Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Distributed Application Testing Framework

Ankana Bhattacharjee

August 19, 2022

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MSc (Computer Engineering)

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

A large-scale distributed system needs to be tested thoroughly before production deployment. It is crucial to inject several kinds of faults to assess application behaviour, fault tolerance and scalability. This is often done with the help of various network simulators like NS3, Omnet++, Mininet and Opnet. Generally this involves a considerable amount of coding within the network simulator code in order to facilitate the desired test scenarios, involving its own learning curve that requires substantial time and effort. This in turn affects and prolongs the overall life-cycle of the software development and deployment. It is this process that we seek to simplify with this project. In this dissertation, we present a proof of concept in which a developer/tester can easily test the behaviour any large-scale application, independent of language and architecture with the help of a graphical user interface.

Apart from the network simulator, a major component of the approach is to use an application containerization framework like Linux containers (LXC) or Docker for a dynamic and configurable deployment of any application in the test environment. One of the most important uses of containerization frameworks to to maintain a clear separation between the network simulator and the test application code bases. This involves facilitating containers with work in conjunction with the network simulator.

For the proof of concept, our algorithm demonstrates a few specific functionalities that we think are critical in the process of testing any networked application namely: topology generation, link up/down, adding/removing network nodes during application run-time. We illustrate the effectiveness of the system by testing how dynamically we can create nodes, run applications on them, how easily we can inject faults and test the behaviour of the application being simulated.

Acknowledgements

I am deeply grateful to my supervisor, Dr. Vinny Cahill for his impeccable guidance and encouragement throughout my endeavour in this thesis. His invaluable teachings and advice have been an irreplaceable source of learning and growth throughout this dissertation.

I would like to express gratitude to my dear father, who has always encouraged me to pursue my career and this Master's curriculum, for being so supportive and understanding.

Rajesh Gangam, for his indomitable knowledge and insight in network protocols, inter-process communication, OS internals and everything else in between. It would not have been possible to complete this work, or the curriculum for that matter, without his intuition and guidance.

Chiara Paletta for being such a dear friend and confidant throughout my time at Trinity College.

Lastly, a sincere and heart-felt word of thanks to my lovely cats Gingi, Houdini and Mucci, for illuminating my days and especially my darkest hours with their much esteemed cat-quirk.

Contents

List of Figures

1 Introduction

In this work we will consider the challenges involved in testing network behaviours in large-scale distributed applications running across several hosts and varying computing environments. Testing network behaviours in application over several scenarios can potentially involve a large number of software and hardware components. While software components are developed in code, often to test them a large number of hardware components need to be involved to test scaling abilities and latency of the application. In most practical cases this can be cost heavy. To tackle this issue, network simulators are used. Network simulators are capable of simulating large-scale networks in a single host. They are capable of generating and orchestrating various topologies like star, mesh, grid etc and introduce various faults like packet loss, packet congestion etc. It is also possible to control specific interfaces of the network by explicitly bringing them up or down.

Although network simulators largely reduce the time and cost needed to test large-scale applications as compared to real networks, they involve a considerable learning curve of the simulator framework itself, the language it is coded in, the coding standards involved in writing code compatible with the framework and so on. Often this leads to the developer spending a substantial amount of time and effort just to learn the workings and concepts of the simulator framework itself,while getting familiar with the the specific coding standards of the simulator framework in question. Most importantly, the network simulator code needs to be modified each time the application to be tested is changed or modified. This can involve a lot of time and effort which in turn prolongs the overall software development life cycle of the application. Therefore the motivation of this dissertation is to develop a system that simplifies this process by abstracting away the need to learn the workings of the network simulator framework.

1.1 Motivation

Due to the complexities involved in application testing described above, the challenge is to simplify network simulation on large-scale applications involving multiple nodes, as much as possible. This can be done by developing more configurable ways of using the network

simulation framework. The idea is to make all application testing parameters dynamic and configurable. Some of the important parameters include the number of nodes to generate, the applications that each node has to run and the topology of the network. Ideally one should be able to pass all parameters including the speed of the data transfer in the network.

An important aspect of achieving this is to separate the network simulator code and the application code. The network simulator framework and the application to be tested must be completely independent of each other. This can be realized with the help of containerization frameworks. Another desired functionality is to have the ability to control the network simulator with the help of the graphical user interface where the user can input network parameters and call the required framework functions via the GUI itself.

Thus we have developed the new system keeping the above requirements in consideration. The system is evaluated on the following aspects:

- Ease of deployment of applications on the test framework
- Ability to dynamically add/remove nodes
- Ability to dynamically change the applications to be run on the simulator without any code changes to the system, especially the simulator framework itself
- The flexibility with which new network topologies can be generated

1.2 Thesis

From the background so far, we can define our thesis as follows:

Given any distributed application built on any technology stack, is it feasible to build a system based on a currently available network simulator that can make the application testing as easy and configurable as possible. If so, how configurable and dynamic can the system be made and to what extent can the application be simulated with the help of the framework.

Therefore this thesis aims to investigate a way in which large-scale distributed applications can be tested dynamically. Due to the sheer number of parameters that a network can have e.g. topologies, number of nodes, interface types, protocols, channels and data rate among others, for the practical purpose of building a proof of concept, it is necessary to to limit our research to a subset of these parameters. Therefore for this thesis, we impose the following limitations to our system:

- Subnet of 25, which imples that maximum number of nodes can be 255
- The topologies we are testing: bus and star

- We test only with IPv4 addresses that are assigned automatically and not provided by the user
- Channel speed is not configurable

We test the following functionalities:

- Topology generation with a variable number of nodes
- Dynamically create nodes and assign applications to run on them
- Feasibility of adding/removing an interface while the simulator is running
- Feasibility and behaviour of a network when bringing a node up/down

1.3 Contributions of this work

As described in the Chapter 2, we have identified a few key limitations of using network simulators that considerably slow down the process of large-scale application testing:

- A substantial amount of work involved in simulating applications that involve understanding the complexities of the network simulator framework itself and tweaking the framework code to generate test scenarios
- Tweaking the framework itself requires a working knowledge of the coding language in which the framework is built in
- Modifying the network simulator code is very specific to the application being tested. The code needs to be changed for every application which in turn negatively impacts the scalability of application simulation

This thesis contributes towards mitigating the above mentioned limitations by building a framework that allows a user to tweak some of the important parameters of a given network dynamically through a GUI interface.

1.4 Overview

Chapter 2 comprises an overview of what network simulators are, how they can be used to simulate applications. Specifically we discuss a few prominent network simulators in wide use, the features they offer as well as few of their limitations where applicable to our problem statement. We then identify the network simulator we will use and the driving factors behind our decision. Chapter 3 takes a dive into the basic concepts that make up the network simulator framework we have opted for. Chapter 4 discusses the need for containerization of applications, how our network simulator can be made to work with

containers and finally the factors and considerations that drive our choice of the specific container framework. Chapter 5 discusses the overall architecture of the system built, some important aspects of the communication mechanism between the back-end and the GUI interface and key design decisions behind the architecture. Chapter 6 contains the results of experiments run, a high level evaluation of the performance of the system and how it measures up with respect to the aims of the thesis. Chapter 7 discusses the conclusions of the thesis and possibilities of further work.

2 State Of The Art

Given the objectives of this thesis,it is imperative to understand what network simulators are available and what they offer. Although programmers have the choice of working with a wide variety of simulators, each with their own features and capabilities, it is out of the scope of this work to analyse each and every one of them. We will take a look at few of the most widely used network simulators available to us and discuss the features they offer and how they perform. The simulators described here vary widely in terms of capabilities, in that some simulators like NS3, NS2 and Omnett++ are more suitable for industrial use and some are more suitable for academic and research purposes like OPNET. But we still talk about the latter because it is relatively easier to work with and it provides a comprehensive GUI for simulating and debugging networks, which is of high interest for the objectives of this thesis.

Although it would be ideal to compare all considered network simulators on a fixed set of parameters that cater to our requirements, it is not feasible because all available network simulator frameworks have been built targeting widely varying requirements. Hence as we go through them, we mention some of their specific positives and negatives in relation to our thesis requirement.

2.1 Optimized Network Engineering Tools (OPNET)

OPNET is a network simulator built keeping low latency and high scalability in mind. It can simulate all components of the network: routers, servers, switches and protocols. A major advantage of the OPNET framework is its modular structure that allows for code reuse. OPNET network model is a hierarchy of sub-networks. There are three different hierarchies available.(1)

- Node
- Network

- Process

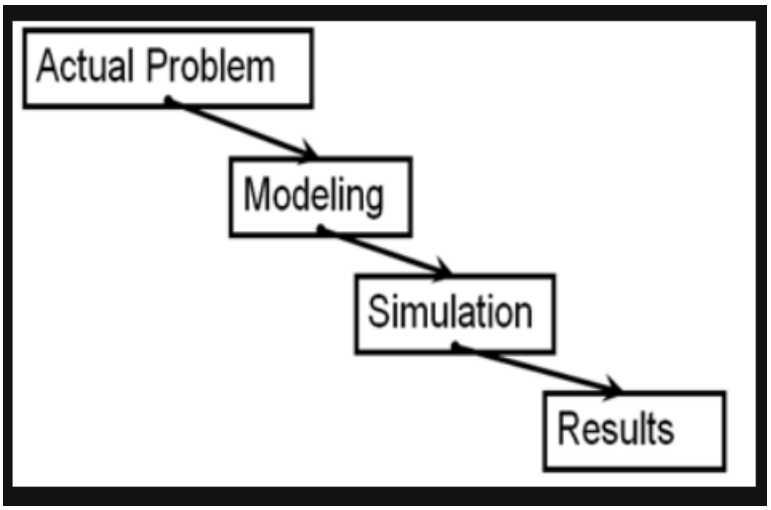

Figure 2.1: Architecture of OPNET Network Simulation (1)

2.1.1 Main OPNET Features

The following features of OPNET are useful for building a testing framework.

- Source code provides lots of components with its library
- Fast and discrete event simulation engine
- Object Oriented Modelling
- Graphical user interface supports 32-bit and 64-bit
- 32-bit and 64-bit parallel simulation kernel
- GUI-based debugging
- High performance framework capable of delivering testing scenarios for industrial scale applications (7)

2.1.2 OPNET Limitations

We found the following limitations that could hinder the use of OPNET as a network simulator framework of choice

- Although it has a comprehensive GUI, it still requires a substantial amount of coding as demonstrated in (8)
- Lacks container support that makes the overall process of orchestrating and managing application testing less configurable

- OPNET is an expensive software that could prevent many developers from testing their applications using the framework.

2.2 Objective Modular Network Testbed in C++ (OMNet++)

Andras Varga gives us an introduction of OMNet++ in (2) as a discrete event simulation framework written in C++. It was developed mainly for modelling distributed systems in computer networks. The motivation for developing OMNet++ was to try and fill the gap between research oriented complicated software like Network Simulator (NS) and expensive commercial software like OPNET (2).

2.2.1 OMNet++ Design

The development of this software was done keeping distributed applications and network scaling in mind. This influenced the following design requirements of the framework:

- Simulations are hierarchical and made up of simple reusable modules to support large-scale simulation.
- To cut down on debugging time, the simulation software should emphasize the need of simple traceability and debuggability of simulation models.
- Modularity, adaptability, and the ability to incorporate various simulation models and scenarios. This was realised with object-oriented programming concepts.

Figure 2.2 is the model structure in OMNet++ (2). The figure demonstrates the modular structure following by the framework, emphasizing on code reusability, robustness and maintainability that is imperative to the system we are trying to build.

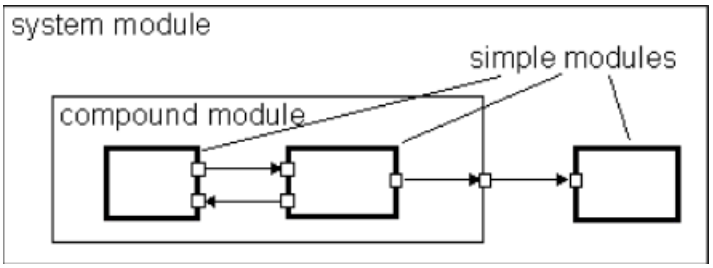

Figure 2.2: Model Structure of OMNet (2)

OMNet++ has defined its own language to describe the desired topology of the simulated network, called the NED. The components of NED consist of simple module declarations, compound module definitions and network definitions (9). For better readability and

management, it is possible to partition large NED files into many smaller ones using file inclusion.

2.2.2 Main Omnet++ Features

The following features of Omnet++ are useful for building a testing framework:

- OMNet++ provides its own GUI tool called Tkenv. It reflects state changes of the nodes in the display and animates the flow of messages in the network.
- With OMNet++, network topologies can be modified on the fly, while the simulator is running.
- OMNet++ has the capability of working with Docker containers (10).
- OMNet++ provides a message passing interface (MPI) that is used to facilitate network simulation on applications running across multiple processors (11). This feature is especially attractive to the aim of this thesis because large-scale applications spanning a very large number of nodes can be supported more easily when multiple processors are involved.
- Network topologies and parameters can be modified while the simulator is running

2.2.3 OMNet++ Limitations

We found the following limitations with OMNet++:

- Requires the use of NED programming language, involving a substantial learning curve.
- Can work with Docker only and not Linux containers, which may pose a security concern.

2.3 Network Simulator 3 (NS3)

In 2008 Henderson et al. introduced the Network simulator-3 (NS3) (12). It has been developed on top of the Network Simulator-2 (NS2) (3). Using NS2, it is possible to simulate both wired and wireless network functions and protocols (such as routing algorithms, TCP, and UDP). NS2 gives users a mechanism to define these network protocols and simulate the related behaviour. NS2 was the first modular discrete event simulator, a model that has inspired the development of the simulators discussed in the preceding sections of this text.

2.3.1 NS3 Design

The basic architecture of NS2 is demonstrated in figure 2.3 which again emphasizes the modular structure of NS3, similar to OMNet++.

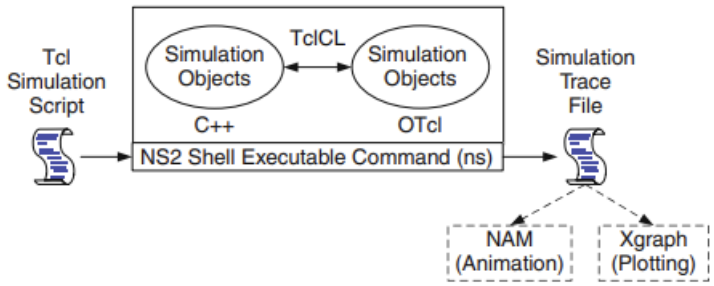

Figure 2.3: Basic architecture of NS2 (3)

Like OMNet++, NS3 has its own object-oriented language called Object-oriented Tool Command Language(OTcl) to simulate network behaviours. OTcl needs to be used in conjunction with C++ for network simulations. NS3 has adopted several concepts from various open source simulators like yans and Gnets (12).

2.3.2 Main NS3 Features

- Like OMNet++, NS3 supports message passing interface to allow network simulation across multiple processors
- Has the ability to work with both Linux containers and Docker, thus providing more room for choice
- Netanim: Netanim is an offline animator that animates the network using trace files generated from the simulation. The simulation is now animated using an XML trace file that was gathered throughout the simulation (13).
- ns3-lxc: This project automates the integration of NS3 with Linux containers. Topologies and functionalities are specified using YAML files. It does however require a learning curve in that the YAML files need to be written with high accuracy to not lead to simulation errors and failures (14).

2.3.3 NS3 Limitations

- Unlike OMNet++, it is not possible to modify network parameters in NS3 while the simulation is running. This is a major drawback of the the framework as it potentially hinders the user from testing a wide range of functionalities like adding/removing hosts and merging networks.

2.4 Performance comparison between OMNet++ and NS3

To fit the requirement of this thesis, it was important for us to choose a simulator that can

- Work and synchronise across multiple processors
- Integrate with application containers
- Scale sufficiently to simulate distributed application comprising of many nodes

To that effect, we have found two network simulators that meet all the above requirements: NS3 and Omnet++. Wehrle et al. in (4) gives an overview of the performance of three widely used simulators which is summarized here.

In the survey all simulations were carried out on a mesh network in the same computational environment. The simulation run time shows NS3 to be faster than OMNet++. Figure 2.4 demonstrates the simulation run-time vs drop probability of the simulators.

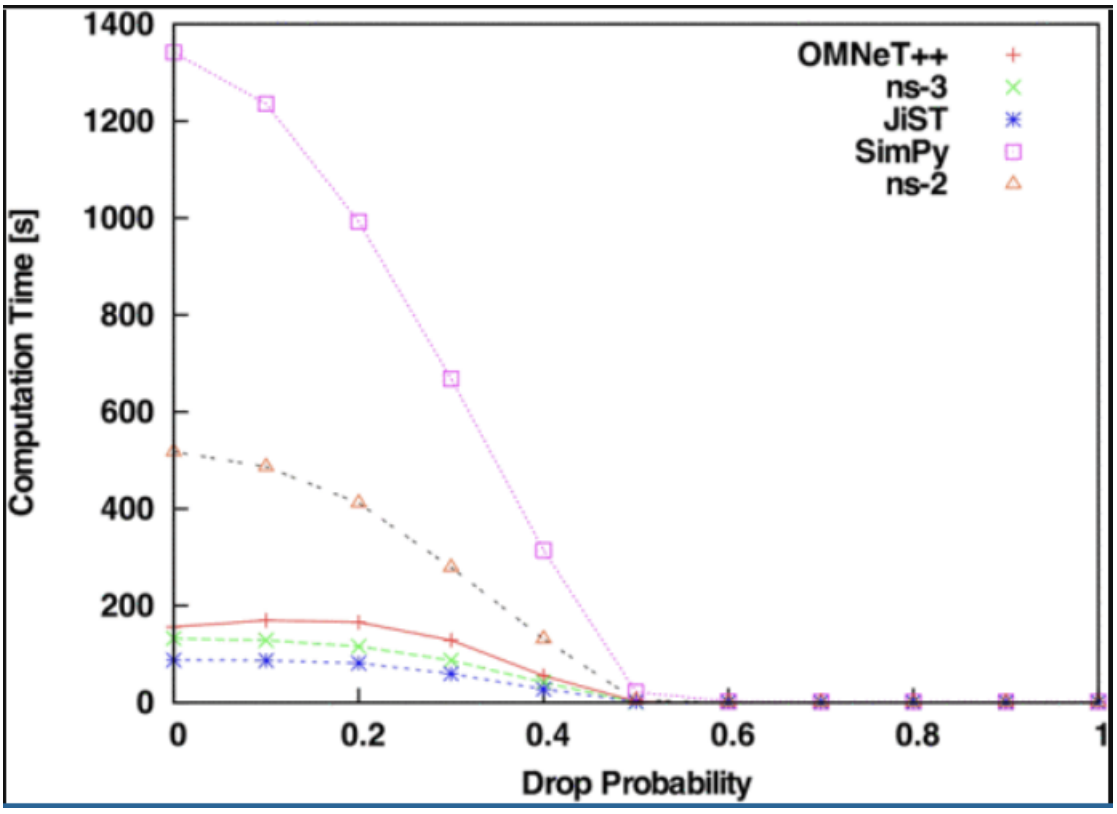

Figure 2.4: Simulation Runtime vs Drop Probability (4)

NS3 also has the lowest memory usage with increasing network size as shown in 2.5.

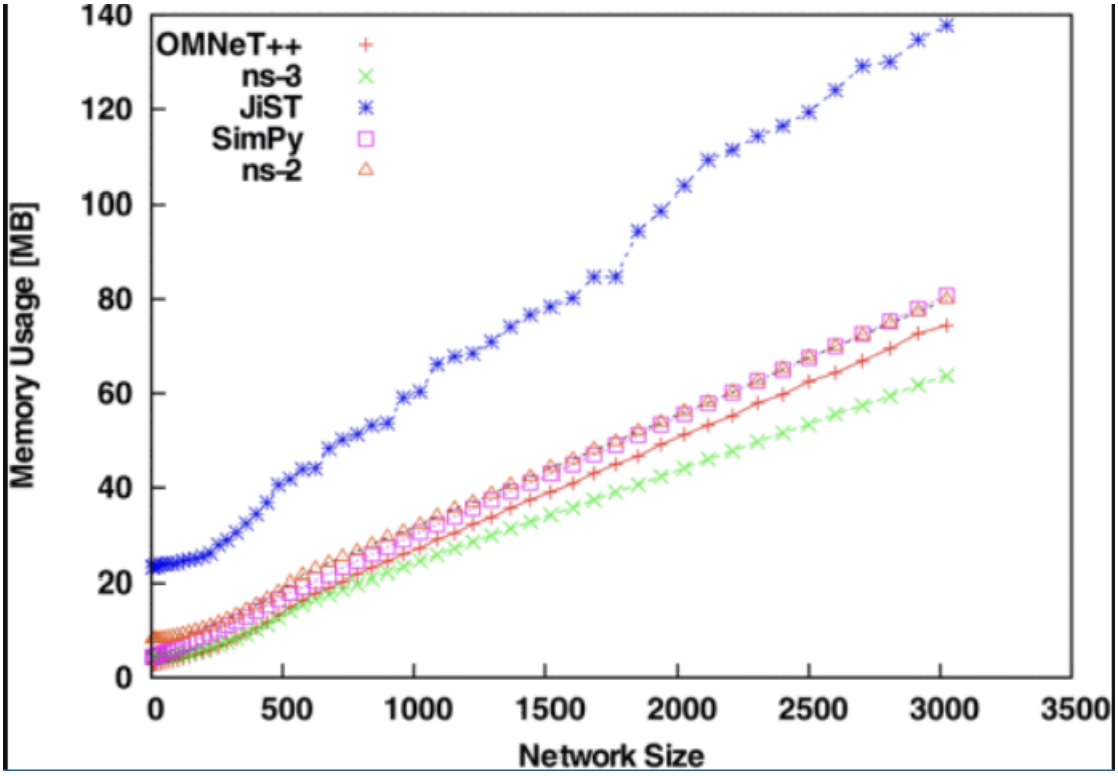

Figure 2.5: Memory usage vs Network size (4)

2.5 Conclusion

We see in this chapter that network simulators have a wide range of capabilities. Each simulator framework has been developed keeping a specific set of requirements in mind, and there really is no one-size-fits-all network simulator. However for this work, we have two network simulator frameworks that come very close to fulfilling all of the requirements: NS3 and OMNet++. We have picked NS3 over Omnet++ for the following reasons:

- Flexibility to work with both Linux container and Docker, potentially ensuring higher security
- Better performance as compared to OMNet++

The drawback of using NS3 is its inability to take simulation parameters while the simulation is running. However, with proper handling of code, this can be bypassed at the network level.This is possible because we use containers to house the applications being tested. So the connection to the containers can be handled during run-time at the network level, which in turn can simulate topology changes. But the containers will consume memory and resources when their network interfaces are switched off, which can potentially lead to a lot of redundancy depending on the application in test.

3 Network Simulator 3 (NS3)

The aim of this chapter look at some of the security considerations of NS3. It is also to introduce the concepts that make up the building blocks of NS3 as we will be using NS3 for our application simulations. It has a built-in capability to integrate with containerized applications and like Omnet++ it also has a message passing interface that allows network simulations across multiple processes thus allowing simulations of applications running across multiple hosts. It can be integrated with Linux containers (LXC) as well as Docker. Also, it is faster in comparison to OMNet++ and will be more suitable for simulating large-scale applications.

3.1 Security Considerations for NS3

NS3 has not reported any security issues at present. However NS3 being an open-source software, it is unlikely to be devoid of any security issues. NS3's attack surface is dependent largely on vulnerabilities of the open-source libraries used in its code. Being an open-source software, the code can be considered dynamic in nature and likely to have vulnerabilities waiting to be exploited. Since the software focuses mainly on simulating network errors in various network protocols, it does not deal with any application logic that is being tested on it. Therefore, it does not anonymize the application data by default, which may cause sensitive information like user credentials etc. to be leaked. Thus, it is critical to examine the applications being used on NS3 and provide data anonymization by the application developer themselves before running network simulations. One method to reduce security breaches is to implement capability-based requests that conceal the source of every request (15).

3.2 NS3 Concepts

In the official website of NS3 (16) we can get a good understanding of the building blocks and concepts of NS3. NS3 is a C++ framework and all major components, topologies and functionalities are composed of C++ classes. NS3 has the following components

3.2.1 Node

In NS3, the simulation of the most basic computing device is called the node. The class Node in C++ serves as a representation of this idea. The Node class offers ways to control how computing devices are represented in simulations.

3.2.2 Application

The Application class in NS3 is the fundamental abstraction for a user program that creates some function to be simulated. It offers techniques for handling user level applications that are tested. New applications are created by inheriting this parent class. This is the class that handles and contains the application being tested. So we will not be using this as the applications will be running in containers.

3.2.3 Channel

Channel refers to the fundamental communication sub-network abstraction, which is represented by the class Channel in C++. For managing communication sub-network objects and tying nodes to them, the Channel class offers methods. For this work we will use the CsmaChannel to demonstrate building network topologies.

3.2.4 Net Device

In NS3, the simulated hardware and software driver are both covered by the net device abstraction. In order for a Node to be able to connect with other Nodes in the simulation via Channels, a net device must be installed in the Node. A Node may be connected to more than one Channel via different NetDevices, just like in a physical computer.

3.2.5 Topology Helpers

In NS3, topology helpers are used to connect NetDevices to Nodes, NetDevices to Channels, assigning IP addresses. As we will see later in the implementation, NS3 has several topology helpers to create specific topologies like bus, star and point-to-point. Making topologies involves arranging nodes in a specific way and assigning suitable IP address to nodes in multiple subnets. Topology helper classes in NS3 make managing nodes easier and our framework will call the helper classes at the back-end.

3.3 Network Simulator Framework and the Application Code: the need for application containers

Network simulators in general have a very large code base that inevitably arises from the need to be able to generate a plethora of network topologies and scenarios. Oftentimes the code base needs to be modified to simulate specific network faults on a given application. It is rather easy to incorporate application code inextricably into the network simulator code. This can make all the code changes extremely specific to the application in questions and cannot be reused for other applications. It is also possible to exploit the application code on the basis of security vulnerabilities. Therefore to make the system as configurable and reusable as possible, we need a clear separation between the network simulator framework and the application code. One of the most reliable ways to achieve this is by deploying applications in containers. The containerization of application goes a long way in achieving the aim of this thesis in the following ways:

- It ensures the independence of the network simulator code
- It guarantees that the different components of the simulated network are isolated from each other, minimising ways to exploit security vulnerabilities of applications

4 Containerization of Applications

Containerization of applications means isolating applications to run in their own environment. In a Linux environment, this is done by utilising two main functionalities: control groups (cgroups) (17) and namespaces (18).

Control groups are used to limit and measure the total resources available to a process or a set of processes: CPU, memory, network I/O, file system. Namespaces are used to restrict the visibility of that process or group of processes to the rest of the system. A cgroup is like a process in certain ways:

- It is hierarchical
- It inherits certain attributes from its parent process

The primary distinction is the possibility of many hierarchies of cgroups existing concurrently on a system. The cgroup model is one or more distinct, disconnected trees of tasks if we consider the Linux process model as a single tree of processes (17).

A namespace encapsulates a system resource in a way that enables the processes in the namespace to have their own instance of the system resource. Any modifications to the system resource are visible to all processes present in the namespace. However, they are invisible to other processes (18).

One of the most prominent benefits of containerization is more efficient and clean use of system resources. Incompatibility issues that frequently arise while utilizing virtualization are dealt with when employing containerization. The development and test environments, the test and production environments, as well as the local physical environment and the virtual cloud environment, might not be compatible with one another. As a result, containerization makes it simpler for both operators and application developers to deliver apps to the production environment.

4.1 Containerization Frameworks

Similar to virtual machines, containers offer a means of separating distinct applications from one another and offer a virtual environment in which to operate them.

For this thesis we considered working with Linux or Docker containers in conjunction with NS3. Both Linux and Docker containers use cgroups and namespaces but with some differences. Docker provides application level isolation whereas LXC provides OS level isolation. The following diagram shows an overview of the level of isolation provided by Docker and LXC.

The primary distinguishing feature of Docker is that, unlike LXC technology, it eliminates the conventional virtualization and isolation of the full operating system in favour of concentrating more on the applications and services and their separation (5). Even while Docker was first built on top of LXC, it has since evolved into its own environment. In contrast to LXC, which only supports one guest operating system (Host OS in the left part of the figure above) on which all applications run packaged in containers, each with its own isolated environment, Docker offers an environment consisting of only one guest operating system. The Docker image is used to specify and create this environment.

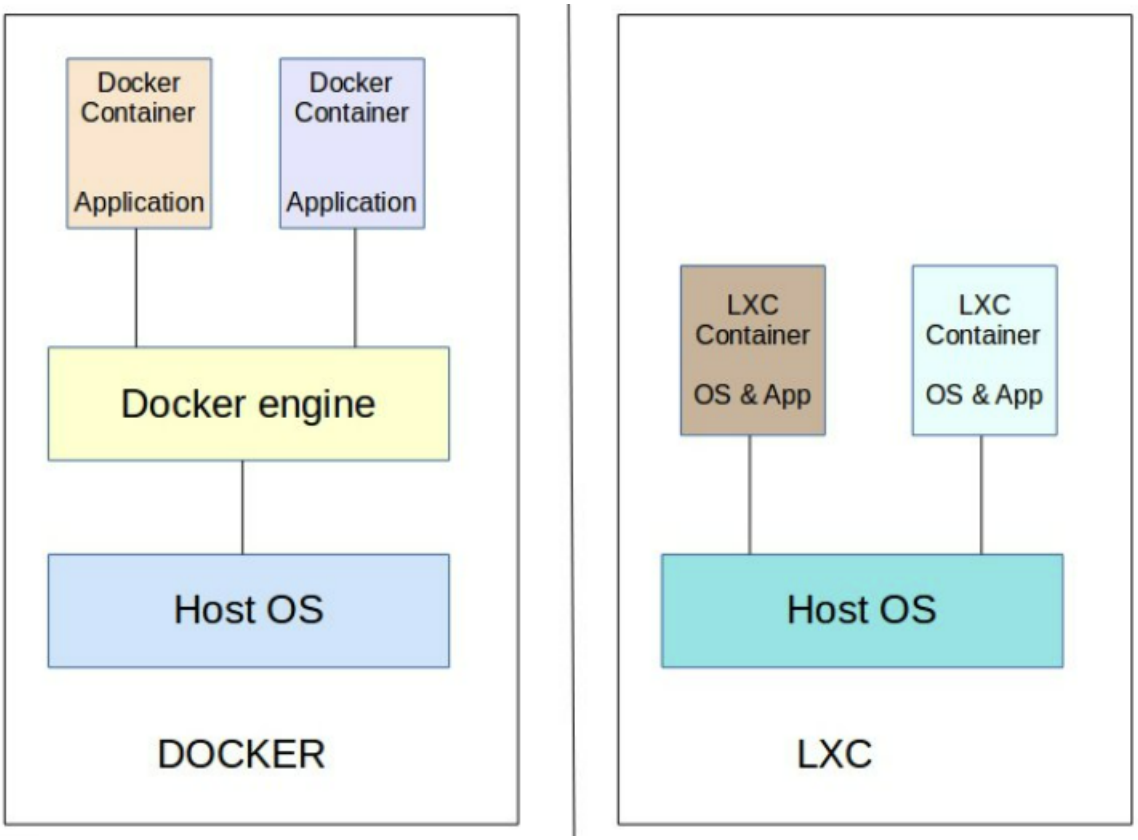

Figure 4.1: Difference between LXC and Docker (5)

4.2 Container Networking

Container networking involves connecting physical and virtual network interfaces.

We consider the virtual ethernet (veth) kernel module to set up a connection between the containers and the network simulator. The veth kernel mode creates a pair of virtual networking devices connected to each other. One of the ends is then placed in the container namespace. The veth pipe is also connected to a bridge in the default networking namespace. The following figure demonstrates how veth pairs are connected between the default networking namespace and the container namespace.

To make it simple to connect a namespace to a bridge in the default networking namespace, veth pipes are frequently used in conjunction with Linux bridges.

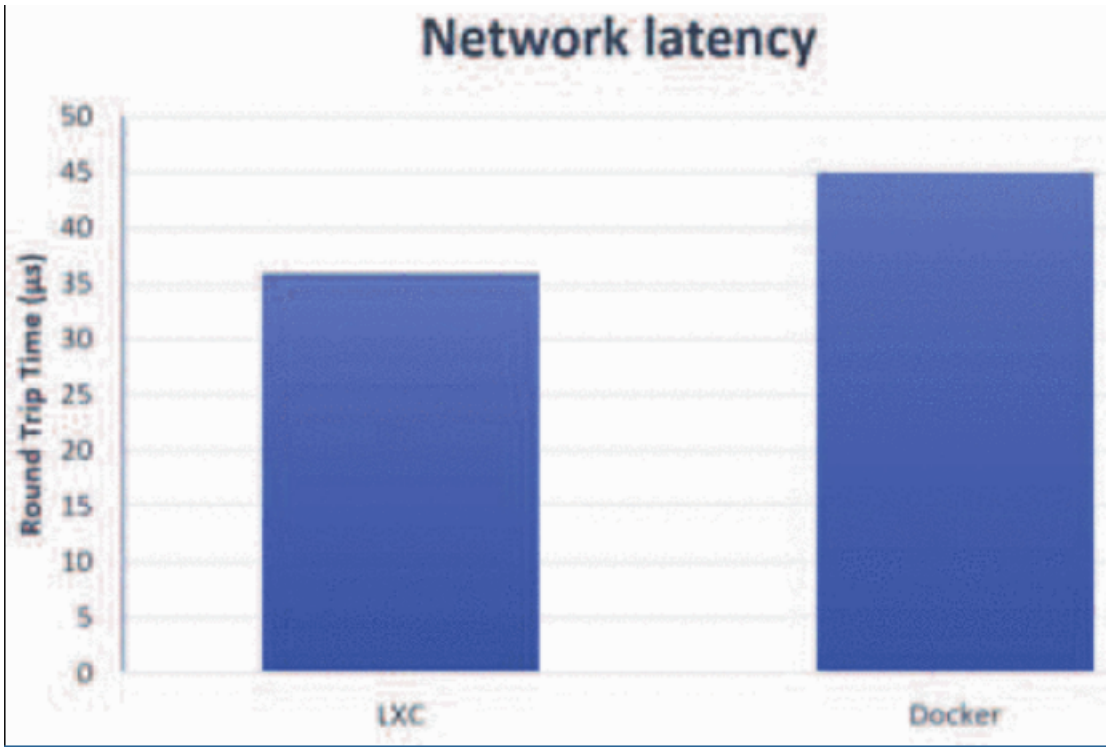

Figure 4.2: Network Latency Comparison (5)

We can claim that Linux containers and Docker are equivalent from a networking perspective (5). For both LXC and Docker

- One can assign an IP address to a container
- Bridge the container to the host system

Figure 4.2 from (5) shows that in the experiment, Docker had a Round Trip Time (RTT) latency of 45 micro seconds, compared to 36 micro seconds for LXC. In other words, Docker has a 1.25 times higher latency than LXC.

4.2.1 Container Networking in LXC with NS3

NS3 can be coupled with Linux containers using veth interfaces. The official page of NS3 demonstrates the steps involved in detail (6). Linux containers are built upon chroot jails. The jail mechanism is a FreeBSD implementation of OS-level virtualization that enables system administrators to divide a computer system into a number of separate mini systems called that share the same kernel (19).

Figure 4.3 illustrates how NS3 works with Linux Containers using veth pairs (6). In the

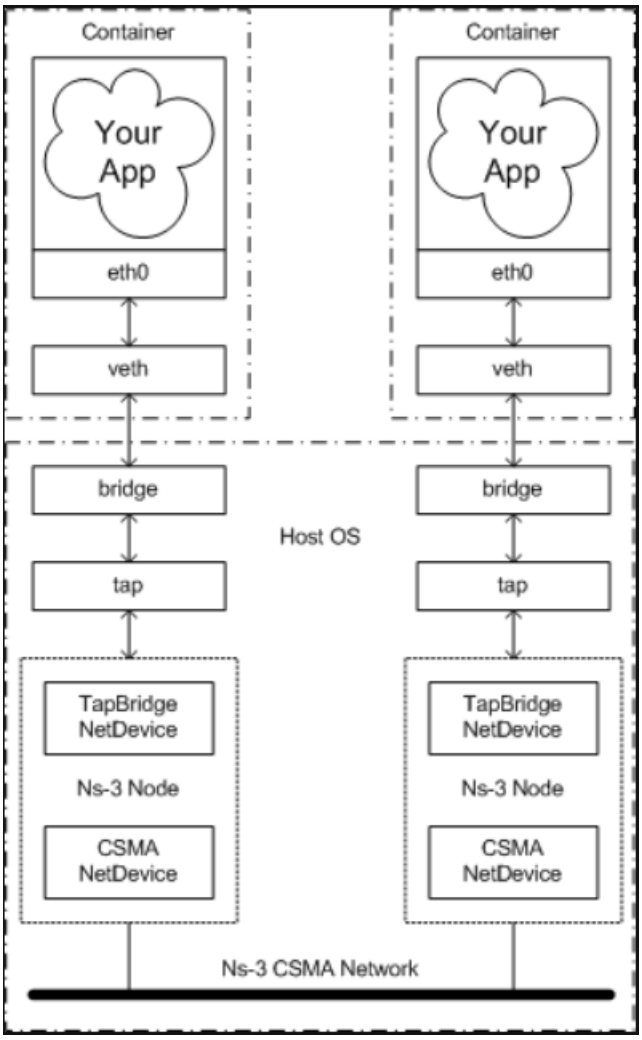

Figure 4.3: NS3 with Linux Containers (6)

figure, we can see that the Linux containers are connected to the system (NS3 running on the host) via veth pairs. The veth pairs are connected to a tap device that is created prior to creating the Linux containers. The NS3 framework must with objects of the Node class explained in section 3.2 to simulate nodes to the network. In this scenario, the Linux containers are the nodes of the network. Therefore, they need to be "installed" into the nodes created by NS3. Thus, NS3 creates stand-in nodes as demonstrated in the lower dotted-dashed box. Each node is connected to a tap device, each tap device is connected to

the Linux container which forms the actual node of the network.

The associated CSMA net device receives packets that enter through the network tap, and the network tap receives packets that enter through the CSMA net device.
Consequently, the container (and its application) will believe they are connected to the NS3 CSMA network (6).

4.2.2 Container Networking in Docker with NS3

As explained in the previous section, Docker with NS3 works pretty much the same way with the LXC containers replaced by Docker containers. Because Docker provides application level isolation as opposed to OS level isolation of LXC, namespaces in Docker behave a little differently from namespaces in LXC. A comprehensive set of steps required to set up NS3 with Docker are provided in (20).

4.3 Security Considerations with Docker

Unlike Linux containers, most operations in Docker require it to have root privileges to execute. This led us to investigate into the security concerns that may arise from the use of Docker.

4.3.1 Docker Daemon Attack Surface

It is possible to running Docker without root permissions, but not in the case of carrying out network operations. Since this project is all about networks and simulations, it must run on root. This considerably magnifies the attack surface of the Docker daemon. Therefore, only trusted applications must be run on Docker containers. But in this scenario, the term "trusted application" is very broad and rather vague. It is only too easy to exploit a piece of code to misbehave in various ways. Running a docker container in root mode, the application is at a point where it talks to the host kernel and has access many kernel subsystems that are not namespaced, like SELinux, Cgroups, files under /sys etc (21). This can make a user privilege escalation easy to orchestrate. A privileged process running inside a container is the same as a privileged process running outside a container.

Pietro et al. in (22) explain following security challenges associated with Docker usage that generally arise from poor configuration:

- privilege escalation (23)
- containers can be mounted with sensitive host directories

4.4 Conclusion

In this chapter we have seen how containers work and the differences between Docker and Linux Containers. We looked at how NS3 can be used in conjunction with Docker and LXC. Finally, we briefly talked about performance differences between Docker and LXC and the security considerations concerning the former.

The performance difference between Docker and LXC and the security concerns regarding the former has led to an important design decision in the architecture of the system we have built. We concluded that Linux containers would be a better choice as a container framework because of its general robustness in terms of latency and security. Although Docker offers higher configurability of containers and general ease of use, we have favored Linux containers over Docker because the user ideally should not think about making Docker files for every application that they want to run on a large number of nodes.

5 System Architecture

So far we have talked about the features offered by a few popular network simulators and how applications can be containerized using Linux containers and Docker. In the previous chapter, we discussed the driving factors for our decision to use Linux containers over Docker. We also decided upon the network simulator we would use for our system.

In this chapter we will discuss the overall high level architecture of the system and some factors that influenced the way we facilitate communication between the front and back ends.

5.1 The NS3 Control Interface

Let's once again take a look at the previous reference on how NS3 communicates with Linux containers (6). Figure 4.3 demonstrates how containers communicate with other containers via veth pairs connected to NS3. Our NS3 control interface will involve a method to call NS3 simulation parameters via a GUI interface. Therefore, the figure 4.3 can be modified to Figure 5.1.

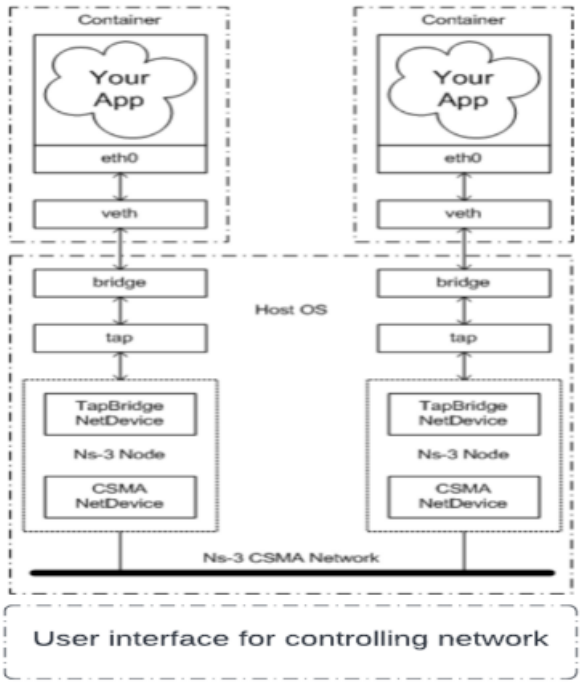

Figure 5.1: Control interface to NS3

The input parameters of NS3 can be passed as command line arguments. The user should be able to execute the following operations via the control interface:

- Create and destroy containers
- Run application inside containers
- Add/remove interfaces/nodes to the running simulation
- Create basic network topologies

5.1.1 Graphical User Interface (GUI)

For the GUI, we have used the Java Swing framework (24) to build a user interface. Every simulation starts with specifying the number of nodes to create. The first GUI prompt is to let the user specify the number of nodes. That determines how many containers the back-end should create and the IP addresses are assigned in sequence. The current implementation does not allow the user to choose the IP address.

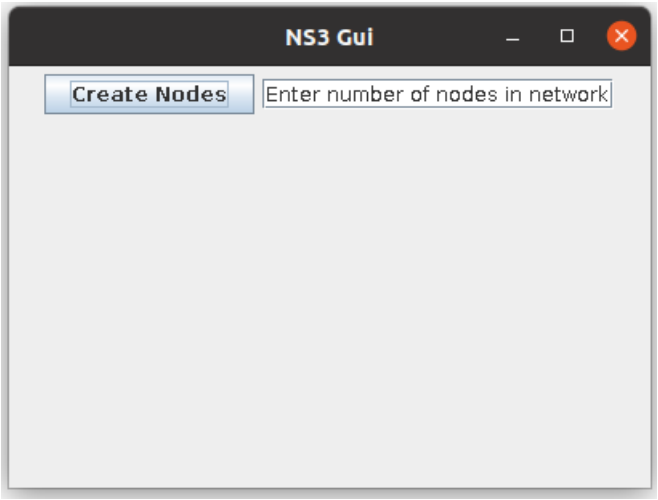

Figure 5.2: Initial GUI screen

Once the user inputs the number of nodes, the GUI renders a second screen that allows the user to enter node names and corresponding program files to run in the nodes. A screenshot of the second input screen is provided in Figure 5.3.

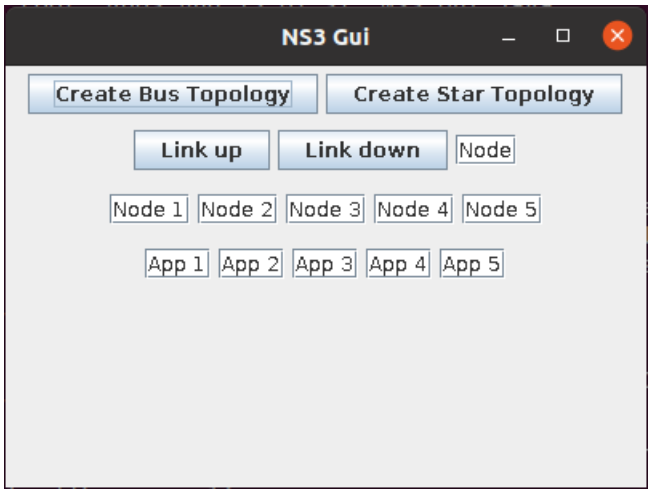

Figure 5.3: GUI screen for node name and exe

The current implementation can create bus and star topologies.

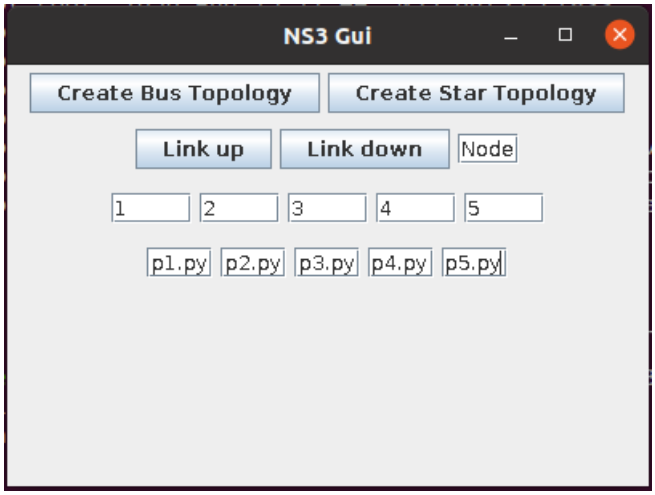

Figure 5.4: User inputs for node name and corresponding application

All user inputs are written into a JSON string which is later parsed by the main thread in the back end. The values parsed are used as arguments to the simulator function.

5.2 Communication Between Front and Back Ends

We considered a few options for inter process communication methods for the front and the back ends of the system:

- **RestAPI endpoints**: Using RestAPI (25) endpoints would enable the user to test their applications remotely via a web server. Although it was a practical and viable option, it would require managing network connections and possible network latency. For the goal of building a simple system to demonstrate a proof of concept, this was rather an overkill
- **TCP/IP socket connection**: Although it is relatively simple to build a socket connection between the front and back ends, it still involves managing the connection. Things are likely to get more complicated if for any reason we want to maintain multiple connections to the server.
- **Remote Procedure Calls and pipes**: We considered building a mechanism of communication using remote procedure calls like gRPC (26), pipes (27) and message queues like RabbitMQ (28). However, implementing these do involve a considerable amount of time.
- **File monitoring**: We considered a scenario in which, when an application is being tested extensively, the user may want to fire a long sequence of instructions and want to replay the entire sequence or part of the sequence more than once. Therefore, we

considered a case where we might want to provide a check-pointing mechanism to the user for replaying instructions to the simulator.

As none of the previous options satisfy this probable requirement, we came upon the shared file-system mechanism of sharing data between the back and front ends.We decided to go with a file system monitoring mechanism mainly because it can provide a simple and straightforward way of check-pointing operations performed by the simulator. In a large distributed system, we would like to perform a number of successive operations. In several scenarios, it may be necessary to replay all operations in sequence. In that case, the user does not need to manually input all previous instructions through the GUI.

5.2.1 File Monitoring System

The inotify Linux system call provides a way to monitor files and directories. An event is fired each time any change is made to the file or directory being monitored. For this system, we use inotify to monitor a specific directory. We have few minor changes and reused code in (29) that uses the inotify system call (30).

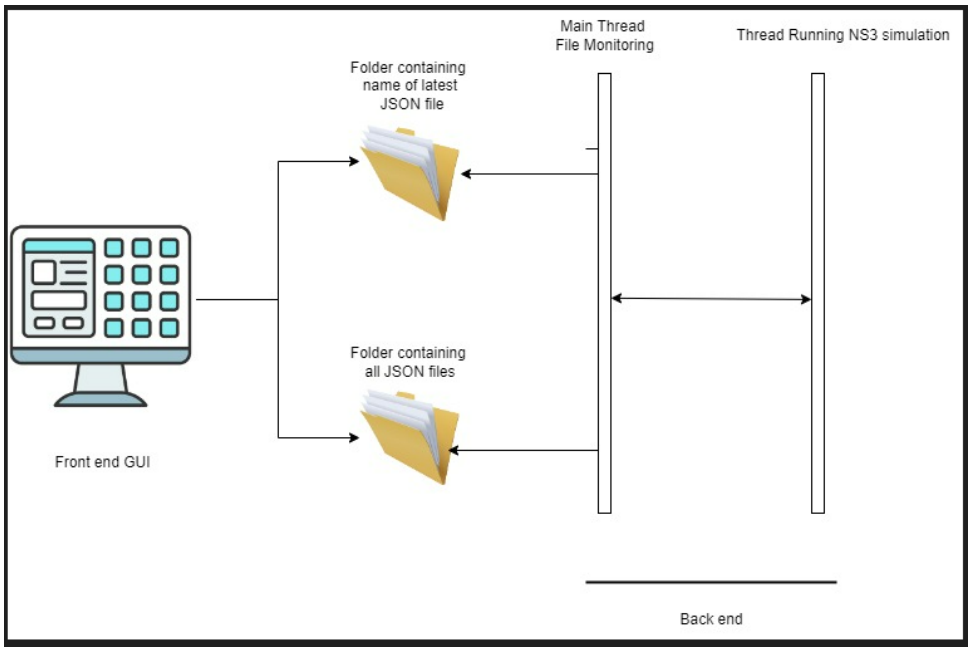

Figure 5.5: Communication Between Front and Back Ends

This has changed the overall design of the system, in that the main thread of the back-end is actually a directory monitoring system.Thus, every time a user inputs parameters:

- Front end writes parameters into a new JSON file and saved
- Front end writes the filename of the new file created into the file monitored by main thread at back end

- Main thread in back end gets the event of the file modified
- Main thread the latest filename written to the file being monitored and opens the latest file
- Main thread parses JSON string in the latest file and calls the simulator function with the parameters retrieved from the latest file as command line arguments.

The flow above is illustrated in the Figure 5.5.

We have paid careful attention to the synchronisation aspect of the above operations. We have made sure that the file monitoring system does not act upon the latest file before it is properly written and closed. Therefore, the file monitoring system only waits for the file close event. It reads the latest file name only after it is written into the file and closed.

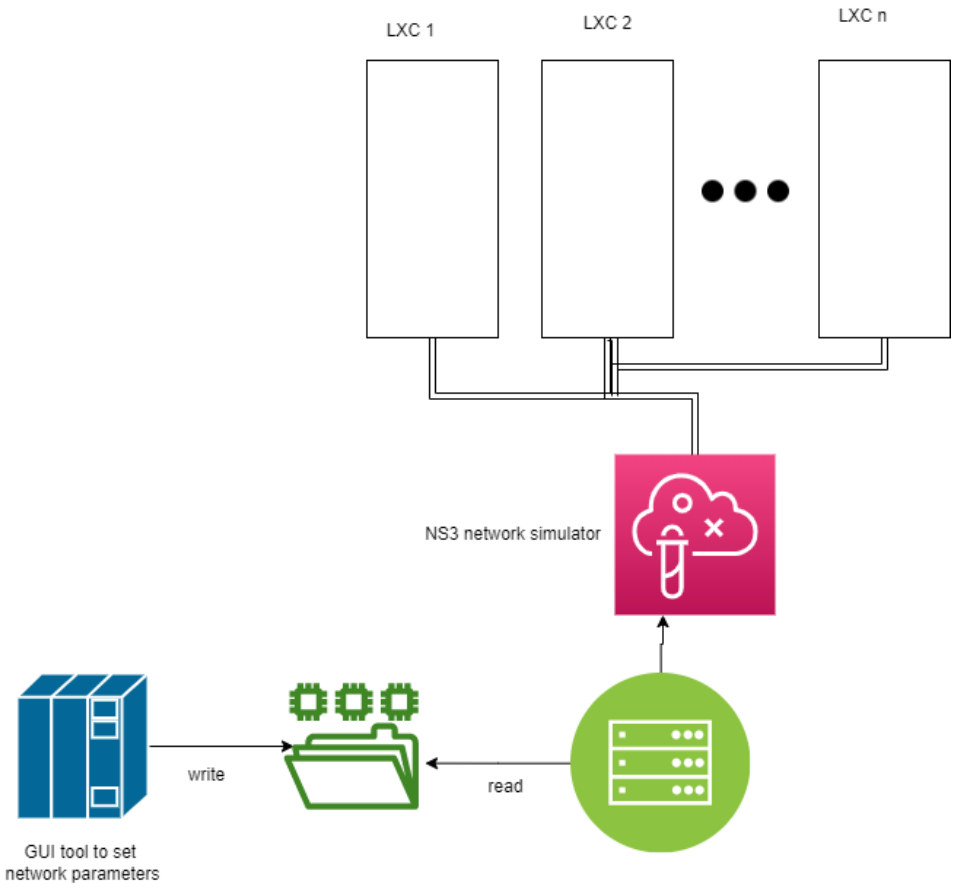

Figure 5.6: High Level Architecture

Now that we have discussed in detail about the individual components of the framework, we can take a look at the high level architecture of the system built. figure 5.6 illustrates the system overall. When the latest file is read, contents of the file are parsed, and the required simulator function is called with the input parameters.

6 Evaluation

We have the following criteria for evaluation of this framework:

- Creation of a variable number of nodes
- Running applications in the nodes
- Link up/down
- Creating and modifying topologies

We analyse these criteria individually in the sections that follow.

6.1 Creation of variable number of nodes

The first GUI screen requires the user to input the number of nodes needed. The input is then parsed and Linux containers are created . For a sample input of 5, we list the Linux containers to get the following output:

```
osboxes@osboxes:/Diss/ns-3-allinone/ns-3.36.1$ sudo lxc-ls -f
NAME STATE   AUTOSTART GROUPS IPV4       IPV6 UNPRIVILEGED
1    RUNNING 0         -      10.63.0.1  -    false
2    RUNNING 0         -      10.63.0.2  -    false
3    RUNNING 0         -      10.63.0.3  -    false
4    RUNNING 0         -      10.63.0.4  -    false
5    RUNNING 0         -      10.63.0.5  -    false
osboxes@osboxes:/Diss/ns-3-allinone/ns-3.36.1$
```

Figure 6.1: List of containers created

With the current implementation, we use a subnet of 24 and the IP addresses of the nodes are assigned beginning at 10.63.0.1. Due to the subnet, a maximum of 255 nodes can be created. Figure 6.1 shows the containers created after the user enters the input to create 5 containers.

6.2 Running applications in the nodes

Once nodes are created, a second GUI screen prompts the user to input a mapping of applications to run in nodes. Nodes are required to be named from "1". Node 1 has the IP address 10.63.0.1, node 2 has 10.63.0.2 and so on. We have coded the container creation in this way for the sake of simplicity.

For testing the ease of running applications in the containers, we initially created a client and a server using TCP sockets in Python. The server code listens continuously on the port and the client keeps sending a dummy message to the server. We have enabled packet capture (pcap) for testing and tracking the network behaviour. Pcap files are created by NS3 as the test runs, which are then read and interpreted using Wireshark (31).

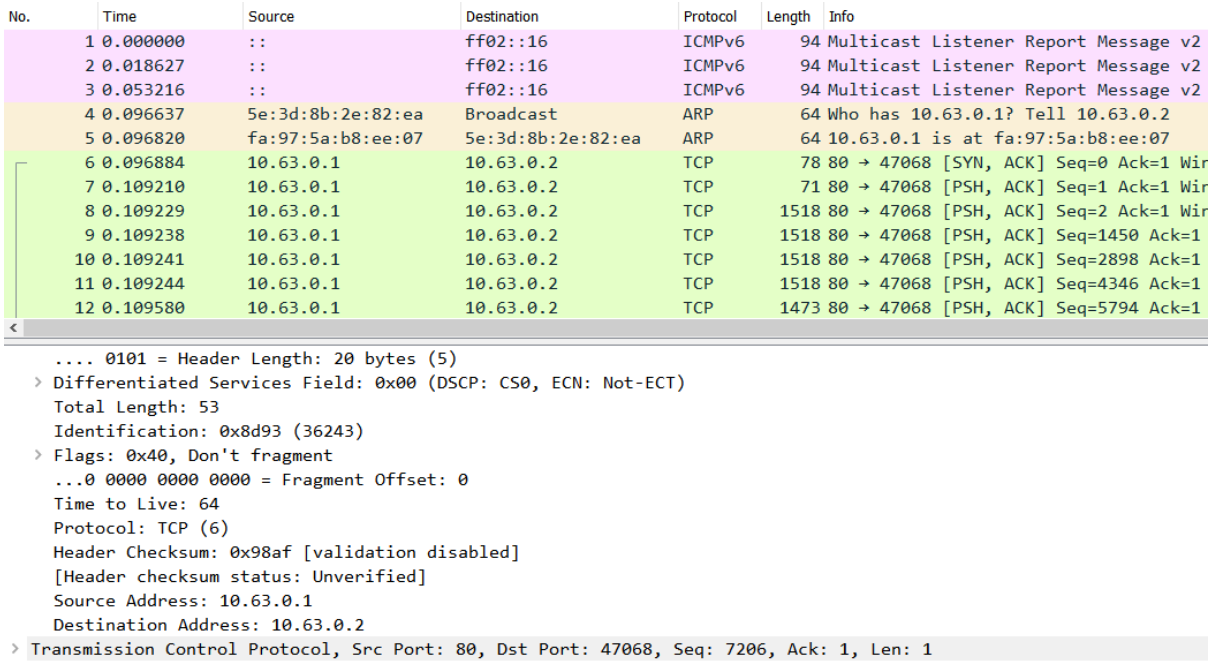

Figure 6.2: Packet capture for a simple client server interaction

We see in Figure 6.2 the client (10.63.0.2) sends data to the server (10.63.0.1). Clicking on any of the tabs in green, we can see the payload. We can see the dummy data being sent. This is demonstrated in Figure 6.3.

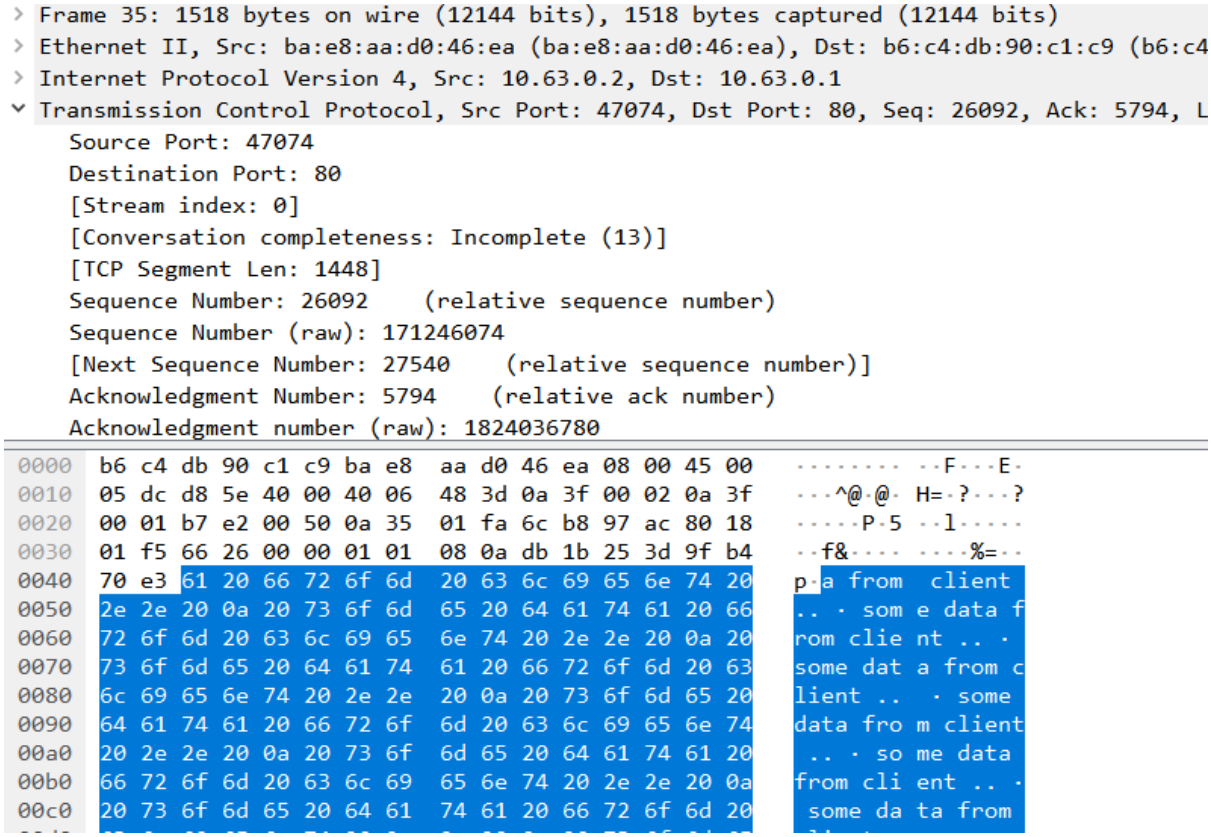

Figure 6.3: Data from client to server

6.3 Link up/down

Applications running on Linux containers are connected to NS3 via tap bridges. (6) specifies the steps in which we can use Linux commands to bring up/down the bridge connection. Thus when the user gives a link up/down instruction, the node name needs to be specified. The back-end then calls a shell script to call the required Linux commands to bring the bridge up/down.

The link up/down functionality was tested with a simple client and server application each running on separate containers connected to NS3 via tap bridges. The client continuously sends data to the server and the latter prints a message received log every time it gets a message from the client. The functionality was evaluated as follows:

- Server should stop receiving messages from client once the client tap-bridge is switched off
- Client should resume sending messages to the server once the client tap-bridge is switched on

Both the above criteria were met in the experiment that demonstrates the correct working of the link up/down functionality.

6.4 Creating and modifying topologies

In this work we have created the star and the bus toppologies. The bus topology does not have any complications and all nodes in the network can ping every other node.

The star topology creation needs a more detailed discussion. Once the simulator is running, the nodes need to discover the topology. NS3 provides a way to discover the same using broadcast mechanism where each nodes broadcasts its address and finds the hub it is connected to. NS3 imitates a real world star topology wherein each spoke node is placed in a different subnet. The network can be depicted in Figure 6.4. The nodes in the network are in the subnets 10.63.0.0, 10.64.0.0, 10.65.0.0, 10.66.0.0, 10.67.0.0. Therefore to make our applications connect to the correct IP address of the server, the back-end code needed to change. This brings us to an important conclusion: As network topologies differ, so does the arrangement of subnets. The code needs to handle a wide range of changing topologies and addresses and it can be very easy to have an address collision.

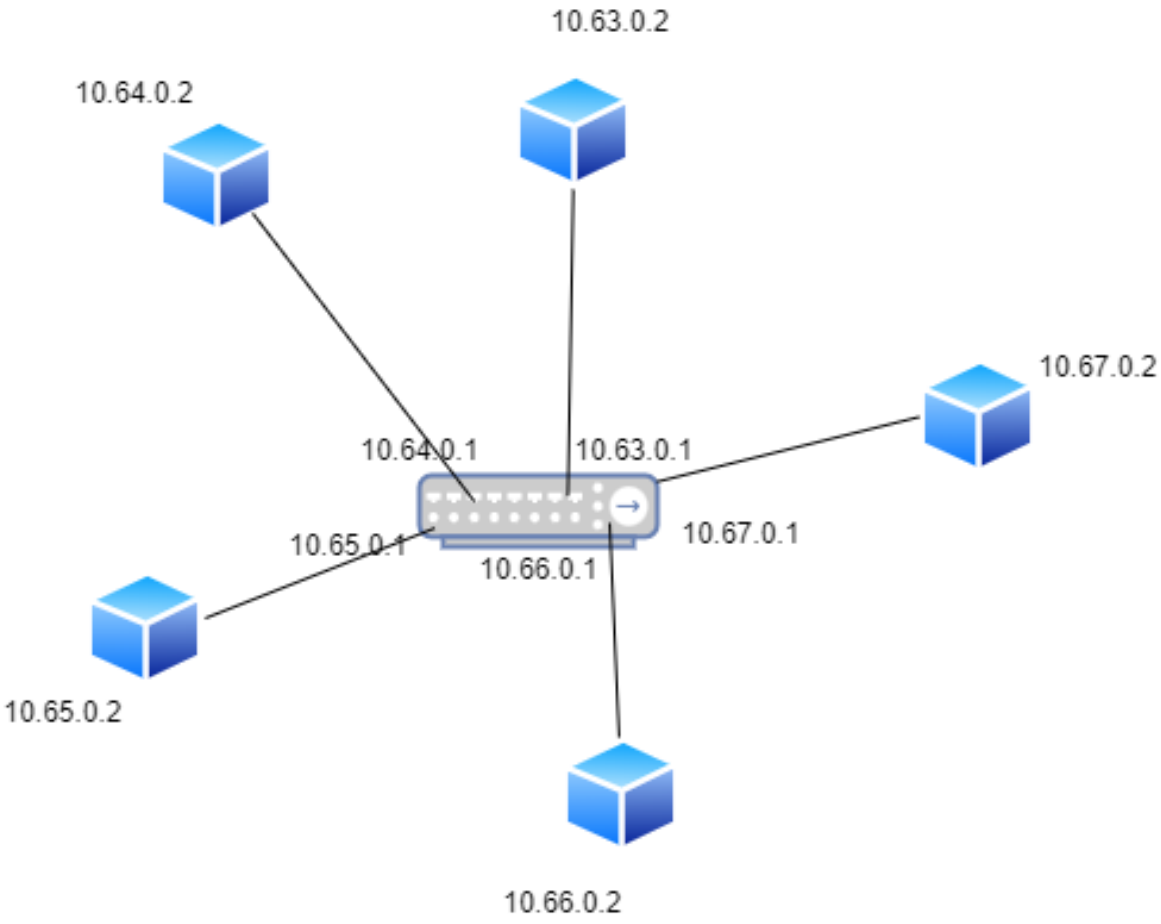

Figure 6.4: NS3 star topology

In the test scenario, we have created a star topology of 5 nodes, deployed a client

application in four of the nodes and a server application in the fifth node. The packet capture (pcap) files demonstrate how the topology is discovered by the nodes. For the topology depicted in 6.4, the nodes send broadcast message to do an address resolution as demonstrated by pcap files generated. Two such packet captures are shown in figures

	Time	Source	Destination	Protocol	Length	Info
1	0.000000	00:00:00_00:00:02	Broadcast	ARP	64	Who has 10.63.0.1? Tell 10.63.0.2
2	0.002011	00:00:00_00:00:01	00:00:00_00:00:02	ARP	64	10.63.0.1 is at 00:00:00:00:00:01
3	0.002011	10.63.0.2	10.63.0.1	TCP	74	49153 → 50000 [SYN] Seq=0 Win=65535 Len=0
4	0.007023	00:00:00_00:00:01	Broadcast	ARP	64	Who has 10.63.0.2? Tell 10.63.0.1
5	0.007023	00:00:00_00:00:02	00:00:00_00:00:01	ARP	64	10.63.0.2 is at 00:00:00:00:00:02
6	0.009036	10.63.0.1	10.63.0.2	TCP	74	50000 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=
7	0.009036	10.63.0.2	10.63.0.1	TCP	70	49153 → 50000 [ACK] Seq=1 Ack=1 Win=13107
8	0.070285	10.63.0.2	10.63.0.1	TCP	207	49153 → 50000 [ACK] Seq=1 Ack=1 Win=13107
9	0.072308	10.63.0.1	10.63.0.2	TCP	70	50000 → 49153 [ACK] Seq=1 Ack=138 Win=131
10	0.148571	10.63.0.2	10.63.0.1	TCP	207	49153 → 50000 [ACK] Seq=138 Ack=1 Win=131
11	0.226857	10.63.0.2	10.63.0.1	TCP	207	49153 → 50000 [ACK] Seq=275 Ack=1 Win=131

Figure 6.5: Address Resolution by Node 10.63.0.2

	Time	Source	Destination	Protocol	Length	Info
1	0.000000	00:00:00_00:00:0a	Broadcast	ARP	64	Who has 10.67.0.1? Tell 10.67.0.2
2	0.002011	00:00:00_00:00:09	00:00:00_00:00:0a	ARP	64	10.67.0.1 is at 00:00:00:00:00:09
3	0.002011	10.67.0.2	10.67.0.1	TCP	74	49153 → 50000 [SYN] Seq=0 Win=65535 Len=0 TSval=1000 TSe
4	0.007023	00:00:00_00:00:09	Broadcast	ARP	64	Who has 10.67.0.2? Tell 10.67.0.1
5	0.007023	00:00:00_00:00:0a	00:00:00_00:00:09	ARP	64	10.67.0.2 is at 00:00:00:00:00:0a
6	0.009036	10.67.0.1	10.67.0.2	TCP	74	50000 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSva
7	0.009036	10.67.0.2	10.67.0.1	TCP	70	49153 → 50000 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=10
8	0.075285	10.67.0.2	10.67.0.1	TCP	207	49153 → 50000 [ACK] Seq=1 Ack=1 Win=131072 Len=137 TSval=
9	0.077308	10.67.0.1	10.67.0.2	TCP	70	50000 → 49153 [ACK] Seq=1 Ack=138 Win=131072 Len=0 TSval=
10	0.153571	10.67.0.2	10.67.0.1	TCP	207	49153 → 50000 [ACK] Seq=138 Ack=1 Win=131072 Len=137 TSva
11	0.231857	10.67.0.2	10.67.0.1	TCP	207	49153 → 50000 [ACK] Seq=275 Ack=1 Win=131072 Len=137 TSva

Figure 6.6: Address Resolution by Node 10.67.0.2

6.4.1 Simplifying Creation of Topologies

With the challenges involved with developing a simple star topology, there inevitably arose a need to make topology creation simpler and more intuitive. The current star topology we have is overly simplistic and will be rather unhelpful in mimicking practical scenarios for the following reasons:

- Real world networks are an amalgamation of many kinds of network topologies. Subnets can be connected to bigger networks in numerous ways that can change the way nodes communicate in the network. This introduces several parameters that need to be taken as user inputs, for example which node is to be assigned as the hub. A very large network can have multiple hubs and creating the entire topology at one go is highly error-prone as well as impractical
- This inevitably makes both the front and back ends more complex, which in turn defeats the purpose of this thesis

Due to the limitations mentioned, we then aimed to create a functionality in which a user could create a simple point-to-point node and attach containers to it via tap bridges. This

would then allow the user to extend the simple point-to-point network to any kind of topology: mesh, star, ring or bus by adding one node at a time to the existing network topology. This is a more practical solution as it mimics real world networks better, because in reality different networks are merged into one big network very frequently. Conversely, parts of a given network are often divided into smaller networks.

NS3 provides a point-to-point helper class (32) that allows the user to create one node at a time and join it to other existing nodes. However, the point-to-point helper module does not support bridging. Also, the user cannot create a point-to-point node while the simulator is running. Therefore, point-to-point networking is only possible in NS3 when used without containers. This is a critical limitation of the NS3 framework because it takes away most of the flexibility around creating topologies.

6.5 Evaluation Summary

From the experiments conducted on the framework, we can claim that the framework can be used to successfully:

- Create a variable number of containers
- Run applications in the nodes that interact with NS3 and other nodes in the network
- Bring links up/down
- Create basic topologies

However, NS3 framework has three major limitations:

- Nodes cannot be created/destroyed on the fly, while the simulator runs. This means that the user must decide on the final topology and create all nodes before starting the simulator. This is not practical for testing large-scale networks where nodes and sub-networks are frequently added to/removed from existing networks.
- As nodes cannot be created/destroyed during simulator run-time, they must be provisioned before starting the simulator. Many nodes in the network can have containers running, while the tap bridge linking the containers to NS3 may be turned off to mimic the absence of nodes. This can consume a lot of computing resources that may render the framework not practical for industrial use.
- Although NS3 provides a point-to-point helper class that can be used to create a variety of network topologies, it does not support bridging. This eliminates the possibility of creating customized point-to-point networks involving application containers, thus critically hindering the ability of the framework to create topologies intuitively.

7 Conclusion and Further Work

In this thesis we have identified some of the challenges involved in the testing of large-scale applications involving network simulator frameworks. A detailed analysis of some of the most popular network simulators has led to an understanding of what they offer and in what ways they are limited. Our main approach was to ensure complete isolation between the network simulator code base and the test application framework. This was facilitated with the use of Linux containers.

We constructed a graphical user interface (GUI) through which a user can input some of the most important network parameters such as number of nodes, topology, and applications to be run in the containers created. The instructions given by the user are written into files in JSON format that are then parsed by the main thread in the back-end. The input parameters are read and passed to the simulator function using command like arguments.

As a proof of concept, we have successfully demonstrated the following key functionalities involved in testing any large-scale application:

- Node creation
- Running applications on created nodes
- Topology creating/modification
- Link up/down

The system can deliver the above functionalities relatively easily, as compared to performing the same operations in the traditional way. However, the NS3 framework has a few limitations that makes the development of more advanced functionalities not possible, or at least very complex. The following limitations were uncovered in this work:

- Point-to-point helper class does not support bridge mode, thus hindering the intuitiveness of topology creation.
- Once the simulator is started, it can only be stopped. No other operations can be performed on the simulation level.

Further work in this direction would involve making necessary changes to the NS3 framework

to eliminate the limitations revealed by this work. Another way to tackle the limitations would be to replace NS3 with OMNet++ and Linux containers with Docker, as the main reasons for choosing NS3 over OMNet++ was better overall performance and higher security through the use of Linux containers.

Bibliography

[1] Opnet network simulation. https://opnetprojects.com/opnet-network-simulation.

[2] András Varga. The omnet++ discrete event simulation system. *Proc. ESM'2001*, 9, 01 2001.

[3] Teerawat Issariyakul and Ekram Hossain. Introduction to network simulator 2 (ns2). In *Introduction to network simulator NS2*, pages 1–18. Springer, 2009.

[4] E. Weingartner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *2009 IEEE International Conference on Communications*, pages 1–5, 2009. doi: 10.1109/ICC.2009.5198657.

[5] Marek Moravcik, Pavel Segec, Martin Kontsek, Jana Uramova, and Jozef Papan. Comparison of lxc and docker technologies. In *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 481–486, 2020. doi: 10.1109/ICETA51985.2020.9379212.

[6] Ns3lxc. https://www.nsnam.org/wiki/HOWTO_Use_Linux_Containers_to_set_up_virtual_networks.

[7] Sparsh Mittal. Opnet: An integrated design paradigm for simulations. *Software Engineering : An International Journal (SEIJ)*, pages 68–84, 09 2012.

[8] Hongji Yang Zheng Lu. *Unlocking the Power of OPNET Modeler*. Cambridge, London, 2012.

[9] Andras Varga. A practical introduction to the omnet++ simulation framework. In *Recent advances in network simulation*, pages 3–51. Springer, 2019.

[10] Omnet++ gui in docker. https://omnetpp.org/articles/2019/07/04/omnetpp-docker.html.

[11] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5 (5):440–452, 1979. doi: 10.1109/TSE.1979.230182.

[12] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14 (14):527, 2008.

[13] netanim. https://www.nsnam.org/wiki/NetAnim#:~:text=NetAnim.

[14] yaml. https://github.com/buzz66boy/ns3-lxc/blob/master/README.md.

[15] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. A capability-based security approach to manage access control in the internet of things. *Mathematical and Computer Modelling*, 58(5):1189–1205, 2013. ISSN 0895-7177. doi: https://doi.org/10.1016/j.mcm.2013.02.006. URL https://www.sciencedirect.com/science/article/pii/S089571771300054X. The Measurement of Undesirable Outputs: Models Development and Empirical Analyses and Advances in mobile, ubiquitous and cognitive computing.

[16] ns3concepts. https://www.nsnam.org/docs/tutorial/html/conceptual-overview.html.

[17] cgroups. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[18] namespaces. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[19] jail. https://en.wikipedia.org/wiki/FreeBSD_jail/.

[20] Ns3docker. https://sites.google.com/thapar.edu/ramansinghtechpages/step-wise-establishing-connection.

[21] Dsec. https://cloud.redhat.com/blog/detecting-docker-exploits-and-vulnerabilities-your-how-to-guide.

[22] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016. doi: 10.1109/MCC.2016.100.

[23] Privesc. https://www.sciencedirect.com/topics/computer-science/privilege-escalation.

[24] Javaswing. https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html.

[25] rest. https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[26] grpc. grpc.io.

[27] pipe. https://www.gnu.org/software/libc/manual/html_node/Creating-a-Pipe.html.

[28] Mq. https://www.rabbitmq.com/.

[29] inotify. https://github.com/jrelo/fs_monitoring/blob/master/inotify-example.c, .

[30] inotifyman. https://man7.org/linux/man-pages/man7/inotify.7.html, .

[31] wireshark. https://www.wireshark.org/.

[32] p2phelp. https://www.nsnam.org/doxygen/classns3_1_1_point_to_point_helper.html.