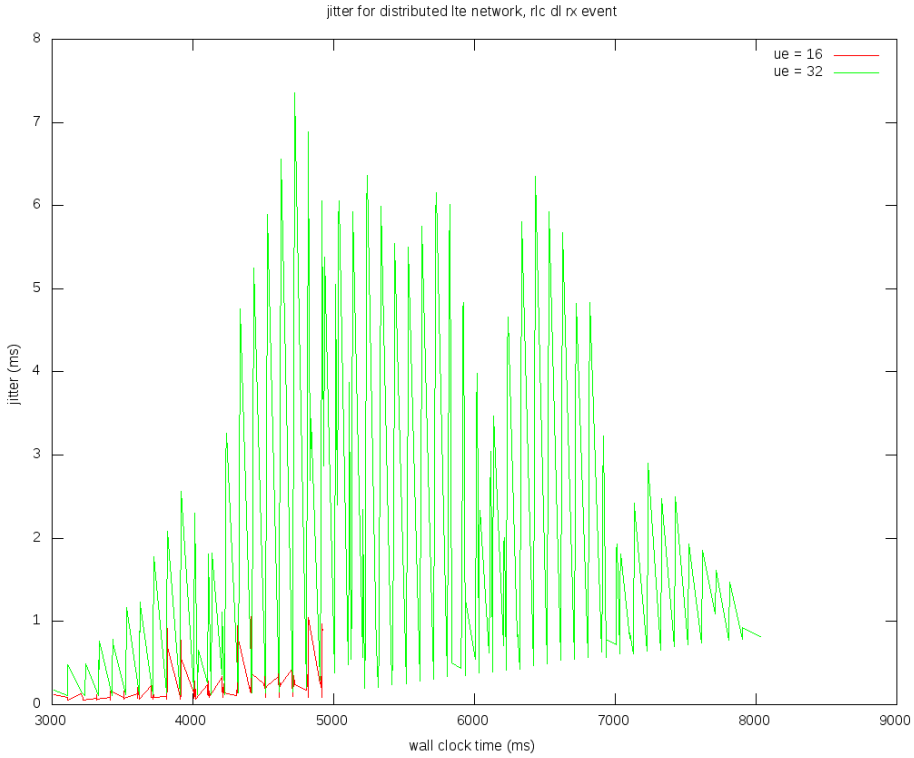


Figure B.29: LTE distributed network test case (intranode): Timing jitter for DL RLC RX events at UE (numUe={16,32})

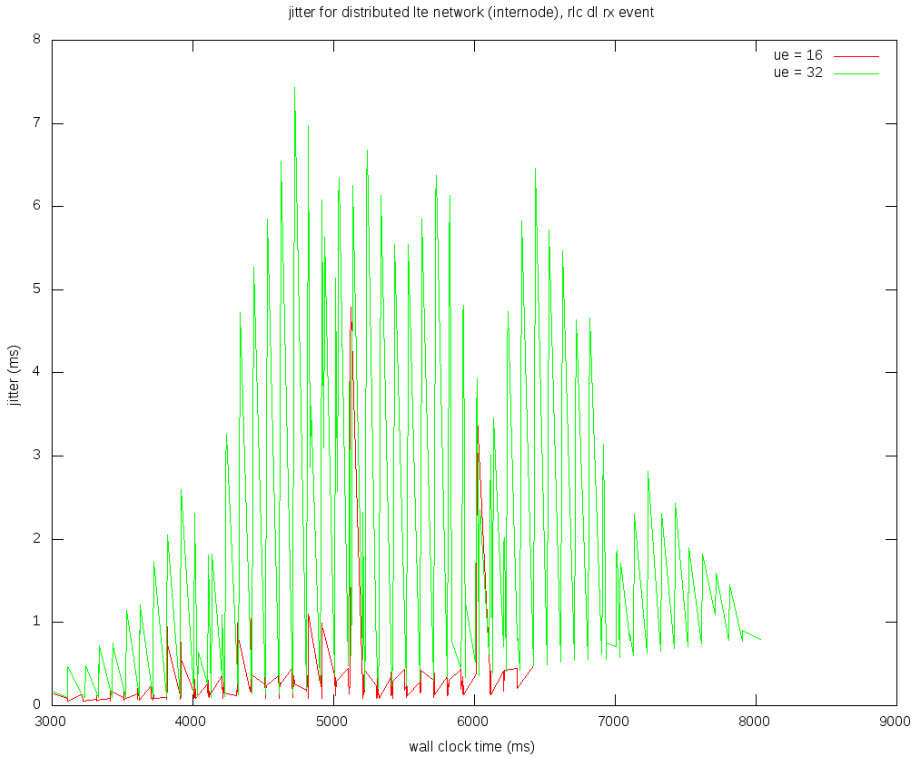


Figure B.30: LTE distributed network test case (internode): Timing jitter for DL RLC RX events at UE (numUe={16,32})

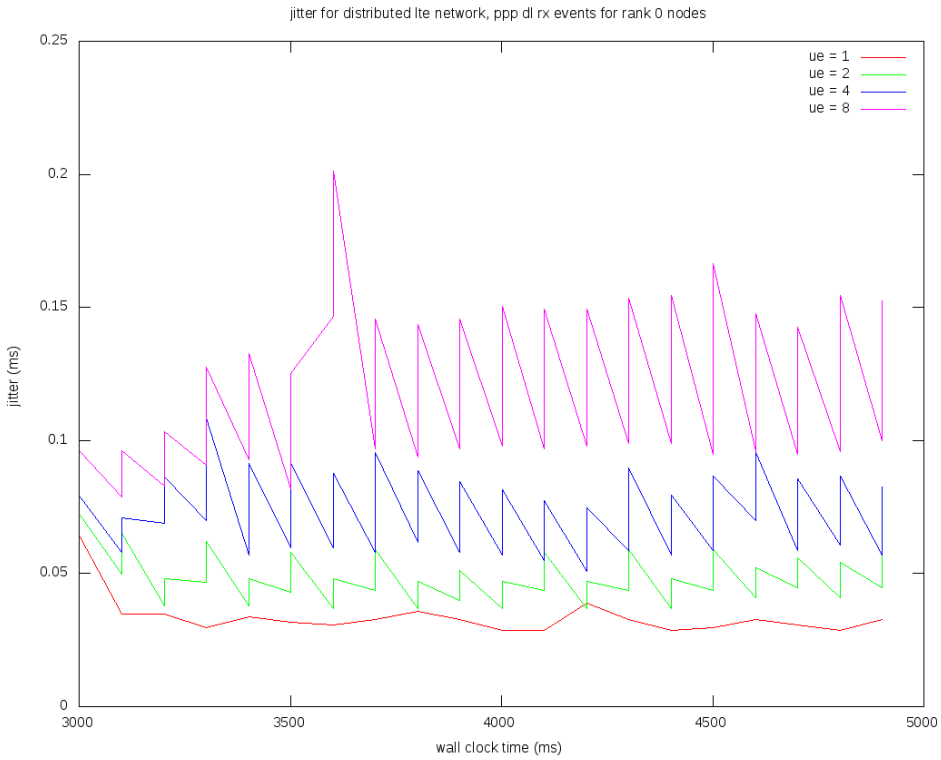


Figure B.31: LTE distributed network test case (intranode): Timing jitter for DL PPP RX events at node 2 (rank 1, numUe={1,2,4,8})

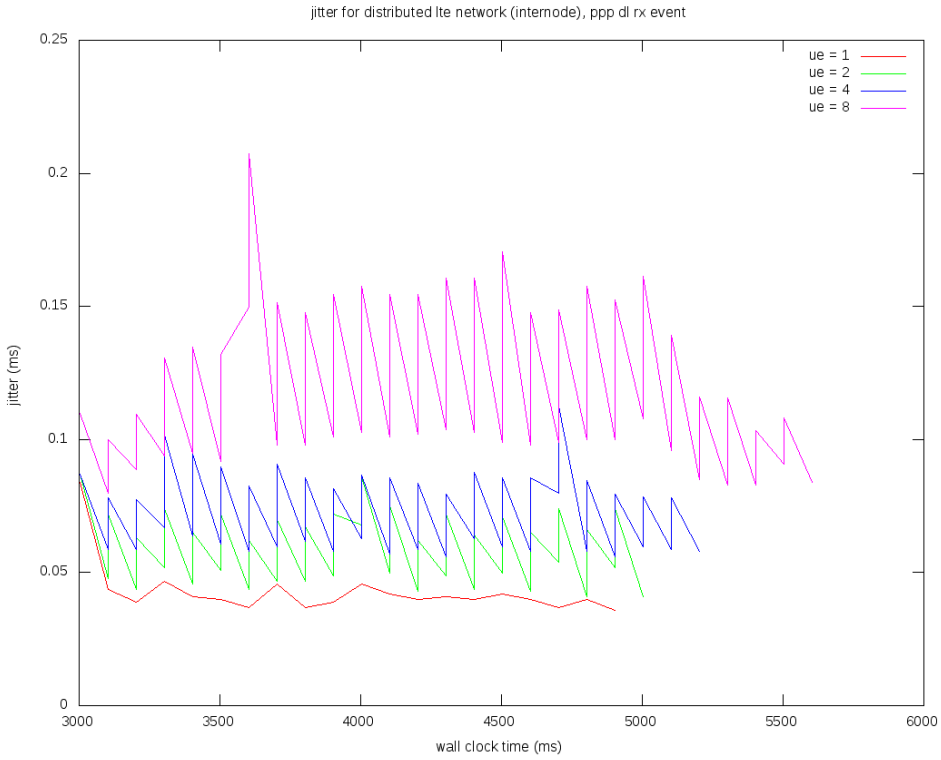


Figure B.32: LTE distributed network test case (internode): Timing jitter for DL PPP RX events at node 2 (rank 1, numUe={1,2,4,8})

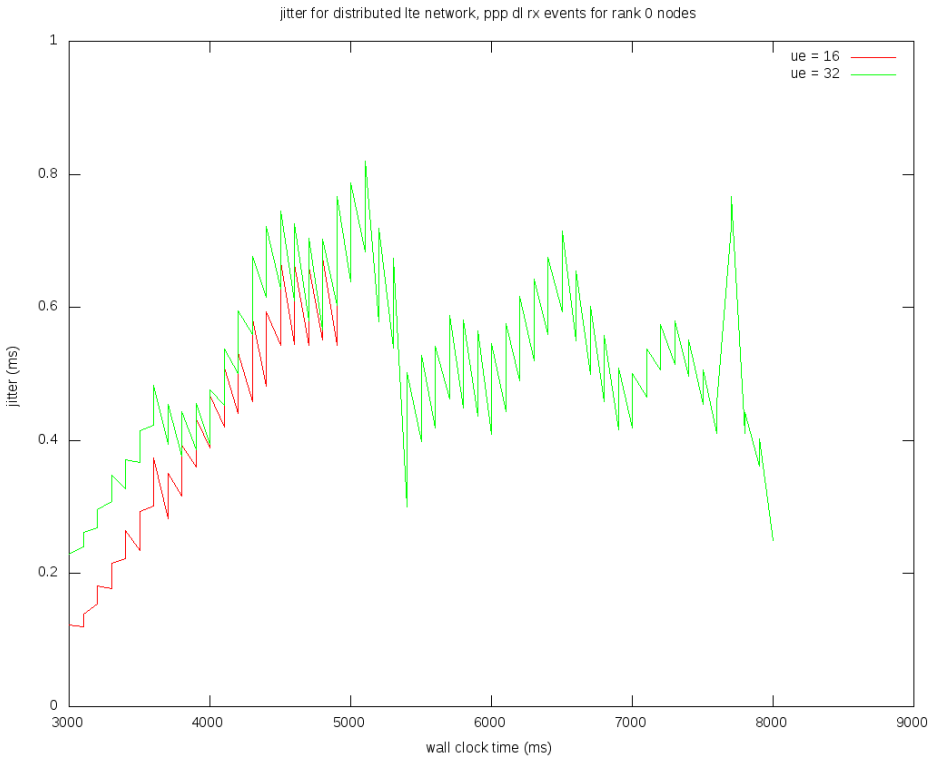


Figure B.33: LTE distributed network test case (intranode): Timing jitter for DL PPP RX events at node 2 (numUe={16,32})

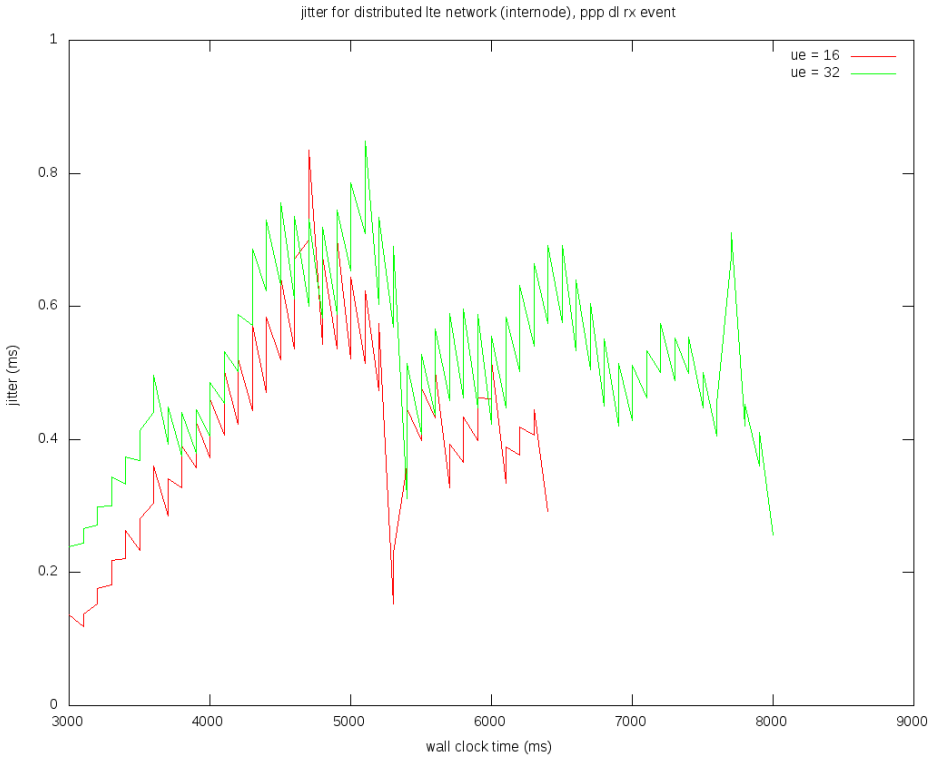


Figure B.34: LTE distributed network test case (internode): Timing jitter for DL PPP RX events at node 2 (numUe={16,32})

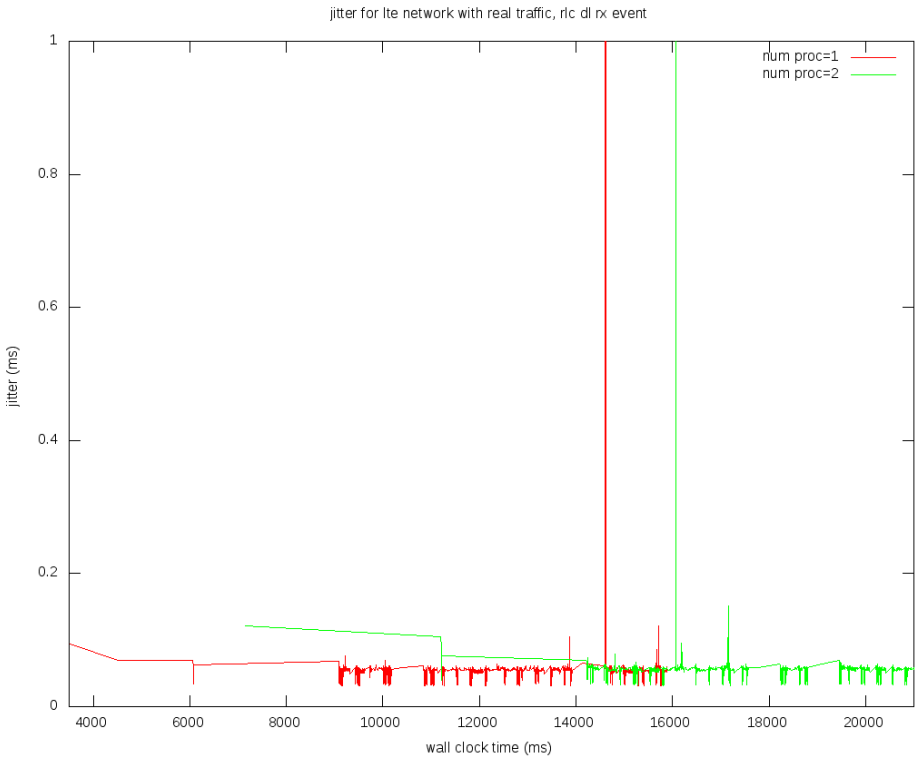


Figure B.35: LTE real traffic test case (): Timing jitter for DL RLC RX events at UE (node 3) (num proc=1,2)

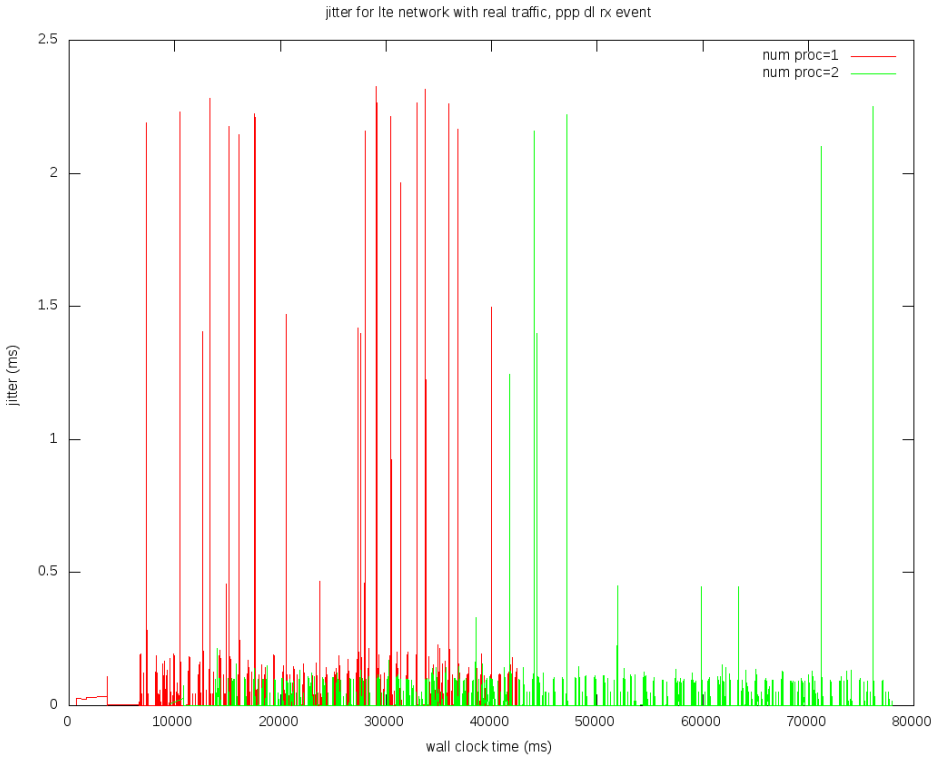


Figure B.36: LTE real traffic test case (): Timing jitter for DL PPP RX events at nodes 0,1,2 (num proc=1,2)

C Tables

C.1 Simulator Platform Implementation - API

Table C.1: Useful `DefaultSimulatorImpl` public methods

Method Name	Arguments	Return Type	Description
`Run`	`void`	`void`	Initiates the simulation.
`Stop`	`Time const &time`	`void`	Takes a const ref to a Time object (e.g. `Seconds (10.0)`) for specifying simulation duration.
`Schedule`	`Time const &time` `EventImpl *event`	`EventId`	Schedules an event for the current context at `time`. Returns the UID of the event.

`ScheduleWithContext`	`uint32_t context`
	`const &time`
	`EventImpl *event`
		`EventId`	Schedules event for specific context.[82]

`Now`	`void`	`Time`	Returns the current simulation time.
`Cancel`	`const EventId &ev`	`void`	Removes an event from the event set prior to processing.

Table C.2: `DistributedSimulatorImpl` attributes

Name	Type	Default Value	Description
IsRealtime	BooleanValue	false	Enables synchronization of event execution with real-time.

Table C.3: Useful `MpiInterface` public methods

Method Name	Arguments	Return Type	Description
`Enable`	`int* pargc` `char*** pargv`	`void`	Calls `MPI_Init`, passing in arguments.

`SendPacket`	`Ptr<Packet> p`
	`const Time& rxTime`
	`uint32_t node`
	`uint32_t dev`
		`void`	Serializes and sends packet to remote *node* to be scheduled at time *rxTime* on net device *dev*.

`ReceiveMessages`	`void`	`void`	Receives and deserializes packets from remote nodes and schedules receive events with the appropriate context.
`TestSendComplete`	`void`	`void`	Determines if all `MPI_Request` send requests have completed and removes them from the pending `SendBuffer` data structure if so.

Table C.4: Useful `MultiThreadingHelper` public methods

Method Name	Arguments	Return Type	Description
`Enable`	`void`	`void`	Constructs a `MultiThreadedSimulatorImpl` singleton and sets it as the simulator type. Must be called before `Install`.
`Install`	`void`	`void`	Creates partitions and adds them to the core simulator partition list by calling `MultiThreadedSimulatorImpl::AddPartition`.

Table C.5: `MultiThreadedSimulatorImpl` attributes

Name	Type	Default Value	Description
ThreadsCount	UintegerValue	1	Number of parallel event execution units to run.
BarrierType	EnumValue	*BARRIER_TREESPIN*	Synchronization barriers implementation (see [83]).
ThreadDedicatedPercent	UintegerValue	0	Percentage of thread-dedicated partitions.
SharedPartitionsSets	UintegerValue	0	Number of shared sets of partitions. Must be smaller than number of threads.
IsRealtime	BooleanValue	false	Enables synchronization of event execution with real-time.

Table C.6: Useful `MultiThreadedSimulatorImpl` public methods

Method Name	Arguments	Return Type	Description
`Schedule`	`Time const &time` `EventImpl *event`	`EventId`	Schedules an event for the current context at `time`. Returns the UID of the event. If called from outside of a partition, the event is inserted into the global event set.

`ScheduleWithContext`	`uint32_t context`
	`const &time`
	`EventImpl *event`
		`EventId`	Schedules event for specific context.

Table C.7: `RealtimeSimulatorImpl` attributes

Name	Type	Default Value	Description
SynchronizationMode	EnumValue	*SYNC_BEST_EFFORT*	What to do if the simulation cannot keep up with real time.
HardLimit	`TimeValue`	`Seconds (0.1)`	Maximum acceptable real-time jitter.

C.2 LTE Model Class Descriptions

Table C.8: Spectrum model classes used by LTE model

Name	Header File	Description
SpectrumModel	spectrum-model.h	Defines a frequency band.
SpectrumValue	spectrum-value.h	Wrapper for frequency-dependent values such as PSD.
SpectrumType	spectrum-type.h	Defines type of signal (e.g. LTE) transmitted by a SpectrumPhy-derived object over a SpectrumChannel-derived channel.
SpectrumErrorModel	spectrum-error-model.h	Shannon error model as presented in [81].
SpectrumPropagationLossModel	spectrum-propagation-loss-model.h	Used for calculating RX PSD from TX and channel PSD (based on mobility model).
SpectrumChannel	spectrum-channel.h	Base class for channel model implementations.
SingleModelSpectrumChannel	single-model-spectrum-channel.h	Single spectrum (frequency band) channel realization. Encapsulates SpectrumModel, PropagationDelayModel, and SpectrumPropagation-LossModel objects.
SpectrumPhy	spectrum-phy.h	Base class for PHY layer classes encapsulated by all Spectrum-based net device classes.

Table C.9: LTE channel model classes

Name	Header File	Description
`LteSpectrumPhy`	lte-spectrum-phy.h	Implements LTE channel characteristics. Derives from SpectrumPhy. Aggregates a SpectrumChannel and MobilityModel.
`ChannelRealization`	channel-realization.h	Encapsulates JakesFadingLossModel, PathLossModel, ShadowingLossModel and PenetrationLossModel.
`JakesFadingLossModel`	jakes-fading-loss-model.h	Models propagation loss due to Jakes fast fading.
`PathLossModel`	path-loss-model.h	Models path loss based on MobilityModel of sender and receiver.
`ShadowingLossModel`	shadowing-loss-model.h	Models propagation loss due to shadowing.
`PenetrationLossModel`	penetration-loss-model.h	Models penetration loss.
`LtePropagationLossModel`	lte-propagation-loss-model.h	LTE propagation loss model. Aggregated to SingleModelSpectrumChannel. Encapsulates ChannelRealization. Calculates RX PSD.

Table C.10: LTE PHY-layer classes

Name	Header File	Description
`IdealControlMessage`	ideal-control-messages.h	Implements basic PDCCH RB allocation map and CQI feedback messages.
`LtePhy`	lte-phy.h	Base class for LteEnbPhy and LteUePhy. Aggregates two (DL and UL) LteSpectrumPhy and SingleModelSpectrumChannel objects. Has PHY-layer radio characteristics such as TX power, noise figure, EARFCN, bandwidth (in terms of RBs), cell ID.
`LteEnbPhy`	lte-enb-phy.h	eNodeB PHY layer implementation. Derives from LtePhy. Handles transmission and reception of DL and UL frames/subframes. Handles ideal PDCCH RB allocation map and CQI messages.
`LteUePhy`	lte-ue-phy.h	UE PHY layer implementation. Derives from LtePhy. Builds UL subframes as trigged by LteEnbPhy object for eNodeB. Receives ideal control messages and generates CQI feedback messages. Has a unique RNTI.
`LteUePhySapProvider`	lte-ue-phy-sap.h	PHY Service Access Point interface to MAC: PHY methods called by MAC (methods implemented by LtePhy).
`LteUePhySapUser`	lte-ue-phy-sap.h	PHY Service Access Point interface MAC layer: MAC methods called by PHY (methods implemented by LteEnbMac/LteUeMac).
`LteAmc`	lte-amc.h	Implements Adaptive Modulation and Coding scheme. Methods are invoked by the MAC scheduler for determining MCS from CQI feedback, determining TB size from MCS, and generating CQI reports.
`LtePhyTag`	lte-phy-tag.h	Packet tag for PHY PDU for conveying RNTI and LCID.

Table C.11: LTE MAC-layer classes

Name	Header File	Description
FfMacScheduler	ff-mac-scheduler.h	Abstract base class for MAC scheduler.
LteEnbMac	lte-enb-mac.h	LTE eNodeB MAC layer. Encapsulates MAC scheduler, MAC SAP, and CMAC SAP. Maintains BSRs, CQIs and other parameters used by scheduler.
LteEnbCmacSapProvider	lte-enb-mac.h	Base class for SAP offered to RRC by CMAC function.
EnbMacMemberLteEnbCmacSapProvider	lte-enb-mac.h	Derives from LteEnbCmacSapProvider. SAP used by RRC for notifying CMAC function about UE/LC information.
LteMacSapProvider	lte-enb-mac.h	Base class for SAP offered to RLC by user-plane MAC function.
EnbMacMemberLteMacSapProvider	lte-enb-mac.h	Derives from LteMacSapProvider. SAP used by RLC for sending RLC PDUs and BSRs to MAC layer.
FfMacCschedSapUser	lte-enb-mac.h	Base class for methods used by CMAC for accessing the scheduler.
EnbMacMemberFfMacCschedSapUser	lte-enb-mac.h	Derives from FfMacCschedSapUser. Used by CMAC to configure UE/LC parameters in scheduler.
EnbMacMemberLteEnbPhySapUser	lte-enb-mac.h	
FfMacCschedSapProvider	ff-mac-csched-sap.h	Abstract base class for MAC scheduler control-plane SAP offered to CMAC (based on Femto Forum MAC API). Contains basic data structures, parameters and constants as defined in Section 4.1 of [59].
FfMacSchedSapProvider	ff-mac-sched-sap.h	Abstract base class for MAC scheduler user-plane SAP offered to MAC (based on Femto Forum MAC API). Contains basic data structures, parameters and constants as defined in Section 4.2 of [59].

Table C.12: LTE higher-layer protocol classes

Name	Header File	Description
`LteRlcSm`	lte-rlc.h	RLC "satuturtion mode" implementation. See [80].
`LteRlcUm`	lte-rlc.h	RLC Unacknowledged Mode implementation. One instance per each RAB for both UE and eNodeB. Created and managed by LteUeRrc and LteEnbRrc at the UE and eNodeB, respectively.
`RlcDataHeader`	lte-rlc-header.h	Derives from Header. Provides Serialize and Deserialize methods.
`LteEnbRrc`	lte-enb-rlc.h	RRC layer at eNodeB. Manages radio bearers.
`LteUeRrc`	lte-ue-rlc.h	RRC layer at UE. Manages radio bearers.
`EpcGtpU`	epc-gtpuv1.h	GTP-U protocol. One instance for each GW and eNodeB. Encapsulates/decapsulates user data over GTP/UDP-based tunnels.
`GtpuHeader`	gtpuv1-header.h	GTP-U header. Provides Serialize and Deserialize methods.

C.3 Results from Core Simulator Implementation Performance Tests

Table C.13: Non-real time, distributed implementation - bottleneck test (all times in seconds)

(a) Total user and kernel (sys)-space times

	num proc														
	1			**2**			**4**			**8**			**16**		
num UE	real	user	sys	real	user	sys	real	user	sys	real	user	sys	real	user	sys
128	1.11	0.9	0.08	1.03	0.35	0.55	2.03	1.09	1.56	2.08	0.35	0.55	2.24	0.37	0.64
256	3.06	2.81	0.09	3.88	1.05	1.69	3.87	1.94	3.46	3.88	0.95	1.76	4.03	0.95	1.76
1024	45.95	45.54	0.19	45.14	16.99	27.97	45.83	16.73	27.88	45.55	15.92	28.37	47.98	33.47	59.82

(b) User and kernel (sys)-space times normalized by number of processes

	num proc														
	1			**2**			**4**			**8**			**16**		
num UE	real	user	sys	real	user	sys	real	user	sys	real	user	sys	real	user	sys
128	1.11	0.90	0.08	1.03	0.18	0.28	2.03	0.27	0.39	2.08	0.04	0.07	2.24	0.02	0.04
256	3.06	2.81	0.09	3.88	0.53	0.85	3.87	0.49	0.87	3.88	0.12	0.22	4.03	0.06	0.11
1024	45.95	45.54	0.19	45.14	8.50	13.99	45.83	4.18	6.97	45.55	1.99	3.55	47.98	2.09	3.74

Table C.14: Non-real time, distributed implementation - embarrassingly parallel test (all times in seconds)

(a) Total user and kernel (sys)-space times

	num proc					
	1			**2**		
num UE	real	user	sys	real	user	sys
128	0.61	0.44	0.04	1.66	0.25	0.28
256	1.41	1.23	0.05	2.46	0.61	0.72
2048	78.63	78.16	0.21	76.71	48.75	27.72

(b) User and kernel (sys)-space times normalized by number of processes

	num proc					
	1			**2**		
num UE	real	user	sys	real	user	sys
128	0.61	0.44	0.04	1.66	0.125	0.14
256	1.41	1.23	0.05	2.46	0.305	0.36
2048	78.63	78.16	0.21	76.71	24.375	13.86

Table C.15: Non-real time, multithreaded implementation - bottleneck test (all times in seconds)

(a) Total user and kernel (sys)-space times

	num threads														
	1			**2**			**4**			**8**			**16**		
num UE	real	user	sys	real	user	sys	real	user	sys	real	user	sys	real	user	sys
128	0.62	0.43	0.05	0.69	0.91	0.02	0.71	1.72	0.09	0.7	3.24	0.17	0.7	6.26	0.58
256	1.11	0.94	0.02	1.16	1.59	0.04	1.16	2.8	0.07	1.13	4.92	0.14	1.11	9.04	0.64
1024	11.86	11.68	0.02	11.73	13.74	0.08	11.64	17.26	0.1	11.16	22.62	0.2	11.86	34.14	0.8

(b) User and kernel (sys)-space times normalized by number of processess

	num threads														
	1			**2**			**4**			**8**			**16**		
num UE	real	user	sys	real	user	sys	real	user	sys	real	user	sys	real	user	sys
128	0.62	0.43	0.05	0.69	0.46	0.01	0.71	0.43	0.02	0.70	0.41	0.02	0.70	0.39	0.04
256	1.11	0.94	0.02	1.16	0.80	0.02	1.16	0.70	0.02	1.13	0.62	0.02	1.11	0.57	0.04
1024	11.86	11.68	0.02	11.73	6.87	0.04	11.64	4.32	0.03	11.16	2.83	0.03	11.86	2.13	0.05

Table C.16: Real time, distributed implementation - bottleneck test - jitter (intranode, num nodes=16)

(a) Jitter for hub node (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
1	1.47e3	7.10e3	1.10e3	3.34e2	2.69e2	5.00e3	0.00	4.59e2
2	1.46e3	5.60e3	1.10e3	2.87e2	2.88e2	1.00e3	0.00	4.53e2
8	1.66e3	8.60e3	6.00e2	3.37e2	3.26e1	6.00e3	0.00	2.29e2
16	3.70e4	9.16e6	6.00e2	4.61e5	7.02e4	9.52e6	0.00	6.46e5

(b) Jitter for spoke nodes (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
1	1.19e3	4.22e4	7.00e2	1.09e3	1.33e2	8.10e4	0.00	2.15e3
2	1.16e3	3.70e3	7.00e2	1.77e2	8.75e1	3.00e3	0.00	2.90e
8	1.21e3	3.70e3	1.20e3	1.32e2	2.78	1.00e3	0.00	5.27e1
16	3.95e4	6.50e6	1.20e3	4.86e5	7.66e4	9.78e6	0.00	7.68e5

Table C.17: Real time, distributed implementation - bottleneck test - jitter (internode, num nodes=16)

(a) Jitter for hub node (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
2	1.49e3	6.86e4	6.01e2	6.01e2	3.46e2	6.70e4	0.00	1.82e3
8	9.98e4	9.29e6	6.00e2	6.01e2	1.11e5	1.58e6	0.00	2.89e5
16	3.44e5	2.05e7	6.00e2	1.60e3	5.16e5	7.49e6	0.00	1.02e6
32	.57e5	2.88e7	6.00e2	6.00e2	8.45e5	1.28e7	0.00	1.39e6

(b) Jitter for spoke nodes (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
2	1.21e3	1.82e4	2.01e2	1.20e3	3.06e1	1.70e4	0.00	6.45e2
8	1.31e5	8.73e6	1.20e3	1.20e3	2.10e5	2.28e6	0.00	2.67e5
16	3.30e5	1.64e7	1.20e3	1.20e3	4.66e5	7.50e6	0.00	7.45e5
32	4.36e5	1.38e7	1.20e3	1.20e3	6.45e5	7.66e6	0.00	1.24e6

Table C.18: Real time, distributed implementation - bottleneck test - jitter (intranode, num nodes=32)

(a) Jitter for hub node (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
1	1.38e3	5.91e4	6.00e2	1.16e3	3.25e2	3.70e4	0.00	8.26e2
2	1.61e3	3.41e4	6.01e2	6.15e2	3.06e1	6.10e4	0.00	1.15e3
14	4.90e6	7.90e6	1.43e6	1.95e6	3.41e6	7.70e6	8.97e5	1.91e6
16	1.63e5	7.78e5	6.00e2	2.43e5	3.23e5	1.54e6	0.00	4.85e5

(b) Jitter for spoke nodes (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
1	1.20e3	1.32e4	1.20e3	2.25e2	8.68	2.10e4	0.00	3.96e2
2	3.79e3	5.42e4	2.01e2	1.05e4	5.43e3	9.70e4	0.00	2.06e4
14	2.24e6	5.80e6	1.20e3	2.75e6	3.83e5	6.11e6	1.00	5.29e5
16	1.10e5	4.49e5	1.20e3	1.89e5	2.18e5	8.94e5	0.00	3.78e5
32	2.42e7	2.51e7	2.38e7	2.05e5	1.17e7	1.28e7	1.10e7	2.80e

Table C.19: Real time, distributed implementation - bottleneck test - jitter (intranode, num nodes=128)

(a) Jitter for hub node (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
1	5.49e3	1.51e6	1.10e3	2.12e4	2.62e2	1.35e5	0.00	2.61e3
2	1.31e5	1.10e6	1.60e3	2.00e5	2.52e5	1.24e6	0.00	3.83e5
14	1.72e4	1.00e6	1.10e3	7.13e4	2.24e4	1.22e6	0.00	1.22e5
16	4.22e6	5.49e6	2.63e6	5.34e5	8.43e6	1.07e7	5.26e6	1.04e6
32	8.29e7	8.90e7	7.53e7	2.37e6	2.79e6	9.15e6	2.00e3	1.05e6

(b) Jitter for spoke nodes (over 2 trials)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)	Δmean(ns)	Δmax(ns)	Δmin(ns)	Δstd(ns)
1	3.35e3	9.07e5	7.00e2	1.34e4	4.46e2	1.36e5	0.00	3.52e3
2	4.64e5	1.76e6	2.01e2	5.82e5	9.22e5	2.36e6	0.00	7.61e5
14	1.77e4	5.69e5	7.00e2	7.61e4	3.31e4	1.13e6	0.00	1.52e5
16	2.26e6	5.46e6	1.20e3	2.28e6	4.51e6	1.03e7	1.00	4.55e6
32	4.25e7	8.77e7	3.20e3	4.21e7	1.91e6	3.80e7	0.00	2.00e6

C.4 Results from LTE Simulation Test Cases

Table C.20: LTE saturation test case: Timing jitter for DL RLC RX events at UE

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	4.02e4	1.01e5	2.60e4	1.06e4
2	7.10e4	2.68e5	3.50e4	4.62e4
4	1.49e5	1.27e6	3.00e4	1.98e5
8	1.78e5	3.40e6	4.10e4	3.36e5
16	2.63e5	2.42e6	5.50e4	3.12e5
32	2.90e8	5.85e8	1.28e5	1.69e8
64	1.19e9	2.37e9	3.81e5	6.85e8

Table C.21: LTE simple traffic generator test case (num proc=1)

(a) Timing jitter for DL RLC RX events at UE

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	6.20e4	9.20e4	5.40e4	7.96e3
2	6.07e4	1.07e5	4.60e4	1.52e4
4	6.70e4	1.33e5	4.10e4	3.04e4
8	1.02e5	2.29e5	5.10e4	5.06e4
16	2.64e5	1.52e6	5.50e4	2.60e5
32	2.53e6	7.56e6	1.00e5	1.76e6
48	2.16e7	1.44e8	1.14e5	3.62e7
64	2.25e8	5.18e8	1.36e5	1.53e8

(b) Timing jitter for DL PPP RX events for nodes 0, 1 and 2 (server, GW, eNodeB)

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	3.06e4	4.26e4	2.28e4	4.96e3
2	4.95e4	7.22e4	3.18e4	1.20e4
4	7.03e4	1.15e5	3.58e4	2.26e4
8	1.50e5	3.60e5	5.68e4	6.42e4
16	5.08e5	2.09e6	8.88e4	2.50e5
32	1.34e6	1.04e7	1.75e5	1.92e6
48	3.17e7	1.45e8	2.78e5	4.92e7
64	3.21e8	6.39e8	4.26e5	1.94e8

Table C.22: LTE dist. network test case (intranode, num proc=2)

(a) Timing jitter for DL RLC RX events at UE

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	6.05e4	9.20e4	5.30e4	8.80e3
2	5.18e4	1.00e5	3.80e4	1.40e4
4	6.49e4	1.24e5	4.00e4	2.82e4
8	9.29e4	2.15e5	4.50e4	4.94e4
16	2.26e5	1.07e6	5.70e4	1.97e5
32	2.41e6	7.35e6	1.05e5	1.66e6

(b) Timing jitter for DL PPP RX events at nodes 0 and 1 (server and GW)

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	3.32e4	1.03e5	1.25e3	4.51e4
2	3.57e3	1.81e4	1.25e3	4.56e3
4	1.32e4	5.53e4	8.17e2	1.65e4
8	2.20e4	1.08e5	1.25e3	2.90e4
16	3.85e5	1.87e6	2.48e2	2.69e5
32	1.06e6	7.35e6	2.48e2	1.60e6

(c) Timing jitter for DL PPP RX events at nodes 2 (eNodeB)

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	3.37e4	6.38e4	2.88e4	7.57e3
2	4.82e4	7.18e4	3.68e4	8.83e3
4	7.53e4	1.09e5	5.08e4	1.23e4
8	1.27e5	2.01e5	7.88e4	2.22e4
16	4.97e5	6.97e5	1.20e5	1.40e5
32	5.65e5	8.21e5	2.30e5	1.08e5

Table C.23: LTE dist. network test case (internode, num proc=2)

(a) Timing jitter for DL RLC RX events at UE

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	6.17e4	1.15e5	5.50e4	1.28e4
2	5.53e4	1.17e5	4.10e4	3.10e4
8	1.02e5	2.44e5	4.60e4	5.51e4
16	3.19e5	4.79e6	5.70e4	5.13e5
32	2.42e6	7.43e6	9.80e4	1.68e6

(b) Timing jitter for DL PPP RX events at node 2 (eNodeB)

num UE	mean(ns)	max(ns)	min(ns)	std(ns)
1	42717	3.58e4	3.58e4	1.02e4
2	59335	4.08e4	4.08e4	1.32e4
4	75720	5.58e4	5.58e4	1.24e4
8	128640	7.98e4	7.98e4	2.34e4
16	486390	1.19e5	1.19e5	1.45e5
32	572220	2.39e5	2.39e5	1.10e5

Table C.24: LTE real traffic test case - jitter

(a) Timing jitter for DL RLC RX events at UE (node 3)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)
1	5.68e4	1.41e6	3.20e4	4.37e4
2	5.7134e4	1.46e6	3.10e4	6.39e4

(b) Timing jitter for DL PPP RX events (nodes 0,1,2)

num proc	mean(ns)	max(ns)	min(ns)	std(ns)
1	1.29e4	2.25e6	2.82e2	6.05e4
2	1.21e4	2.27e6	6.00	8.53e4

Table C.25: LTE real traffic test case - HTTP transfer rate

num proc	mean(kbps)	max(kbps)	min(kbps)
1	1992	2296	1248
2	1784	2232	1256