Master Thesis

Integration of an LXC-based Node Type into the CORE Network Emulator

Submitted by: Mahnaz Rahman
First examiner: Prof. Dr. Armin Lehmann
Second examiner: Prof. Dr.- Ing. Ulrich Trick
Date of start: 08.02.2019
Date of submission: 29.07.2019

Statement

I confirm that I have written this thesis on my own. No other sources were used except those referenced. Content which is taken literally or analogously from published or unpublished sources is identified as such. The drawings or figures of this work have been created by myself or are provided with an appropriate reference. This work has not been submitted in the same or similar form or to any other examination board.

_____

Date, signature of the student

Contents

1 Introduction

The main objective of this thesis is to develop an extension to support LXC container as node type in the CORE network Emulator. This research focuses on extending the implementation of default CORE node type to LXC node type and then retrieve some possible configurable aspects for LXC node such as container image selection, specify a limitation for CPU and RAM usage.

The names of the main chapters of this thesis document are named as followed:

1. Introduction
2. Theoretical Background
3. Requirements Analysis
4. Realisation
5. Summary and Perspectives
6. Abbreviations
7. References

2 Theoretical Background

This chapter explains the theoretical background of all technologies which have been used in this thesis work. In order to realise and evaluate the research work, it is essential to get some insight of different possible technologies, methods and techniques. This chapter consists of the explanation of OS kernel and its early limitations, a short introduction to container virtualisation and some related aspects of this will be explained. Additionally, some theoretical concept of LXC container and some important features of LXC container will be covered in this chapter as this is the major part of the thesis. Lastly, another main component which is called CORE Network Emulator will be presented and some description of required python modules, LXC-python bindings and Distrobuilder will be provided.

2.1 OS Kernel and Its Early Limitations

The evolution of LXC container is the outcome of early OS problems such as managing memory, I/O and process scheduling in most effective way. In the past, not more than one single process could be possible to schedule for task. It got wasted due to blocked on some I/O operation. The solution to this problem was to develop better CPU schedulers, so that more task can be allocated in a fair way for ensuring maximum CPU utilization. Although the modern schedulers called Completely Fair Schedulers (CFS) in Linux handles the work quite well in terms of allocating fair amounts of time to each process, there is still prevailed priority-based allocation to a process and its subprocesses. Traditionally, this can be accomplished by the `nice()` system call or real-time scheduling policies. However, there are limitations to the level of granularity or control that can be achieved (Ivanov, K., 2017).

Similarly, before the advent of virtual memory, multiple processes would allocate memory from a shared pool of physical memory. The virtual memory supplied some form of memory isolation per process, in the sense that processes would have their own address space and extend the available memory by means of a swap, but still there was no good approach of limiting the memory usage by each process and its children can use (Ivanov, K., 2017).

To further complicate the matter, running different workloads on the same physical server usually resulted in a negative impact on all running services. A memory leak or a kernel panic could cause one application to bring the entire operating system down. For example, a web server that is mostly memory bound and a database service that is I/O heavy running together became problematic. To avoid such scenarios, system administrators would separate the various applications between a pool of servers, leaving some machines underutilized, especially at certain times during the day, when there was not much work to be done. This is a similar problem as a single running process blocked on I/O operation is a waste of CPU and memory resources. Eventually, the solution to this problem is the use of hypervisor-based virtualisation, containers or the combination of both (Ivanov, K., 2017).

In this thesis work, LXC container has been used. Therefore, the further discussion is based on LXC container.

2.2 Introduction to Container Virtualisation

Container virtualisation can be defined as a single operating system image, bundling a set of isolated applications and their dependent resources so that they run separated from the host machine. There may be multiple such containers running within the same host machine (Kumaran, S., 2017). Container virtualisation is divided into two kinds. One is Operating system-level virtualisation, and another is Application-level virtualisation.

Container-based virtualisation utilizes kernel features to create an isolated environment for processes. In contrast to hypervisor-based virtualisation, containers do not get their own virtualised hardware but use the hardware of the host system. Therefore, software running in containers does directly communicate with the host kernel and has to be able to run on the operating system and CPU architecture the host is running on. Not having to emulate

hardware and boot a complete operating system enables containers to start in a few milliseconds and be more efficient than classical virtual machines. Container images are usually smaller than virtual machine images because container images do not need to contain a complete tool chain to run an operating system, for example, device drivers, kernel or the init system.This is one of the reasons why container-based virtualisation gained more popularity over the last few years. The small resource fingerprint allows better performance on a small and larger scale and there are still relatively strong security benefits (Eder, M., 2016). Figure 2.1 shows that on a single host there are two containers are running. Containers are running in userspace, separated from each other. Container can itself create the environment to run the applications.

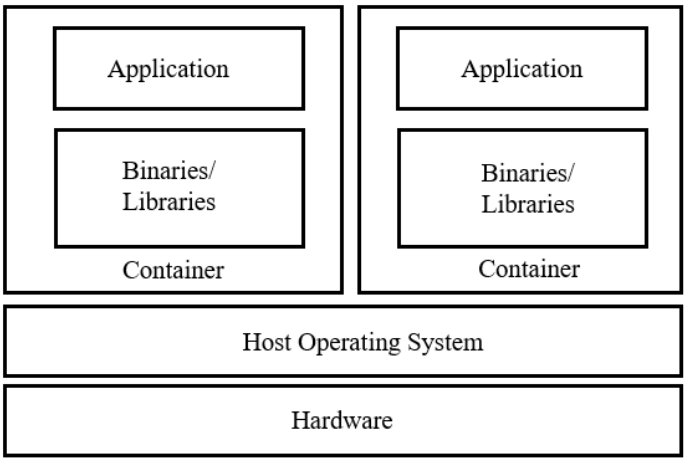

Figure 2.1 Container-based Virtualisation Based on (Sikeridis, D. et al, 2017)

2.2.1 History of Containerisation

Virtualisation was developed as an effort to fully utilize available computing resources. Virtualisation enables multiple virtual machines to run on a single host for different purposes with their own isolated space. Virtualisation achieved such isolated operating system environments using hypervisors, computer software that sits in between the host operating system and the guest or the virtual machine's operating system. Container technology has existed for a long time in different forms, but it has significantly gained popularity recently in the Linux world with the introduction of native containerisation support in the Linux kernel. The earliest version of container technology back in year 1982, Unix operating system introduced Chroot, which provides filesystem isolation by switching the root directory for running processes and children (Kumaran, S., 2017).

Then in 2000, FreeBSD Jail came to improve the system. In 2001, Linux VServer took from the above jail system and enhanced it by securely partitioning resources on any computer system. Linux kicked off a long wave of other container technologies, including Solaris containers in 2004. Open VZ was yet another in this container string and somewhat like Solaris. Google got into the container business by 2006 with their Process Containers. One year later, they renamed this Control Groups and added to the Linux Kernel (Kumaran, S., 2017).

Common modern-day containers are descended from LXC, which was first introduced in 2008. LXC was possible due to some key features added to the Linux kernel (Kumaran, S., 2017).

2.2.2 Difference between Containers and Virtual Machines

From the previous discussion it is clear that Container is one of the most popular virtualisation technology now. But there are other virtualization techniques which are widely used such as virtual machines. But there are some basic difference in between virtual machines and containers. In below it is widely discussed.

Containers and virtual machines are two ways to deploy multiple, isolated services on a single platform. They both provide a way to isolate applications and provide a virtual platform for application to run on. This concept is explained in below Figure 2.2

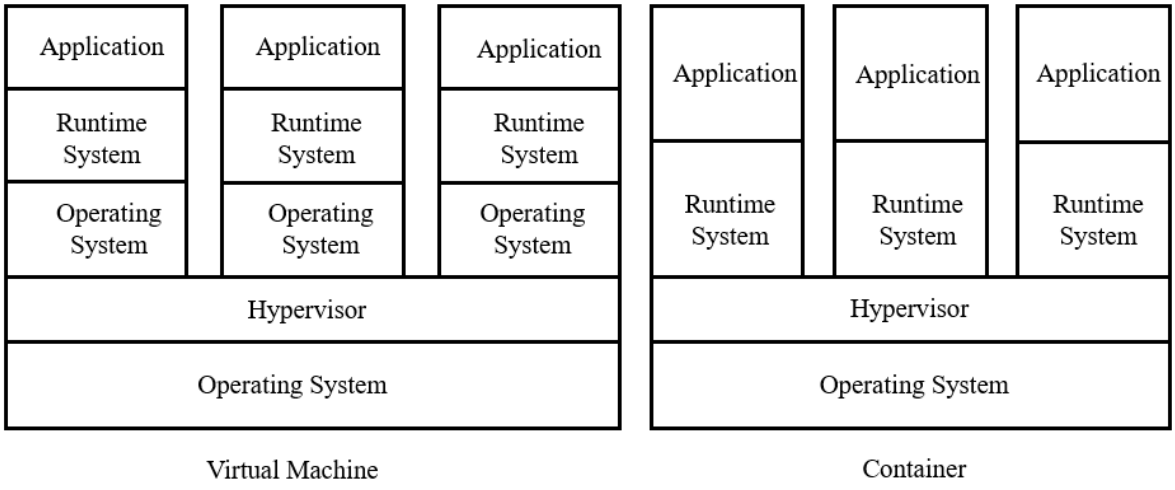

Figure 2.2 Comparison between Container and Virtual Machine Architecture Based on (Kelly, D., 2016)

Virtual machines (VM) are managed by a hypervisor and utilize VM hardware, while container systems provide operating system services from the underlying host and isolate the applications using virtual-memory hardware. In a nutshell, a VM provides an abstract machine that uses device drivers targeting the abstract machine, while a container provides an abstract OS. A para-virtualised VM environment provides an abstract Hardware Abstraction Layer (HAL) that requires HAL-specific device drivers. Applications running in a container environment share an underlying operating system, while VM systems can run different operating systems. Typically, a VM will host multiple applications whose mix may change over time versus a container that will normally have a single application. However, it's possible to have a fixed set of applications in a single container (Kelly, D., 2016).

Containers provide many advantages over VMs, although some can be addressed using other techniques. One advantage is the low overhead of containers and, therefore, the ability to start new containers quickly. This is because starting the underlying OS in a VM takes time, memory, and the space needed for the VM disk storage. Normally, a VM needs at least one unique image file for every running instance of a VM. It contains the OS and often the application code and data as well. Much of this is common among similar VMs. In the case of a raw image, a complete copy of the file is needed for each instance. This could require copying multiple gigabytes per instance. an initial instance of the VM is set up and the operating system is installed possibly with additional applications. On the other hand, container does not need a host operating system to initialize the application. So, the overhead is less in container than a VM (Kelly, D., 2016). Because of this it is possible to run more containers than VM in a specific hardware.

2.2.3 LXC Container

LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers (Linux containers, 2019). LXC containers is an operating system-level virtualisation method for running multiple isolated Linux system or containers on a single control host (LXC host). It does not provide a virtual machine but rather provides a virtual environment that has its own CPU, memory, block I/O, network etc. space and resource control mechanism. This is provided by namespaces and cgroups features in Linux kernel on LXC host. It has similarity to chroot but offers much more isolation (archlinux, 2019).

To create different operating systems containers, templates are used to bootstrap specific operating system. The templates which are provided in LXC are script specific to an operating system. Each operating system that is supported by the LXC installation has a script dedicated to it. There is a generic script called “download” that can install many different operating systems with a common interface. LXC supported some Linux distributions are for example, Ubuntu, Alpine, CentOS, Debian and some supported architectures are amd64, i386, armel, arm64 (Kumaran, S., 2017).

LXC can be used in two distinct ways - privileged, by running the lxc commands as the root user; or unprivileged, by running the lxc commands as a non-root user. Although starting of unprivileged containers by the root user is possible, but this will be discussed more detail in later chapter. Unprivileged containers are more limited, for instance being unable to create device nodes or mount block-backed filesystems. However they are less dangerous to the host, as the root userid in the container is mapped to a non-root userid on the host (ubuntu documentation, 2019). Generally, LXC life cycle has the following steps (Kumaran, S., 2017):

`lxc-create`: this command will create the container with a given opearting system template and options

`lxc-start`: this command will start the container which has been created with the above mentioned command.

`lxc-ls`: this command is used to display the list of all containers in the host system

`lxc-attach`: this command can be used to obtain a default shell session inside a container

`lxc-stop:` this command can be used when the container needs to be stopped from a running state.

`lxc-destroy:` this command is applicable when the container is no longer is required to be used, so it can be destroyed.

Figure 2.3 presents the life cycle of LXC container with different state.

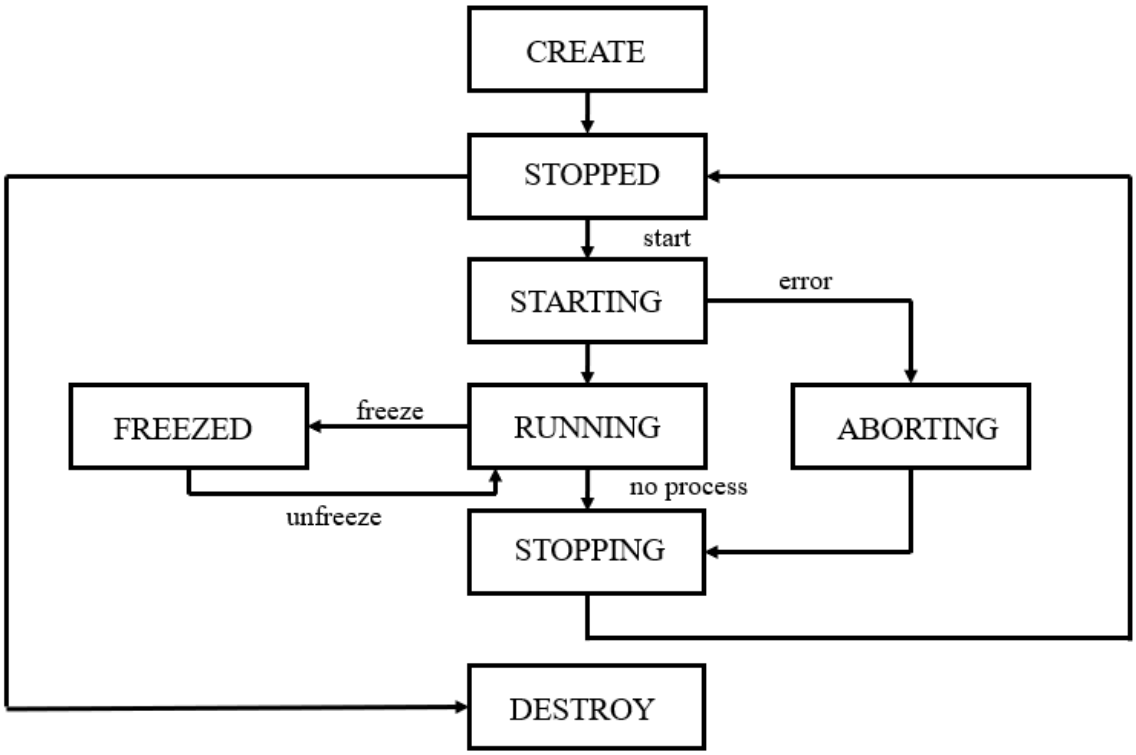

Figure 2.3 LXC Container Lifecycle Based on (Kumaran, S., 2017)

In this Figure 2.3, it shows that a container is being created and after the creation it does not automatically start. It stays in “Stopped” state. Then, “`lxc-start`” command starts the container and it reached to “Running” state. If any problem occurs during the starting phase, it aborts the execution of the command and goes back to “Stopped” state. In addition to this, LXC has a feature to freeze all the processes which is already running inside of a container. All the processes will be in a blocked state until it goes back to unfreeze state. This feature can be useful when the system load is high and it is necessary to free some resources without stopping the container and preserve the running state (Ivanov, K., 2017).

2.2.4 LXC Container’s Feature

LXC containers are often considered as something in the middle between a chroot and a full fledged virtual machine. The objective of LXC is to create an environment as much close as possible to a standard Linux installation but without the need for a separate kernel (Linux containers, 2019).

Some of the main features of an LXC container are following:

- Security is one of the major essential features of a container. Network services can be run in a container, which limits the damage caused by a security breach or violation. An intruder who successfully exploits a security hole on one of the applications running in that container is restricted to the set of actions possible within that container (Linux Journal, 2018).

- Isolation is another important feature of an LXC container. Containers allow the deployment of one or more applications on the same physical machine, even if those applications must operate under different domains, each requiring exclusive access to its respective resources. For instance, multiple applications running in different containers can bind to the same physical network interface by using distinct IP addresses associated with each container (Linux Journal, 2018).

- Virtualisation and transparency are the concern of LXC container. Containers provide the system with a virtualised environment that can hide or limit the visibility of the physical devices or system's configuration underneath it. The general principle behind a container is to avoid changing the environment in which applications are running with the exception of addressing security or isolation issues (Linux Journal, 2018).

2.2.5 Privileged and Unprivileged Container

LXC containers can be of two types: privileged container and unprivileged container.

Privileged containers are denoted as any container, where the UID 0 is mapped to the host’s UID 0. In such case, protection of the host and prevention of escape is entirely done through Mandatory Access Control (AppArmor, SElinux, Seccomp filters, dropping of capabilities and namespaces. These technologies combined will typically prevent any unwanted damage of the host, where damage is defined as things like reconfiguring host hardware, reconfiguring the host kernel or accessing the host filesystem (Linux containers, 2019). In this state, containers run with root permission (Linux containers, 2019).

Unprivileged containers are safe by design. The container UID 0 is mapped to an unprivileged user outside of the container and only has extra rights on resources that it owns itself. Unprilvileged containers run with non-root permission. The advantage of using unprivileged containers is if the security gets compromised or violated, it can be used with extremely limited privileges. With such container, the use of SELinux, AppArmor, Seccomp and capabilities isn't necessary for security. LXC will still use those to add an extra layer of security which may be handy in the event of a kernel security issue but the security model isn't enforced by them. To make unprivileged containers work, LXC interacts with 3 pieces of setuid code: (Linux containers, 2019).

- lxc-user-nic (setuid helper to create a veth pair and bridge it on the host)
- newuidmap (from the shadow package, sets up a UID map)

- newgidmap (from the shadow package, sets up a GID map)

unprivileged containers do not require to be owned by the user since they run in user namespace. this a kernel feature that allows the mapping of a UID of a physical host into a namespace inside where a user with a UID 0 can exist. As a result, most security issues (container escape, resource abuse) in those containers will apply just as well to a random unprivileged user and so would be a generic kernel security bug rather than a LXC issue (Linux containers, 2019).

2.2.6 Linux Namespaces

Namespaces are the foundation of lightweight process virtualization. They enable a process and its children to have different views of the underlying system. This is obtained by the addition of the `unshare()` and `setns()` system calls, and the inclusion of six new constant flags to the `clone()`, `unshare()` and `setns()` system calls (Ivanov, K., 2017):

- `Clone()`: this system call creates a new process and attaches it to a new specified namespace.
- `unshare()`: this attaches the current process to a new specified namespace
- `setns()`: this attaches a process to an already existing namespace.

There are six namespaces currently in use by LXC, with more being developed.

- Mount namespaces, specified by the `CLONE_NEWNS` flag
- UTS namespaces, specified by the `CLONE_NEWUTS` flag
- IPC namespaces, specified by the `CLONE_NEWIPC` flag
- PID namespaces, specified by the `CLONE_NSEWPID` flag
- User namespaces, specified by the `CLONE_NEWUSER` flag
- Network namespaces, specified by the `CLONE_NEWNET` flag

More detailed description of these namespaces is given below (Ivanov, K., 2017).

Mount Namespaces
Mount namespaces first appeared in kernel 2.4.19 in 2002 and provided a separate view of the filesystem mount points for the process and its children. When mounting or unmounting a filesystem, the change will be noticed by all processes because they all share the same default namespace. When the CLONE_NEWNS flag is passed to the `clone()` system call, the new process gets a copy of the calling process mount tree that it can then change without affecting the parent process. From that point on, all mounts and unmounts in the default namespace will be visible in the new namespace, but changes in the per-process mount namespaces will not be noticed outside of it (Ivanov, K., 2017).

In the context of LXC, mount namespaces are useful because they provide a way for a different filesystem layout to exist inside the container. It is worth mentioning that before the mount namespaces, a similar process confinement could be achieved with the `chroot()` system call, however `chroot` does not provide the same per-process isolation as mount namespaces do (Ivanov, K., 2017).

UTS namespaces
Unix Timesharing or UTS namespaces provide isolation for the hostname and domain name, so that each LXC container can maintain its own identifier as returned by the `hostname -f` command. This is needed for most applications that rely on a properly set hostname (Ivanov, K., 2017).

To create a bash session in a new UTS namespace, the `unshare` utility can be used again, which uses the `unshare()` system call to create the namespace and the `execve()` system call to execute bash in it (Ivanov, K., 2017).

IPC namespaces
The Inter-process Communication or IPC namespaces provide isolation for a set of IPC and synchronization facilities. These facilities provide a way of exchanging data and synchronizing the actions between threads and processes. They provide primitives such as semaphores, file locks, and mutexes among others, that are needed to have true process separation in a container (Ivanov, K., 2017).

PID Namespaces
The Process ID or PID namespaces provide the ability for a process to have an ID that already exists in the default namespace, for example an ID of 1. This allows for an init system to run in a container with various other processes, without causing a collision with the rest of the PIDs on the same OS (Ivanov, K., 2017).

User namespaces
The user namespaces allow a process inside a namespace to have a different user and group ID than that in the default namespace. In the context of LXC, this allows for a process to run as root inside the container, while having a non-privileged ID outside. This adds a thin layer of security, because braking out for the container will result in a non-privileged user. This is possible because of kernel 3.8, which introduced the ability for non-privileged processes to create user namespaces (Ivanov, K., 2017).

Network Namespace
Namespace is one of the prime kernel feature of containerisation technique. Among them, network namespace provides a brand-new network stack for all the process within the namespace (Garcia, D., 2016).

Network namespaces provide isolation of the networking resources, such as network devices, addresses, routes, and firewall rules. This effectively creates a logical copy of the network stack, allowing multiple processes to listen on the same port from multiple namespaces. This is the foundation of networking in LXC and there are quite a lot of other use cases where this can come in handy. For example, the `iproute2` package provides very useful userspace tools that can be used to experiment with the network namespaces and is installed by default on almost all Linux systems (Ivanov, K., 2017). From a system's point of view, using `clone()` system call, a namespace will be created, passing the flag CLONE_NEWNET will create a new network namespace into the new process. On the other hand, a simple tool IP or the package name `iproute2` can create a new persistent network namespace (Garcia, D., 2016). Figure 2.4 outlined that when the Linux starts, a new network namespace will be created and every newly created process will inherit the namespace from its parent. Therefore, all the created process will inherit the network namespace used by init (PID 1) (Bregman, A. 2016). In this thesis work, network namespace is an important aspect.

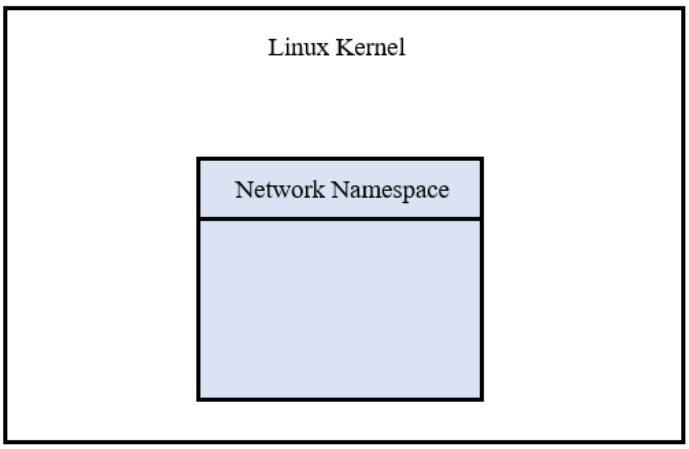

Figure 2.4 Creation of Network Namespace Based on (Bregman, A. 2016)

2.2.7 Filesystem or Rootfs

The next component needed for a container is the disk image, which provides the root filesystem (rootfs) for the container. The rootfs consists of a set of files, similar in structure to the filesystem mounted at root on any GNU/Linux-based machine. The size of rootfs is smaller than a typical OS disk image, since it does not contain the kernel. The container shares the same kernel as the host machine. A rootfs can further be reduced in size by making it contain just the application and configuring it to share the rootfs of the host machine. Using copy-on-write (COW) techniques, a single reduced read-only disk image may be shared between multiple containers (Kumaran, S., 2017).

2.2.8 Control Groups (Cgroup)

Cgroups are one of the most important kernel features that allows fine-grained control over resource allocation for a single process, or a group of processes, called tasks. In the context of LXC this is quite important, because it makes it possible to assign limits to how much memory, CPU time, or I/O, any given container can use (Kumaran, S., 2017).

Control groups, usually referred to as cgroups, allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on) (cgroups, 2019). All the cgroup items required to be mounted in `/sys/fs/cgroup` path location

Some of the available resources are (Carvalho, A., 2017):

- blkio: Block IO controller - limit I/O usage
- cpuacct: CPU accounting controller - accouting for CPU usage
- cpuset: CPU set controller - allows assigning a set of CPUs and memory nodes to a set of tasks
- memory: Memory resource controller - limit memory usage
- hugeTLB: Huge TLB controller - allows limiting the usage of huge pages
- devices: Device whitelist controller - enforce open and mknode restrictions on device files
- pids: Process number controller - limit the number of tasks

Limiting I/O Throughput
To limit I/O, cgroup needs to be mounted on blkio controller. The Block I/O (blkio) subsystem controls and monitors access to I/O on block devices by tasks in cgroups. Writing values to some of these pseudofiles limits access or bandwidth, and reading values from some of these pseudofiles provides information on I/O operations (Red Hat, 2019).

The blkio subsystem offers two policies for controlling access to I/O:

- Proportional weight division — implemented in the Completely Fair Queuing (CFQ) I/O scheduler, this policy allows you to set weights to specific cgroups. This means that each cgroup has a set percentage (depending on the weight of the cgroup) of all I/O operations reserved (Red Hat, 2019).
- I/O throttling (Upper limit) — this policy is used to set an upper limit for the number of I/O operations performed by a specific device. This means that a device can have a limited rate of read or write operations (Red Hat, 2019).

Limiting Memory Usage
The memory subsystem controls how much memory is presented to and available for use by processes. This can be particularly useful in multitenant environments where better control over how much memory a user process can utilize is needed, or to limit memory hungry applications. Containerized solutions like LXC can use the memory subsystem to manage the size of the instances, without needing to restart the entire container. The memory subsystem performs resource accounting, such as tracking the utilization of anonymous pages, file caches, swap caches, and general hierarchical accounting, all of which presents an overhead. Because of this, the memory cgroup is disabled by default on some Linux distributions (Ivanov, K., 2017).

The CPU and CPUSET Subsystems
The CPU subsystem schedules CPU time to cgroup hierarchies and their tasks. It provides finer control over CPU execution time than the default behaviour of the CFS. The cpuset subsystem allows for assigning CPU cores to a set of tasks, similar to the taskset command in Linux. The main benefits that the cpu and cpuset subsystems provide are better utilization per processor core for highly CPU bound applications. They also allow for distributing load between cores that are otherwise idle at certain times of the day. In the context of multitenant environments, running many LXC containers, cpu and cpuset cgroups allow for creating different instance sizes and container flavours, for example exposing only a single core per container, with 40 percent scheduled work time (Ivanov, K., 2017).

2.2.9 LXC Capabilities

Software is deemed to be production ready when it is secure. Security is highly important when dealing with operating systems, which is clearly what LXC provides using system-level containers. The basic idea of Linux containers is to share resources with isolated environments, and this raises a question about security. When LXC started, many security considerations went unaddressed, but as LXC has evolved, many security features have been added, and the latest releases of LXC are considered secure based on certain recommended configurations. Some of the security features will be explored in this chapter (Kumaran, S., 2017).

The Linux capabilities support can be used to control the list of capabilities that should be retained or dropped when the container starts. The capabilities support was added to the Linux kernel from version 2.2; Linux divides the privileges available to “root” into distinct units. In the LXC configuration file, the following values are set in order to retain or drop capabilities (Kumaran, S., 2017):

`lxc.cap.drop` - Specify the capability to be dropped in the container. A single line defining several capabilities with a space separation is allowed. The format is the lower case of the capability definition without the "CAP_" prefix, eg. CAP_SYS_MODULE should be specified as sys_module (Linux containers, 2019).

`lxc.cap.keep` - Specify the capability to be kept in the container. All other capabilities will be dropped. When a special value of "none" is encountered, lxc will clear any keep capabilities specified up to this point. A value of "none" alone can be used to drop all capabilities (Linux containers, 2019).

2.3 CORE Network Emulator

Common Open Research Emulator which is abbreviated as CORE is a tool for building virtual network (CORE, 2015). CORE works as real-time network emulator that allows rapid instantiation of hybrid topologies composed of both real hardware and virtual network nodes (Ahrenholz et al, 2008). Emulated network can be connected to physical networks and routers as it is a live-running emulation. Moreover, it provides an environment for running real applications and protocols, and taking advantage of virtualisation offered by Linux or FreeBSD operating systems.

Some key features of CORE are:

- Efficient and scalable
- Easy to use GUI
- Highly customizable
- Runs applications and protocols without modifying them

In general, CORE is used for network and protocol research, demonstrations, application and platform testing, evaluating network scenarios, security studies and increasing the size of physical test networks (CORE, 2015).

2.3.1 CORE Architecture

CORE consists of two main components. They are: CORE GUI and CORE Daemon. CORE Daemon is the backend of the tool and it manages emulation session of virtual nodes and networks, of which other scripts and utilities may be used for further control. Using kernel virtualisation, it builds emulated network for virtual nodes and some form of bridging and packet manipulation for virtual networks. The nodes and networks connect to each other via interfaces which is installed on nodes (CORE, 2015).

The CORE Daemon is controlled via the graphical user interface (GUI). GUI works as the frontend of the system. The Daemon contains several python modules that can be imported directly by python scripts and GUI is scripted with Tcl/Tk language. The GUI and Daemon make the communication using a custom, asynchronous, socket-based API called CORE API. Figure 2.5 depicts that the components the user interacts with are coloured blue are: GUI, scripts and `vcmd` command line tool. Vcmd command-line tool is used to connect to the `vnoded` daemon in a Linux network namespace, for running commands in the namespace. the CORE Daemon uses the same channel for setting up a node and running processes within it. This program contains two required arguments, the control channel name and the command line to be run within the namespace. This command does not require to run with root privileges (CORE, 2015).

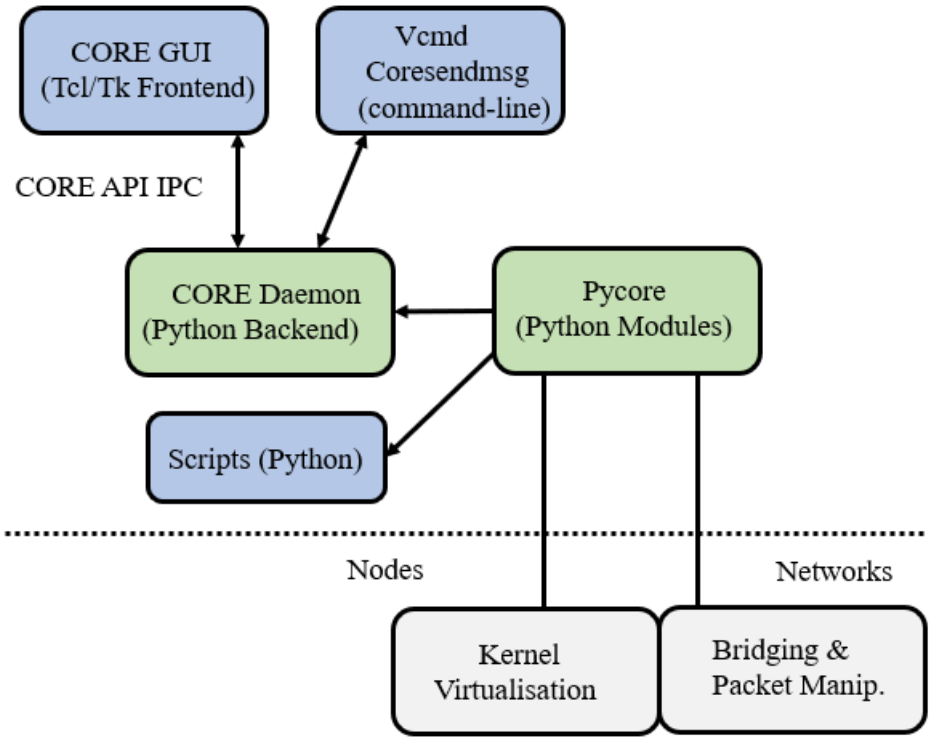

Figure 2.5 CORE Architecture Based on ((CORE, 2015)

The system is modular to allow mixing different components. The virtual network component for example, can be realised with other network simulators and emulators, such as ns-3 and EMANE. Different kinds of kernel virtualisations are supported. As CORE API is socket-based, to allow the possibility of running different components on different physical machines (CORE, 2015).

2.3.2 Working Principle of CORE on Linux

CORE node is a lightweight virtual machine. The CORE framework runs on Linux and FreeBSD systems. The primary platform used for development is Linux. Linux CORE uses Linux network namespace virtualisation to build virtual nodes and ties them together with virtual networks using Linux Ethernet bridging. On the other hand, FreeBSD CORE uses jails with a network stack virtualisation kernel option to create virtual nodes and ties them together with virtual networks using BSD's Netgraph system (CORE, 2015).

Linux network namespace also called as netns, LXC or Linux container, is the initial virtualisation technique used by CORE. LXC has been the part of mainline Linux kernel since 2.6.24. Recent Linux distributions such as Fedora and Ubuntu have isolated namespaces-enabled kernels. Therefore, the kernel does not need to be patched or recompiled. A namespace usually creates using `clone()` system call. Similarly, BSD jails owns namespace which has its individual process environment and private network stack. In CORE, network namespaces share the same filesystem (CORE, 2015).

CORE combines these namespaces with Linux Ethernet bridging to form networks. Link characteristics are applied using Linux Netem queuing disciplines. Ebtables is Ethernet frame filtering on Linux bridges. Wireless networks are emulated by controlling which interfaces can send and receive with ebtables rules (CORE, 2015).

2.3.3 Emulation Workflow

CORE can be used via the GUI or python scripting for creating the network. The GUI is mainly used to draw nodes and network devices on CORE canvas. In order to configure and instantiate nodes and networks, it is possible to write python scripts that imports CORE python module. CORE provides the possibility to observe different phase of an emulated session during the run-time. The session states are stored in CORE-daemon log file. Additionally, CORE can be customized to perform any action at each phase (core, 2019).

Figure 2.6 presents the general emulation workflow of an emulated network. When the session starts, it is in Definition state. This state is used by the GUI to tell the backend to clear any previous state. In this state, GUI connects to the session. Then, the session changes to next state, which is known as, Configuration state. In this state, after pressing the start button in GUI, the information about node type, link and other configuration data are sent to CORE Daemon. Additionally, the network namespace has created in Configuration state. This state also can be achieved by customizing a service. After configuration data has been sent to the backend, the session changes to Instantiation state. In this state, the nodes are ready for operation. The next phase during the session is, Runtime state. In this state, all nodes and networks which have been built, starts running. Then, if user press stop button to terminate the session, it reached to the Data Collection state. At this point, the services and the nodes will be shut down and it is time to collect the log files and other information about the node. Lastly, the session reached to Shutdown state. In this state, all the nodes and networks have been shutdown and destroyed (CORE, 2015).

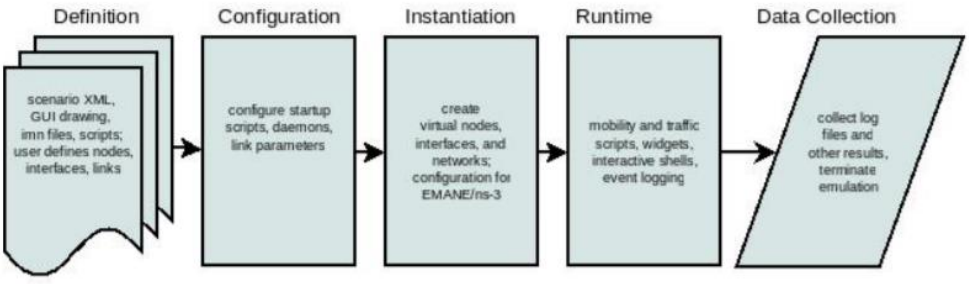

Figure 2.6 Emulation Workflow (core, 2019)

2.3.4 CORE API Message

CORE API creates communication between different components. CORE GUI communicates to CORE Daemon via CORE API. CORE Daemon listens on a TCP socket for CORE API message. CORE GUI connects locally to CORE Daemon and using the API message, it initiates a network topology. The API message is used to transfer different types of message. For example, add, remove or modify nodes and links, configuration data and so on. CORE will also act as an emulation server that listens for a TCP connection from other system. Upon the connection establishment, the other system sends messages to the CORE Daemon that can control the live-running emulation (CORE API, 2019).

Message Format

In a single TCP data packet, one or more than one CORE API control message can appear. Figure 2.7 depicts the format which CORE API control message follows. The TCP header which is presenting in grey colour before "Message Type" indicates the standard TCP header based on TCP specification in RFC 793. The messages shown in black colour is representing the CORE message. Each message starts with four bytes that indicate the message Type, Flags and Message Length. A variable Message Data field is data specific to each Message Type. Each Message Type can implement one or more <Type, Length, Value> tuples or TLVs, for representing fields of data. The allows the flexibility of defining future additional fields. As TCP message contains multiple messages, each message can include several TLV's (CORE API, 2019).

Figure 2.7 CORE API Message Format (CORE API, 2019)

Different Message Types Used in CORE API

CORE API includes different types of messages for each activity. To realise this thesis work, it is necessary to understand the concept of the message type of CORE API. The message types are described below.

- **Node Message:** the Node message is message type 1. The Node message may cause a variety of actions to be performed, depending on the message flags. For example, No flag (00) indicates when neither thee add or delete flags are set, this message is assumed to modify an existing node or node state. Then, Add flag (01) refers to create a node while Delete flag (10) means to give a message to stop or delete the node. Critical flag (100) means the message should be processed if the node's position is not changed but a link calculation is requested. Then, if the flag is Local flag (1000), this means it is a informational message to update a display only. And lastly, another flag is Status request (10000) indicates a response message is requested including the status of the add/delete message. The Node type TLV determines what type of node will be created. The Model Type TLV further determines the configuration of the node depending on node types defined in the GUI (e.g. router, PC). When the Model Type is not present, its value defaults to zero (CORE API, 2019).

- **Link Message:** The Link Message is message type 2. A Link specifies a connection between two nodes, specified by their node numbers.The Link message may cause a variety of actions to be performed, depending on the message flags. When there is no add or delete flag, then No flag (00) message is used to modify an existing link or link state. Add flag (01) for creating link and Delete flag (10) for removing a link are used. Critical flag (100) should not be avoided due to rate limiting. Like Node local flag, Link local flag (1000) message is informative. The Status request (10000) is a response message which is requested consisting the status of the add/delete link message (CORE API, 2019).

- **Execute Message:** The Execute message is type 3. This message is used to execute the specific command on a specified node, and respond with the command output after the command has completed if requested to do so by the message flags. Because commands will take an unknown amount of time to execute, the resulting response may take some time to generate, during which time other API messages may be sent and received. Execute message contains several flag such as, Add flag (01), Delete flag (10) to cancel a pending Execute request message including the given execution number. Then, Critical flag (100) is used when a request needs to be executed immediately. Local flag should be executed on host machine, not within specific node. Moreover, Status request (10000) means an execute response message is requested containing the numeric exit status of the executed process. Text output (100000) flag contains the complete output text of the executed command. Additionaly, Interactive terminal (TTY) flag (1000000) is used to request the command to be executed in order to spawn an interactive user shell on the specified node (CORE API, 2019). This message type is used by CORE observer widget to return the result of a command.

- **Register Message:** The register message is type 4. This message is used by entities to register to recieve certain notifications from CORE or to register services. For example, add flag (01) used for adding a registration, delete flag (10) removes a registration, status request (10000) requests a list of child registrations, Text output (100000) responds for child registrations. Child registrations are defined as follows: if entity 1 registers with entity 2 and requests a list of child registrations, entity 2 will respond with a list of entities that it currently has registeredwith itself (CORE API, 2019).

- **Configuration Message:** The configuration message is type 5. This message is used by an external entity to exchange configuration information with CORE. Using this Configuration Message, CORE can cause the external entity to provide configuration defaults and captions for use in aGUI dialog box (for example when a user clicks configure), and after the dialog box has been filled in, CORE sends the entered data back to the entity. The Configuration Message must contain the Configuration Object TLV. This identifies the object being configured. The Configuration Message must contain the Configuration Type Flags TLV. This TLV contains one or more flags indicating the content of the message.The valid flags are: 0x1 = Request -a response to this message is requested, 0x2 = Update -update the configuration for

this node and 0x3 = Reset -reset the configuration to its default values. The Values TLV contains one or more strings, each representing a value. The strings are separated by a pipe "|" symbol.These values may be default values or data supplied by the user. The Captions TLV contains one or more strings, which may be displayed as captions in a configuration dialog box. The strings are separated by a pipe "|" symbol. Generally, the Data Types, Values, and Captions TLVs will refer to the same number of configuration items. This number may vary depending on the object being configured. One additional caption may be specified in the Captions TLV for display at the bottom of a dialog box (CORE API, 2019).

- **File Message:** The FileMessage is message type 6. This message is used to transfer a file or a file location. Because files may be large, compression may be used or, if the two entities have access to the same file system, this message can provide the path name to a file (CORE API, 2019).

- **Interface Message:** The Interface Message is message type 7. This message will add and remove interfaces from a node. While interface information may be contained in the Link messages, this separate message was defined for clarity.Virtual interfaces typically may be created and destroyed, while physical interfaces are generally either marked up or down (CORE API, 2019).

- **Event Message:** The Event Message is message type 8. This message signals an event between entities or schedules events in a session event queue. For example, when the user presses the button "Start" from CORE GUI, it shows Event (type = "configuration state") to configure the nodes and links (CORE API, 2019).

- **Session Message:** The Session Message is message type 9. This message is used to exchange session information between entities. It contains add flag to add a new session or connect to existing session, delete flag to remove the session and shut it down and Status response flag to request a list of session. This status response flag can be used with add flag to request a list of session objects depending on connection (CORE API, 2019).

- **Exception Message:** The Exception Message is message type 10. This message is used to notify components of warnings, errors, or other exceptions. No flags are defined for the Exception Message (CORE API, 2019).

CORE API Message Flow

In the current CORE Implementation, CORE is using an API to create communication between CORE GUI and CORE Daemon. It is a custom, asynchronous and socket-based API. CORE Daemon listens on a TCP socket for CORE API messages from other components. Using this API, CORE GUI connects with CORE daemon to instantiate any network topologies. When the connection is established, CORE GUI and Daemon transmits messages to each other in a live-running emulation. The message pattern follows request-response type.

Figure 2.8 presents the message flow of current CORE emulator implementation. The session message is used to exchange session information between the components of CORE. The session message consists of three types messages. They are – Add flag (01) which denotes adding new session or connecting to the existing session if it is available, Delete flag (10) indicates to remove a session and shut it down and Status Response Flag request a list of sessions.

Then the Event message triggers an event between the components in a running session queue. When the Start button is pressed for the emulation from CORE GUI, the first Event message "Definition state" appears and then it reached to "Configuration state" to request for the configuration data.

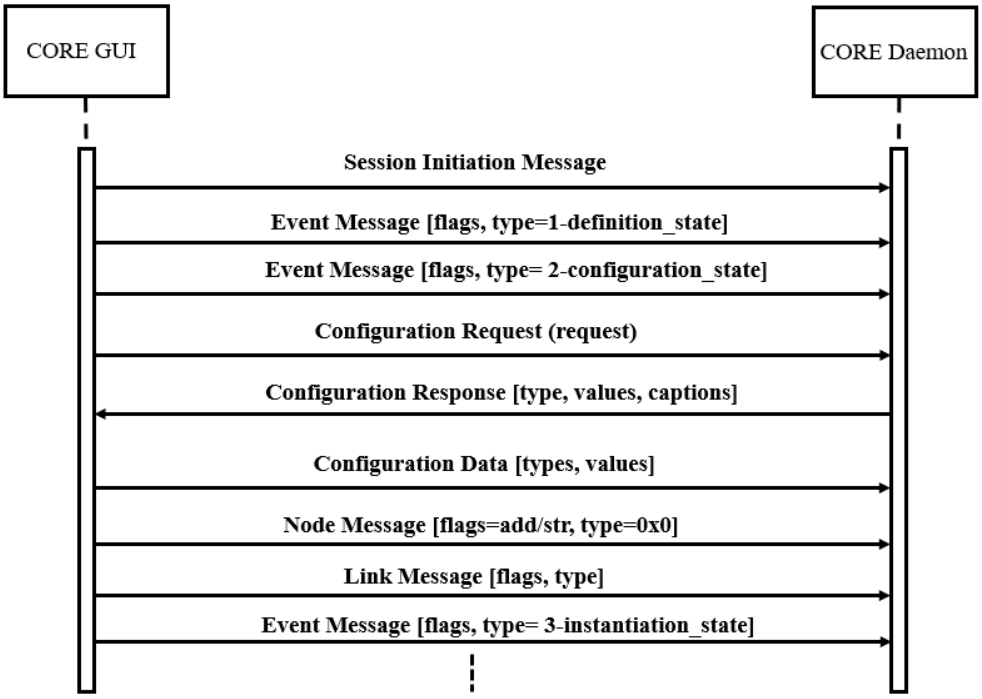

Figure 2.8 Message Flow of Current CORE API Communication

In Configuration state, the configuration message is used by an external entity to exchange configuration data with CORE. Here, external entity mainly provides configuration defaults and caption which appears on CORE GUI dialog box when a user clicks on the configuration. After entering data in the dialog box, CORE sends back the data to the external entity. In this way, CORE API communicates with different components. The configuration message includes the configuration object TLV to identify the object being configured or not. The configuration message also contains Configuration Type Flag TLV. The flag for request is 0x1 which indicates request and flag for reset the configuration to its default value is 0x3. Then the values TLV presents one or more strings containing values. These values can be default values or user defined. The Captions TLV refers to the same number of configuration items as values TLV.

Then, Node message appears in GUI to perform various actions such as Add or remove nodes. The Node Type TLV defines what type of node will be created. Then, the next message is Link message to determine a connection between two nodes, specified by their node numbers. This message is sent to configure the link between the nodes. The Link message also performs variety of tasks depending on the message flags.

After configuring the nodes and link, the event message “Instantiation state” triggers. It signals to CORE Daemon that the node/link configuration is sent so, the emulation can be instantiated. In this state, the nodes and links perform some action based on user requirements. In this way, CORE API exchanges information between CORE GUI and Daemon.

2.3.5 Toolbar

In CORE, toolbar is a row of button that runs vertically along the left side of the CORE GUI window. It changes based on the mode of operation. The CORE GUI has two modes of operations such as Edit and Execute

Editing Toolbar

Editing toolbar is the default mode of CORE. This vertical toolbar exists on the left side of the CORE window. There is a brief description for each toolbar item below. Most of the tools are grouped into related sub-menus which appear when user click on the group icon (CORE, 2015).

-  Selection Tool - default tool for selecting, moveing, configuring nodes
-  Start Button - starts Execute mode, instantiates the emulation
-  Link - the Link Tool allows network links to be drawn between two nodes by clicking and dragging the mouse (CORE, 2015).

Network-layer Virtual Nodes

-  Router - runs Quagga OSPFv2 and OSPFv3 routing to forward packets
-  Host - emulated server machine having a default route, runs SSH server
-  PC - basic emulated machine having a default route, runs no processes by default
- 

MDR - runs Quagga OSPFv3 MDR routing for MANET-optimized routing
-  PRouter - physical router represents a real testbed machine, physical
-  Edit - edit node types button invokes the CORE Node Types dialog. New types of nodes may be created having different icons and names. The default services that are started with each node type can be changed here (CORE, 2015).

Link-layer Nodes

-  Hub - the Ethernet hub forwards incoming packets to every connected node
-  Switch - the Ethernet switch intelligently forwards incoming packets to attached hosts using an Ethernet address hash table
-  Wireless LAN - when routers are connected to this WLAN node, they join a wireless network and an antenna is drawn instead of a connecting line; the WLAN node typically controls connectivity between attached wireless nodes based on the distance between them
-  RJ45 - with the RJ45 Physical Interface Tool, emulated nodes can be linked to real physical interfaces on the Linux or FreeBSD machine; using this tool, real networks and devices can be physically connected to the live-running emulation (RJ45 Tool)
-  Tunnel - the Tunnel Tool allows connecting together more than one CORE emulation using GRE tunnels (Tunnel Tool) (CORE, 2015).

Annotation Tools

-  Marker - for drawing marks on the canvas
-  Oval - for drawing circles on the canvas that appear in the background
-  Rectangle - for drawing rectangles on the canvas that appear in the background

-  Text - for placing text captions on the canvas (CORE, 2015).

Execution Toolbar

When the Start button is pressed, CORE switches to Execute mode, and the Edit toolbar on the left of the CORE window is replaces with the Execution toolbar. The items of this toolbar are presented below.

-  Selection Tool - in Execute mode, the Selection Tool can be used for moving nodes around the canvas, and double-clicking on a node will open a shell window for that node; right-clicking on a node invokes a pop-up menu of run-time options for that node
-  Stop button - stops Execute mode, terminates the emulation, returns CORE to edit mode
-  Observer Widgets Tool - clicking on this magnifying glass icon invokes a menu for easily selecting an Observer Widget. The icon has a darker gray background when an Observer Widget is active, during which time moving the mouse over a node will pop up an information display for that node (Observer Widgets).
-  Plot Tool - with this tool enabled, clicking on any link will activate the Throughput Widget and draw a small, scrolling throughput plot on the canvas. The plot shows the real-time kbps traffic for that link. The plots may be dragged around the canvas; right-click on a plot to remove it
-  Marker - for drawing freehand lines on the canvas, useful during demonstrations; markings are not saved
-  Two-node Tool - click to choose a starting and ending node and run a one-time traceroute between those nodes or a continuous ping -R between nodes. The output is displayed in real time in a results box, while the IP addresses are parsed, and the complete network path is highlighted on the CORE display.
-  Run Tool - this tool allows easily running a command on all or a subset of all nodes. A list box allows selecting any of the nodes. A text entry box allows entering any command. The command should return immediately, otherwise, the display will block awaiting a response. The ping command, for example, with no parameters, is not a good idea. The result of each command is displayed in a results box. The first occurrence of the special text “NODE” will be replaced with the node name. The command will not be attempted to run on nodes that are not routers, PCs, or hosts, even if they are selected (CORE, 2015).

2.4 Python Modules

This section contains some python modules which have been used in this thesis work. Python provides many modules that simplify coding a large application. The functions of the modules can be reused many times throughout a program that helps to reduce replication. The contents of the python modules are accessible via the `import` statement. The modules which have been used in this thesis task, are described below.

OS Module

Python’s `os` module offers methods to interact with the operating system. It is a utility module of python which gives a portable way of using operating system dependant functionality (GeeksforGeeks, 2019). It performs many operating system tasks automatically such as create a file or folder, fetch the contents of a folder.This `os` module has numerous usages. Changing of a directory can be done by `os.chdir()` method or to fetch the contents of a directory is possible by `os.listdir()` method (OS, 2019).

Subprocess Module

The `subprocess` module of python allows to spawn new processes, connect to their input/output/error pipes, and acquire their return code (subprocess, 2019). The `subprocess` module provides a consistent interface to creating and working with additional processes. It offers a higher-level interface than some of the other available modules. Several older modules and functions such as `os.system()`, `os.popen()`, `os.spawn()`, `commands.*()` are intended to be replaced by this module (Hellmann, D., 2019).

LXC Python Bindings

LXC includes stable C API and Python bindings for both version python 2.x and python 3.x versions. Using this API, it is quite easy to create, start, stop and destroy the LXC container. It also provides the functions to modify the configuration of the containers and use the system resources like CPU, RAM (lxc/python3-lxc, 2019). To use in Linux machine, it can be installed as the following command below (lxc, 2019).

```
$ sudo pip install lxc-python2
```

2.5 Tcl/Tk

Tcl refers to Tool Command Language which is very powerful and dynamic programming language. It is suitable for a wide range of uses such as networking, web applications, desktop applications, testing and so on. On the contrary, Tk is a cross-platform widget toolkit for designing graphical user interface (GUI) not only for Tcl but also for many other dynamic languages. So, this two combinedly enable GUI in Tcl. It has the capability of interacting with other programs and works as an embedded interpreter. Many full-fledged applications have been written in Tcl/Tk (Tcl Developer Xchange, 2019). CORE network emulator used Tcl/Tk for the development of CORE GUI.

2.6 Distrobuilder

Distrobuilder is a tool for building system container images for LXC and LXD (Xenitellis, S., 2018a). In this thesis work, distrobuilder is used to create image for LXC container.

To create an image, first, an example image configuration file should be copied from distrobuilder repository (lxc/distrobuilder, 2019) to a user-defined folder. The command for creating template image for LXC is `build-lxc`. This command compiles the YAML configuration file and produce the system container image. Apart from the YAML configuration file, there are other two files respectively rootfs file and a metadata file will also be created (Xenitellis, S., 2018a).

Figure 2.9 shows the container image creation using distrobuilder. From the Distrobuilder Github repository the template images can be either cloned or downloaded to the local directory. Then the image should be copied to a new directory. At this point, it is possible to customize the image YAML configuration file according to the user requirements. Based on the created image templates, multiple containers can be created.

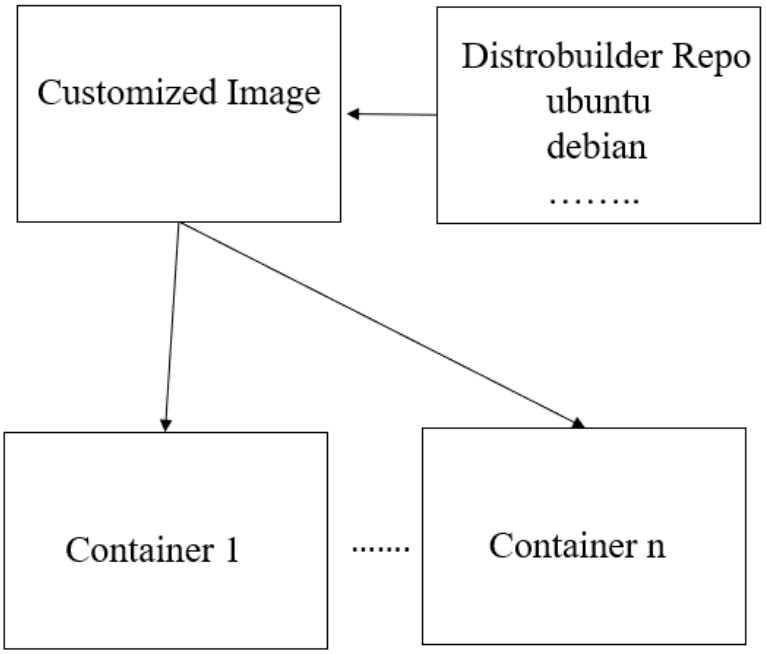

Figure 2.9 Container Image creation Using Distrobuilder

The configuration file of the container image contains three sections. They are explained below in brief Figure 2.10.

- **Image**, it represents the information about the image. User can put anything for the description and distribution name. Moreover, the release version should exist (Xenitellis, S., 2018b).

- **source**, which describes where to get the image, ISO or packages of the distribution. The downloader is a plugin in distrobuilder that knows how to get the appropriate files, as long as it knows the URL and the release version. The url is the URL prefix of the location with the files. keys and keyserver are used to verify digitally the authenticity of the files (Xenitellis, S., 2018b).

- **packages**, which indicates the plugin that knows how to deal with the specific package manager of the distribution. In general, user can also be able to indicate here which additional packages to install, which to remove and which to update (Xenitellis, S., 2018b).

```
image:
  description: My Alpine Linux
  distribution: minimalalpine
  release: 3.8.1

source:
  downloader: alpinelinux-http
  url: http://dl-cdn.alpinelinux.org/alpine/
  keys:
    - 0482D84022F52DF1C4E7CD43293ACD0907D9495A
  keyserver: keyserver.ubuntu.com

packages:
  manager: apk
```

Figure 2.10 A Sample YAML Configuration File ((Xenitellis, S., 2018b).)

The downloader and url go hand in hand. The URL is the prefix for the repository that the downloader will use to get the necessary files (Xenitellis, S., 2018b).

The keys are necessary to verify the authenticity of the files. The keyserver is used to download the actual public keys of the IDs that were specified in the keys. You could very well not specify a keyserver, and distrobuilder would request the keys from the root PGP servers. However, those servers are often overloaded, and the attempt can easily fail. It happened to me several times so that I explicitly use now the Ubuntu keyserver (Xenitellis, S., 2018b).

3 Requirements Analysis

The main goal of this thesis work is to develop and integrate an extension to the CORE Network Emulator to support LXC containers as a node type and analyses the communication between CORE GUI and CORE Daemon to identify the possible interface for the extension.

3.1 General Objectives

In this work, an LXC-based node type will be developed inside the CORE Network Emulator to enable the usage of LXC container as a node in CORE emulator. Moreover, LXC container related configurable aspects such as template image selection, limiting CPU and RAM usage will be developed. CORE GUI component will offer the interface to specify the configuration parameters for LXC node. Then these parameters will be retrieved from a configuration manager which will be developed into the CORE Daemon. CORE GUI will fetch the data from CORE Daemon using CORE API. CORE uses this API to communicate internally between its components. So, the communication between CORE GUI and CORE Daemon for LXC node type also needs be developed.

To implement the task, Tcl/Tk for development in CORE GUI, python for CORE Daemon and CORE emulator network tool will be used in the virtual machine. Finally, the evaluation of the implemented approach will be done based on the following parameters.

- Integrate LXC container as a node type into the CORE network emulator
- Develop the required communication between CORE GUI and CORE Daemon using CORE API
- Implementation of LXC related configurable features
    - Template image selection for LXC node
    - Limiting CPU and RAM usage
    - Mount directories from the host machine to LXC containers
    - Implementation of a start-up command
- Performance analysis

3.2 Clarifying the Requirements

In this section, there will be a detail discussion of the thesis topic which will be followed to reach the target state. The implemented approach should support LXC containers in the CORE network emulator. Therefore, it will be useful to launch some set of applications or services and user can be able to run the applications in a self-contained isolated environment. First of all, there will be some investigation regarding the limitations of Linux Network Namespace which currently CORE is using. After that, there will be detailed discussion about integration of LXC to CORE and investigation due to communication between CORE GUI and CORE Daemon.

3.2.1 Limitation in Network Namespace

CORE consists of two major components – CORE GUI and CORE Daemon. CORE uses Linux network namespace virtualisation technique to create virtual machine that emulate each node during a network simulation. Network namespace offers support for the implementation of the lightweight virtualisation that provide isolation of the networking resources such as devices, routing tables and network addresses. But CORE nodes are not full-fledged virtual machines. CORE network emulator uses namespace to create only minimum necessary resources for each node. In order to ensure each virtual node in the network simulation lightweight, CORE uses own set of tools for configuring and instantiating network namespace containers. However, CORE does not provide the possibility to limit and define hardware aspects for example CPU, RAM, disk space. On the contrary, LXC takes full advantage of namespace and cgroups to achieve the process isolation and resource control. Figure 3.1 presents the

conceptual design of CORE node's network namespace creation. Here, n1, n2 and n3 are three default CORE nodes which are connected to each other via Linux Ethernet Bridge. Each node has individual network namespace and the virtual ethernet (veth) devices are used to create link between the network namespaces.

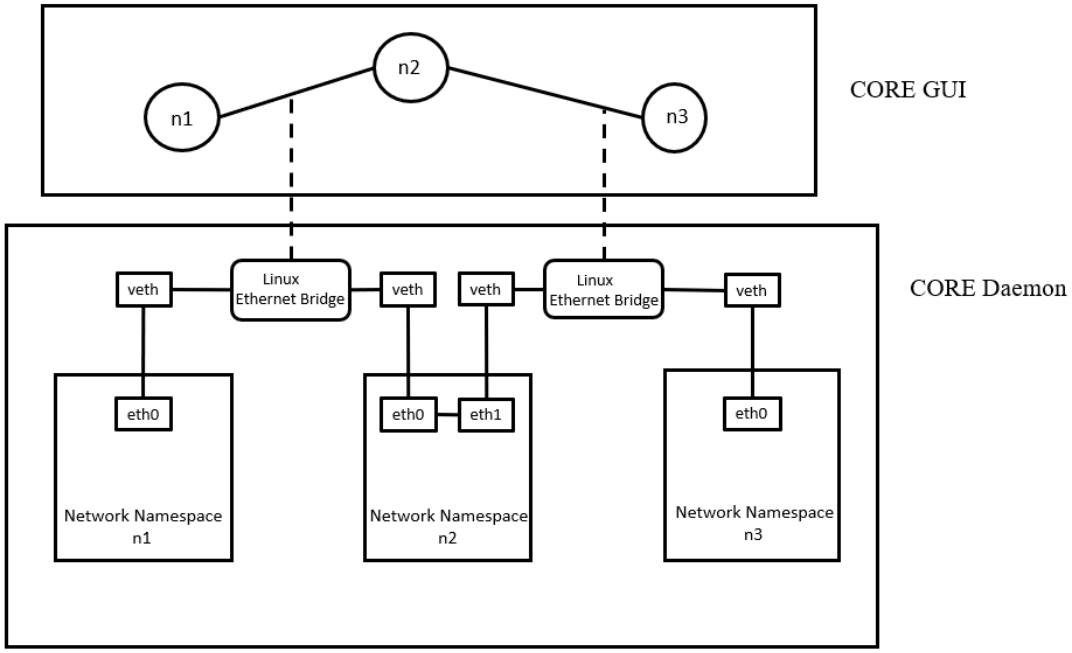

Figure 3.1 Conceptual Design of CORE Node's Network Namespace Creation

3.2.2 Development of an Extension to the CORE

In this thesis work, the developed extension shall support LXC container as node type. The containers should be created, started, stopped and destroyed inside CORE emulator. For that, LXC container should be implemented in such a way so that it can extend the existing CORE node functionality. In this way, inside CORE emulator, the LXC container can be able to communicate with another LXC node as well as to communicate with other network-namespace based CORE nodes. To enhance the usage of the containers, there shall be some configuration parameters such as specifying CPU usage, defining to limit the RAM usage, attach a directory from host machine to LXC container, execute a start-up command during the lifecycle of containers. Based on these parameters, the containers will be configured.

3.2.3 Develop Communication between CORE GUI and CORE Daemon

In this task, there should be a communication interface between CORE GUI and CORE Daemon. In the current CORE implementation, CORE Daemon and CORE GUI is using CORE API for communication. Figure 3.2 shows the conceptual design of the message communication between CORE GUI and CORE Daemon. At first the node will be added to the CORE canvas. To configure the node, a button will be created and click apply to enable the configuration for the node. This should send a configuration request and should receive configurable options for configuration as well. User will give input for configuration and enable the configuration. If the session starts, the LXC node type should be recognised and it should receive the user-defined configuration. After that, the nodes and links will be created according to this conceptual design.

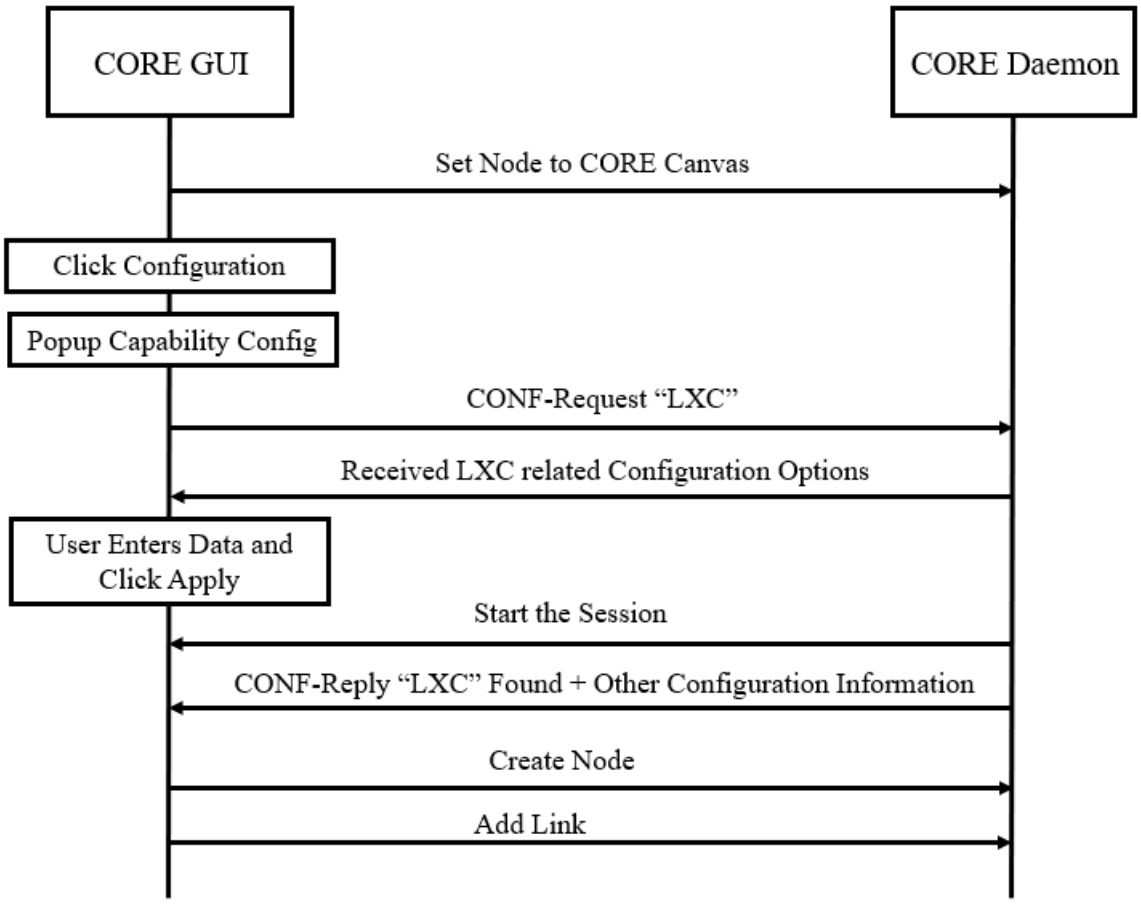

Figure 3.2 Conceptual Design of CORE API Message Communication between CORE GUI and CORE Daemon

3.3 Time frames

Milestones:

08.02.2019 to 20.02.2019:	Analysis of the requirements
21.02.2019:	Submission of the requirements analysis
21.02.2019 to 21.03.2019:	Literature review and learning the theoretical background
21.02.2019 to 27.05.2019:	Writing up the thesis and the documentation
21.03.2019 to 10.04.2019:	Elaborating of the concept and planning of the prototype
21.03.2019 to 30.06.2019:	Implementation of the prototype
27.06.2019:	Submission of the Thesis draft to supervisor
27.06.2019 to 29.07.2019:	Revise the thesis based on the proposals from the supervisor
29.07.2019:	Submitting the Thesis to the examination office

3.4 Target State

For the implementation of the prototype, the final state shown in Figure 3.3 is expected. The figure shows that an LXC node type is linked with a network namespace-based node type through Linux bridge network. When the "Start" button is pressed to start a session in CORE emulator, LXC container should be created and started along with the network namespace CORE node. In case of implementation, the first task is to prepare the environment and install CORE network emulator. Then, observe the current CORE network implementation and identify the phase where network namespace is creating for the node. In addition, the CORE bridge network implementation should be observed for LXC container implementation to CORE emulator as this is an important aspect of this work. Then the next task is, LXC containers will be implemented as a node type into the CORE emulator and thus it will be able to provide the functionality of LXC containers such as creating the container, starting, stopping and finally destroying everything inside CORE. It is required to implement an interface to develop a communication between CORE GUI and CORE Daemon. So, in this case, CORE API message will be used to create the communication between these components.

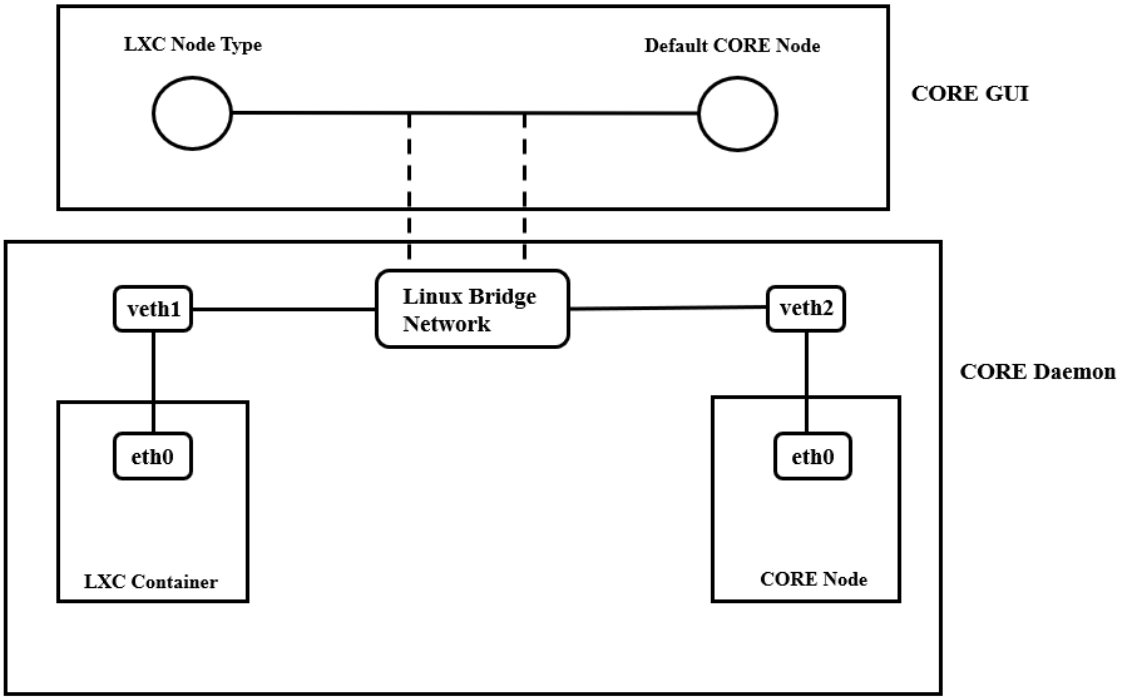

Figure 3.3 Target State

Moreover, a separate configuration manager will be developed to define the configuration parameters for the LXC container. Based on this configuration options, the developed LXC node type will be able to select the image template to create the container. The template images will be stored in a predefined directory and user can be able to fetch the images from that directory to CORE GUI. Then the further implementation of LXC node type will specify the usage of the resources such as CPU and RAM usage for a container. Additionally, LXC container allows attaching directories from the host operating system to the container and it will be possible to traverse the running directory of the container. And, lastly there will be an implementation for startup command to create any file so that once the container is running, the file will be already existing in the specific container.

3.5 Necessary Tools

Required tools for implemnetation are given below:

- Virtual Machine – OS Ubuntu 18.04 LTS
- CORE Emulator Tool – Release 5.2
- Scrpit: Python for CORE Daemon and TCL-TK for CORE GUI implementation

3.6 Use Cases for the Prototype

To demonstrate and evaluate the functionality of the approach for LXC-based node type into the CORE network emulator, the following points will be implemented.

Extension to the CORE Emulator Network

- Implementation of a configuration manager for LXC node for managing the configuration during a CORE session
- Implementation of LXC Lifecycle Management such as create, start, stop and destroy container inside CORE Daemon
- Select container image dynamically
- Specify and limit the usage of CPU and RAM in LXC container
- Mount directory from host system to LXC container and explore the filesystem of a running container
- Implementation and execution of a start-up command inside LXC container
- Enable observer widget on LXC node and display the result in CORE GUI
- Observe the communication between CORE GUI and CORE Daemon via CORE API

4 Realisation

This chapter will describe the implementation and testing of the thesis work in details. The following are the steps of the implementation of the thesis work.

- Configuration on Ubuntu host
- Investigation of current CORE API communication
- Integration of a selectable LXC node type into CORE GUI
- User-definable configuration option for an LXC node type
- Integration of a lifecycle management for LXC node type into CORE Daemon
- Evaluation of the implemented approach

4.1 Configuration on Ubuntu Host

In this implementation LXC container will be integrated as a node type into CORE, so, CORE network emulator needs to be installed on Ubuntu host machine. Before installing CORE, following configuration has been made on the host.

Prerequisites by Ubuntu version 18.04

```
$ sudo apt-get update
```

```
$ sudo apt-get upgrade
```

Uninstalling unnecessary software's

```
$ sudo apt -y remove --purge avahi-daemon
```

```
$ sudo apt -y remove --purge ufw
```

4.1.1 Installing CORE Network Emulator on Ubuntu Host

In this thesis work, CORE version 5.2 has been used and the following steps can be followed to install CORE. To build the CORE system, it is mandatory to install some prerequisite packages need to be installed in the host machine.

Prerequisite packages for Ubuntu 18.04 are given below

```
$ sudo apt-get update
```

```
$ sudo apt-get install git
```

```
$ sudo apt-get install bash bridge-utils ebtables \ iproute libev-dev python
tcl8.5 tk 8.5 libtk-img \ autoconf automake gcc libev-dev make python-dev \
libreadline-dev pkg-config \ imagemagick help2man
```

Then quagga need to be installed for routing:

```
$ sudo apt-get install quagga
```

Configuring Wireshark:

```
$ sudo rm /usr/bin/wireshark
```

```
$ sudo ln -s /usr/bin/wireshark-gtk /usr/bin/wireshark
```

```
$ sudo dpkg-reconfigure wireshark-common
```

```
$ sudo adduser $USER wireshark
```

Hardware Requirements for CORE

A general recommendation would be

- 2.0GHz or better x86 processor, the more processor cores the better.
- 2 GB or more of RAM.
- about 3 MB of free disk space.
- X11 for the GUI, or remote X11 over SSH.

Downloading CORE

To download a specific version of CORE network emulator, by clicking “Branch” tab, CORE v5.2 has been selected through the open source Github repository which is shown in the Figure 4.1

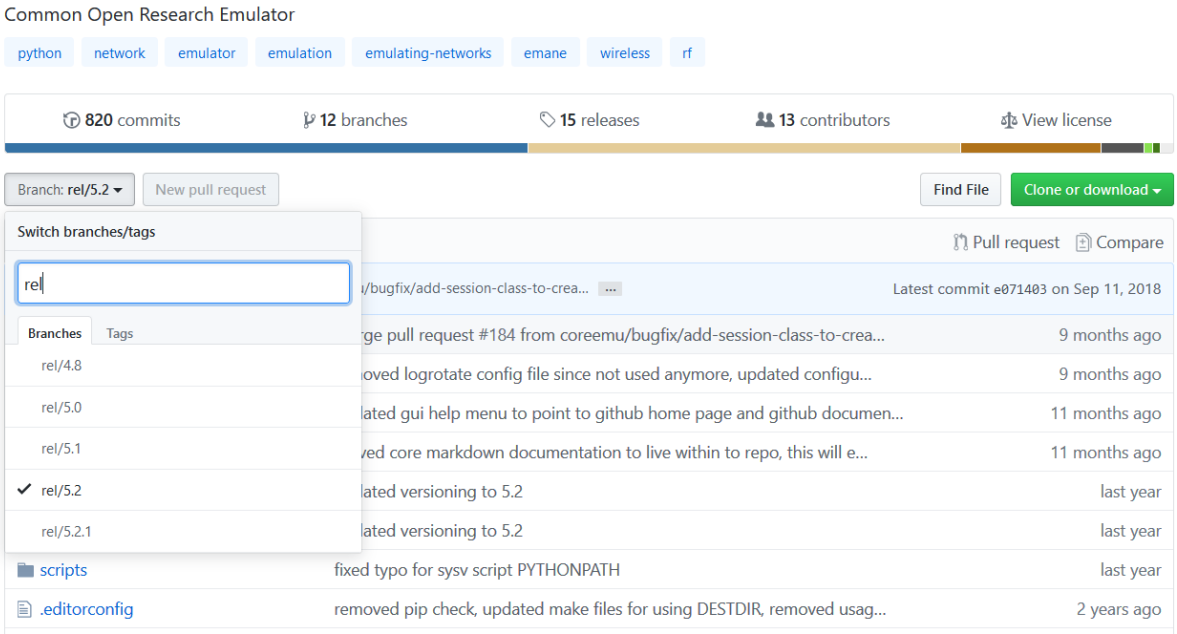

Figure 4.1 Selecting CORE v5.2 from Github Repository

Unzip and Install CORE:

```
$ cd Downloads
```

```
$ unzip core-rel-5.2.zip
```

After unzipping, CORE has been installed by using below commands

```
$ cd core-rel-5.2
```

```
$ ./bootstrap.sh
```

```
$ . / configure
```

```
$ make
```

```
$ sudo make install
```

Setting up Quagga:

```
$ sudo touch /etc/quagga/zebra.conf
$ sudo touch /etc/quagga/ospfd.conf
$ sudo touch /etc/quagga/ospf6d.conf
$ sudo touch /etc/quagga/ripd.conf
$ sudo touch /etc/quagga/ripngd.conf
$ sudo touch /etc/quagga/isisd.conf
$ sudo touch /etc/quagga/pimd.conf
$ sudo touch /etc/quagga/vtysh.conf
$ sudo chown quagga.quaggavty /etc/quagga/*.conf
$ sudo chown quagga.quaggavty /etc/quagga/*.conf
$ sudo chmod 666 /etc/quagga/*.conf
```

Configuring Quagga daemons:

To configure Quagga daemons `/etc/quagga/daemons` has been edited and started `zebra` and `ospfd` daemons by using below command which is shown in Figure 4.2

```
$ sudo nano /etc/quagga/daemons
```

Figure 4.2 Configuration File of Quagga Daemon

Setting up environment variables to avoid the Quagga vtysh END problem in Ubuntu Linux:

```
$ sudo bash -c 'echo "export VTYSH_PAGER=more" >> /etc/bash.bashrc'
```

```
$ sudo bash -c 'echo "VTYSH_PAGER=more" >> /etc/environment'
```

Start CORE with the following command from terminal:

```
$ sudo /etc/init.d/core-daemon restart
```

```
$ core-gui
```

This command will connect to the CORE Daemon and open a CORE Canvas where all the network emulation will take place (Figure 4.3)

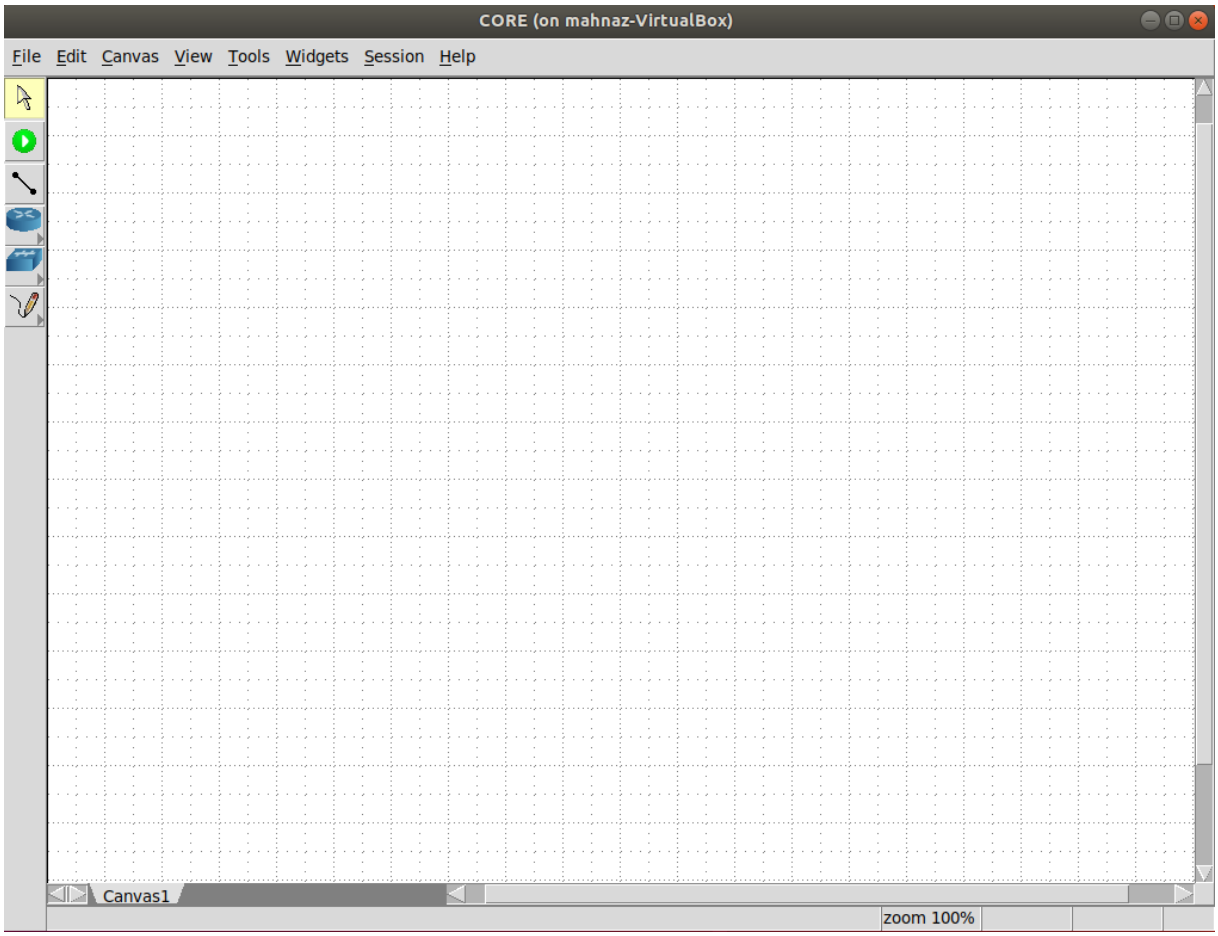

Figure 4.3 CORE-GUI Canvas

There is a separate Git repository for CORE emulator network which is specially dedicated to develop LXC based node type into the CORE emulator. This repository can be found under this URL link: http://192.168.1.100/core-fgftk/core.

4.1.2 Installing Required Python Modules

For the implementation part, lxc-python bindings need to be installed:

```
$ sudo apt-get install python-pip
```

```
$ sudo pip install lxc-python2
```

```
$ sudo apt-get install lxc-dev
```

4.1.3 Installing Distrobuilder for LXC Template Image

Distrobuilder is a tool to build system container image for LXC and LXD. This tool is easy to install and useful for creating LXC template image. Before installing distrobuilder, it is important to install Go programming language first and some dependencies. The commands to do so is given below

```
$ sudo apt-get update
```

```
$ sudo apt install -y golang-go debootstrap rsync gpg squashfs-tools
```

Before proceeding further, Go language has to be installed properly because the distrobuilder tool is written in Golang. To install Go in ubuntu host machine, the following command is,

```
$ wget https://dl.google.com/go/go1.12.3.linux-amd64.tar.gz
```

Download the latest version from the website and extract it to install in the desired location in the host system. Here, the downloaded archive has kept in `/home/mahnaz` directory.

To extract the file, the command is,

```
$ tar xvf go1.12.3.linux-amd64.tar.gz
```

Then the next step is to setup the Go path. To edit the file and setup Go environment, the command is given below.

```
$ sudo nano ~/.profile
```

At the end of the file, add following command:

```
export GOROOT=/usr/local/go
```

```
export GOPATH=$HOME/go
```

```
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

Save and exit from the file. Next, update the profile by this command,

```
$ source ~/.profile
```

To verify the configuration is successful or not, check by running,

```
$ go version
```

Now, Golang is installed, so it is time to move to the next step. The next step is to download the source code from Distrobuilder repository and place it to this `$HOME/go/src/github.com/lxc/distrobuilder` directory. The command to do this is,

```
$ go get -d -v github.com/lxc/distrouilder
```

After this, enter to the directory and run `make` to compile the source code. This will generate the executable program distrobuilder and can be found at `$HOME/go/bin/distrobuilder`. The commands are given below.

```
$ cd $HOME/go/src/github.com/lxc/distrobuilder
```

```
$ make
```

```
$ cd
```

Now, the installation part is done, so it is possible to create a container image. To do this, first create a directory for storing image and enter that directory. Here, the commands are –

```
$ mkdir -p $HOME/ContainerImages/ubuntu/
```

```
$ cd $HOME/ContainerImages/ubuntu/
```

Now, copy any example yaml configuration file from distrobuilder to this directory for container image. In this task, an Ubuntu container image has been created.

```
$ cp $HOME/go/src/github.com/lxc/distrobuilder/doc/example/ubuntu ubuntu.yaml
```

It is also possible to customize an image configuration file according to the requirements. So, here the Linux distribution for the Ubuntu Linux container image has been defined as "bionic" and in Figure 4.4 , it is outlined.

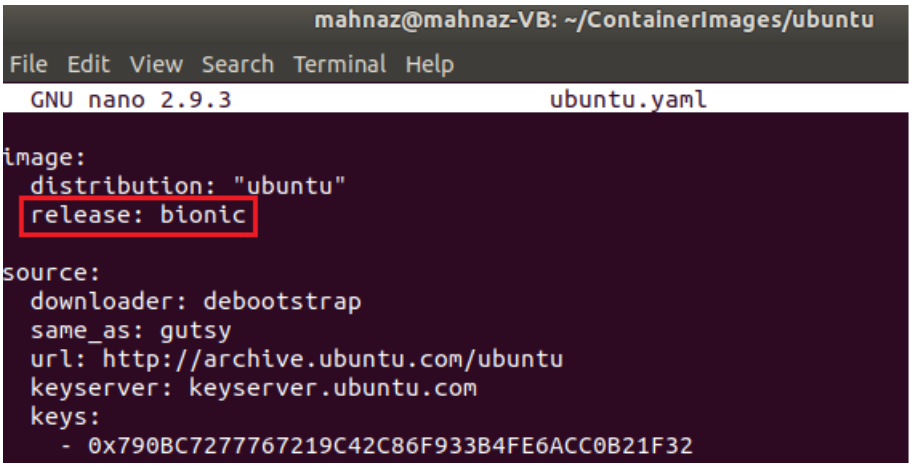

Figure 4.4 Modified Release of Ubuntu Linux Container Image

Some required packages for this thesis work such as `nano`, `systemd`, `net-tools`, `iputils-ping`, `stress`, `htop` have been added under the set of packages of the `ubuntu.yaml` configuration file like the Figure 4.5.

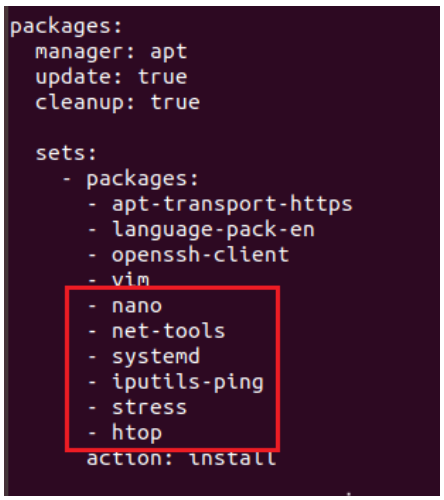

Figure 4.5 Added Packages to Ubuntu Container Image Configuraion File

Finally, to create the container image for LXC run distrobuilder by executing `build-lxc` command. This command requires `sudo` as the rootfs of the image needs to set the ownership and permission of the files. The following command is given below.

```
$ sudo $HOME/go/bin/distrobuilder build-lxc ubuntu.yaml
```

It will take some time to compile and after the successful compilation it will look like this (Figure 4.6).

```
mahnaz@mahnaz-VB:~/ContainerImages/ubuntu$ ls -l
total 144432
-rw-r--r-- 1 root root       784 Jun 13 12:14 meta.tar.xz
-rw-r--r-- 1 root root 147882816 Jun 13 12:14 rootfs.tar.xz
-rw-r--r-- 1 root root      4613 Jun 25 21:04 ubuntu.yaml
mahnaz@mahnaz-VB:~/ContainerImages/ubuntu$
```

Figure 4.6 Set of Files in Container Image

Now, using these files, it is very easy to create container and start the container.

Docker can be installed to LXC template image using distrobuilder. This will enable the possibility to work with docker inside LXC container. Additionally, a docker image also needs to be downloaded before the LXC container creates and starts.

This requires several steps to run docker in LXC. So, the first step is, to install some packages for docker. The following Figure 4.7 shows the marked pre-requisite packages for docker installation in distrobuilder.

```
packages:
  manager: apt
  update: true
  cleanup: true

  sets:
    - packages:
      - apt-transport-https
      - language-pack-en
      - openssh-client
      - vim
      - nano
      - net-tools
      - curl
      - tar
      - jq
      - golang-go
      - wget
      - systemd
      - iputils-ping
      action: install
```

Figure 4.7 Required Packages for Installing Docker Using Distrobuilder

Then, to install docker the following commands should be added to the post-package section of the configuration file.

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
```

```
$ sh get-docker.sh
```

Then, to download the docker-image, add the following lines in the post-package section of the yaml configuration file.

```
$ wget https://raw.githubusercontent.com/moby/moby/master/contrib/download-frozen-image-v2.sh
```

```
$ bash download-frozen-image-v2.sh ubuntu ubuntu_latest
```

```
$ tar -C 'ubuntu' -cf 'ubuntu.tar' .
```

Figure 4.8 presents the configuration file how it should be look like.

```
- trigger: post-packages
  action: |-
    #!/bin/sh
    set -eux

    # Make sure the locale is built and functional
    locale-gen en_US.UTF-8
    update-locale LANG=en_US.UTF-8

    curl -fsSL https://get.docker.com -o get-docker.sh
    sh get-docker.sh
    wget https://raw.githubusercontent.com/moby/moby/master/contrib/download-frozen-image-v2.sh
    bash download-frozen-image-v2.sh ubuntu ubuntu:latest
    tar -C 'ubuntu' -cf 'ubuntu.tar' .

    # Cleanup underlying /run
    mount -o bind / /mnt
    rm -rf /mnt/run/*
    umount /mnt
```

Figure 4.8 Post-packages for Docker and Docker Image

After that, the next step is to enable nested virtualisation in LXC container. To to that, add the following content in `/etc/lxc/default.conf` file which is presenting in Figure 4.9.

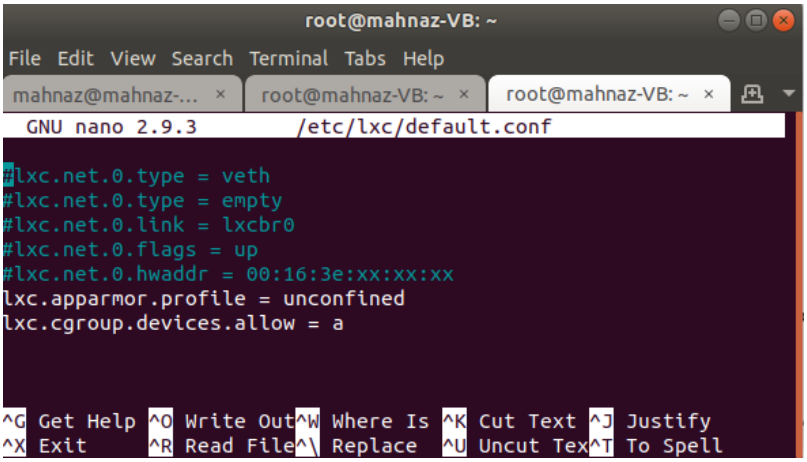

Figure 4.9 LXC Default Configuration File

Then, compile the LXC distrobuilder template image file by using this command

```
$ sudo $HOME/go/bin/distrobuilder/ build-lxc ubuntu.yaml
```

Once the file is compiled, now start the LXC container in CORE. After starting the container, the image which has been downloaded during the creation of the LXC template, required to be imported into docker via the following command

```
$ sudo docker load -i ubuntu.tar
```

Now, finally, to check docker is running or not, execute the following command

```
$ docker run -it ubuntu
```

Here, Figure 4.10 shows the downloaded docker image and also the docker container created inside LXC container.

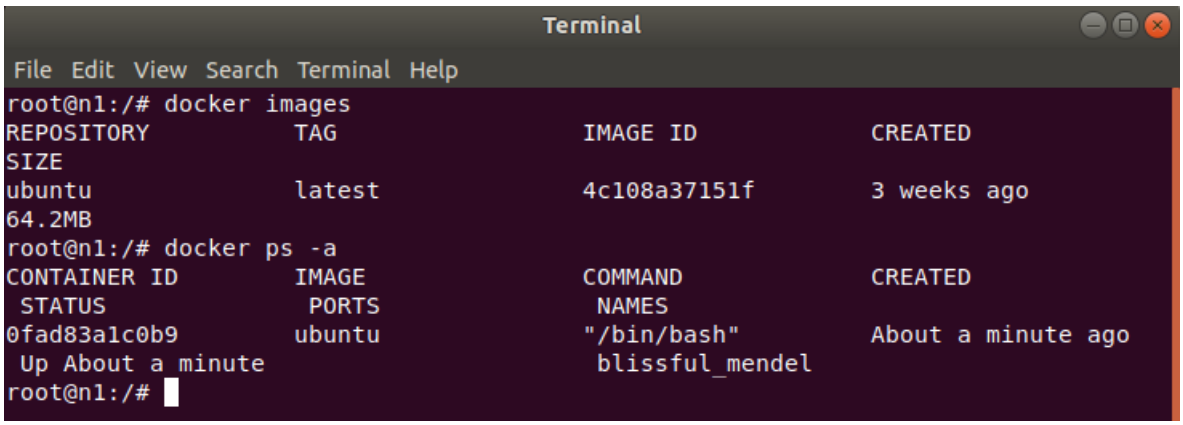

Figure 4.10 Docker Container Status

4.2 Investigation of Current CORE Implementation

This chapter will investigate the current CORE network implementation and will define the concept behind the implementation for LXC node integration into CORE network emulator based on this implementation.

As discussed before in the Requirement Analysis part that LXC container will be integrated to the CORE emulator as a node type, so it is necessary to explore more in depth to figure out the possible solution and the outcome of it. Figure 4.11 shows a simple network topology for the analysis. Here, two CORE nodes respectively, n1 (pc) and n2 (router) is connected to each other via Ethernet bridge network.

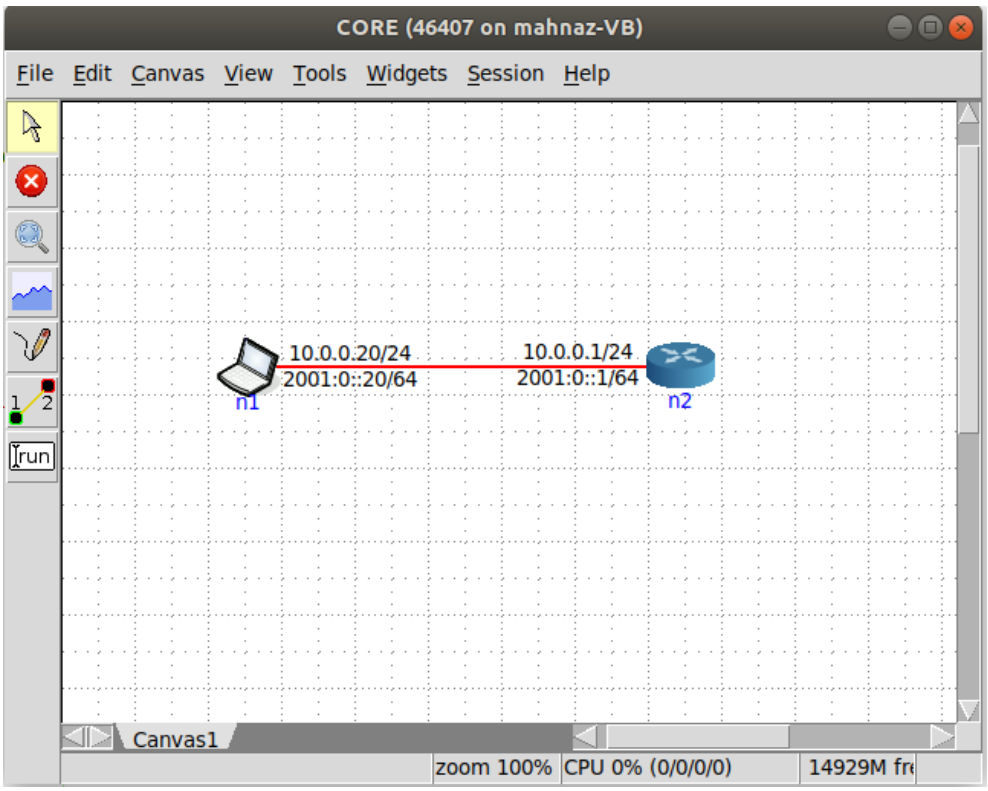

Figure 4.11 Simple Topology for Current CORE Implementation

Figure 4.12 and Figure 4.13 shows respectively, the workflow of creating the network namespace for CORE node using CORE API message and the logfile of CORE Daemon. The “SESSION” message is used to exchange information between different components. Then, the “EVENT” message indicates that the session has started, and it is changing from Definition State to Configuration State. In Configuration State, the nodes, link and some configuration information are sent to the CORE Daemon. The “CONF” message queries to get configuration data and then the “NODE” message triggers.

```
root@mahnaz-VB:~# core-gui
Connecting to "core-daemon" (127.0.0.1:4038)...connected.
>SESSION(flags=0,sids=0,name=untitled,nc=2,thumb=/tmp/thumb.jpg,user=root)
>EVENT(flags=0,type=1-definition_state,)
>EVENT(flags=0,type=2-configuration_state,)
>CONF(flags=0,obj=session,cflags=0types=<>,values=<>) reply
>CONF(flags=0,obj=location,cflags=0types=<2 2 10 10 10 10>,values=<0 0 47.579166
7 -122.132322 2.0 150.0>) reply
>CONF(flags=0,obj=services,cflags=0session=0xb547,types=<10 10 >,values=<PC Defa
ultRoute>) reply
>CONF(flags=0,obj=services,cflags=0session=0xb547,types=<10 10 10 10 10 >,values
=<router zebra OSPFv2 OSPFv3 IPForward>) reply
>NODE(flags=add/str,n1,type=0x0,n1,m=PC,x=193,y=174)
>NODE(flags=add/str,n2,type=0x0,n2,m=router,x=442,y=173)
>LINK(flags=1,1-2,0,0,0,0,0,0,type=1,if1n=0,10.0.0.20/24,2001:0::20/64,00:00:00:aa
:00:00,if2n=0,10.0.0.1/24,2001:0::1/64,00:00:00:aa:00:01,)
```

Figure 4.12 Workflow of Creating Network Namespace

File: /var/log/core-daemon.log

File: /var/log/core-daemon.log

```
2019-07-13 15:42:11,673 - INFO - coreemu:add_node - creating node(CoreNode) id(1) name(n1) start(True)
2019-07-13 15:42:11,680 - INFO - vnode:startup - Created New Netwotk Namespace: n1
2019-07-13 15:42:11,681 - INFO - vnode:mount - node(n1) mounting: /tmp/pycore.46407/n1.conf/var.run at /var/run
2019-07-13 15:42:11,683 - INFO - vnode:mount - node(n1) mounting: /tmp/pycore.46407/n1.conf/var.log at /var/log
2019-07-13 15:42:11,686 - INFO - service:add_services - using default services for node(n1) type(PC)
2019-07-13 15:42:11,686 - INFO - service:add_services - setting services for node(n1): ['DefaultRoute']
2019-07-13 15:42:11,686 - INFO - service:add_services - adding service to node(n1): DefaultRoute
2019-07-13 15:42:11,686 - INFO - coreemu:add_node - creating node(CoreNode) id(2) name(n2) start(True)
2019-07-13 15:42:11,691 - INFO - vnode:startup - Created New Netwotk Namespace: n2
2019-07-13 15:42:11,691 - INFO - vnode:mount - node(n2) mounting: /tmp/pycore.46407/n2.conf/var.run at /var/run
2019-07-13 15:42:11,694 - INFO - vnode:mount - node(n2) mounting: /tmp/pycore.46407/n2.conf/var.log at /var/log
```

Figure 4.13 CORE Daemon Logfile

The “NODE” message also consists of the type, medium and the position of the node. Here, node type = 0x0 denotes the default network namespace based virtual node (Figure 4.12). At this point, `add_node()` method has been called by CORE Daemon to add the node to the session based on the data provided on CORE GUI. Then the vnoded process creates the new namespaces n1 and n2 by invoking the `startup()` method (Figure 4.13). After creating the namespace for nodes, the “LINK” message signals the connection between n1 and n2 nodes. This message also includes the information about interface name and IP addresses of the corresponding nodes (Figure 4.12).

Figure 4.14 depicts the connection between peer to peer nodes n1 and n2. The process `coreemu` calls the method `add_link()` to create the links between the nodes. Then the `vnet` process implements virtual network by invoking `startup()` method by using Linux Ethernet bridging technique. After that, `coreemu` process invokes the `add_link()` method again to connect the node to network by using the bridge ID. When the node and network is linked, then `vnode` process triggers `newnetif()` method to create the new virtual interface for the nodes.

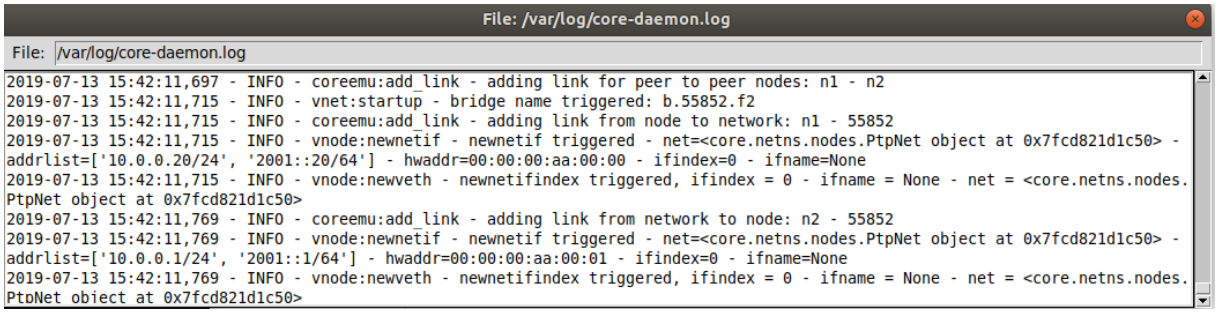

Figure 4.14 CORE Daemon Logfile for Linking Nodes

CORE emualtor provides the possibility of configuring a session and enter some configuration parameters for an network emulation. The Session menu of CORE has an entry called Options. It is used to configure a pre-session options such as setting control network, preserve session directory and so on which is presented in Figure 4.15.

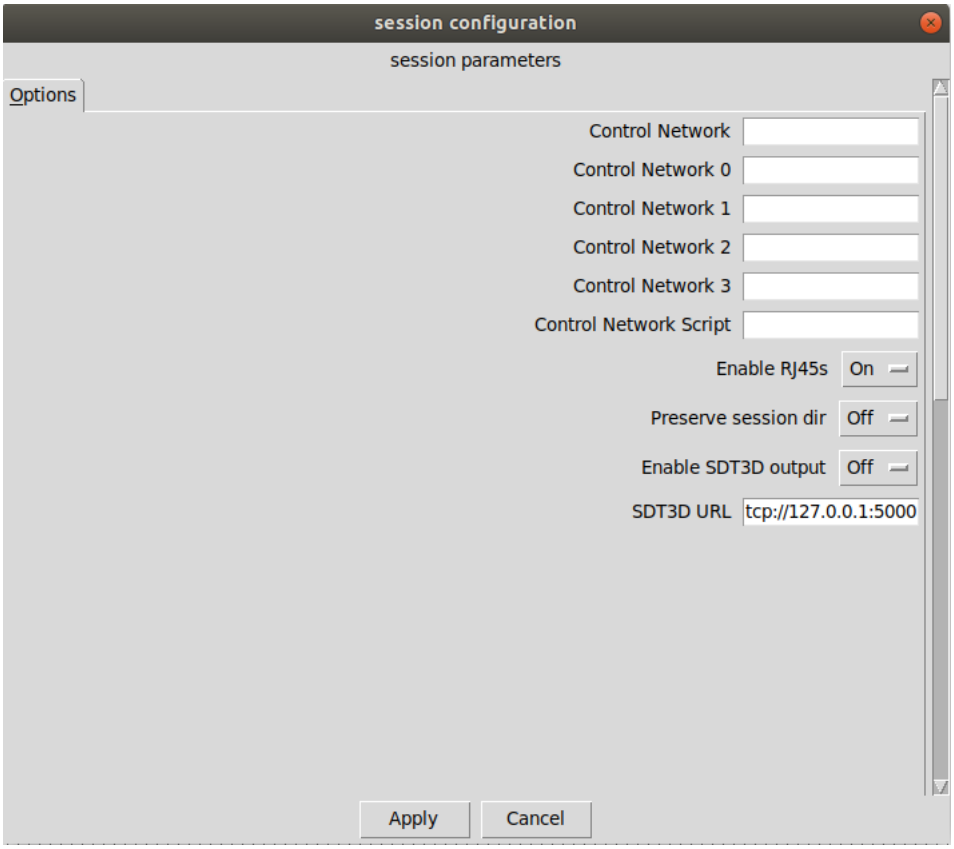

Figure 4.15 Session Configuration Options

Based on this implementation, it is possible to adapt this for LXC node configuration. The configuration message is shown in Figure 4.16 through CORE API message flow.

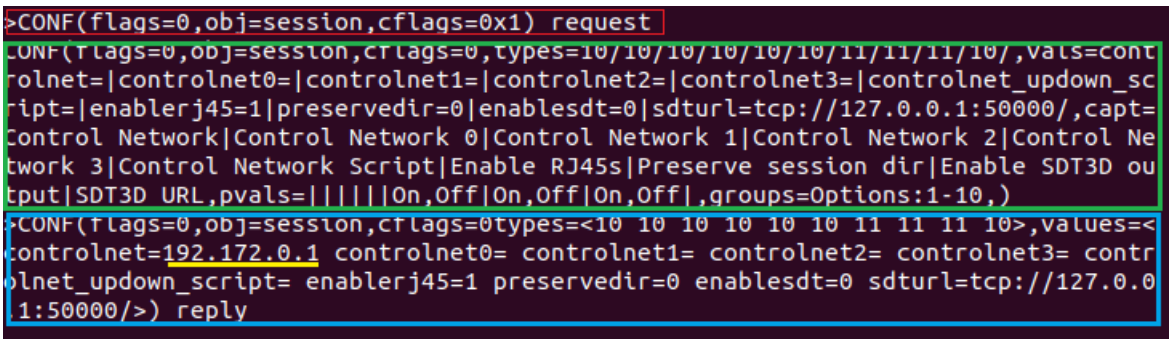

Figure 4.16 CORE Configuration Message through CORE API Message Flow

The “CONF” message transfers configuration data between the components. This message contains some configuration object TLV. The TLV flag 0x1 indicates that it is a request for configuration data. In response to this message the external entity sends the types, values of parameters and captions. If user enters data to this fields, CORE sends back the information to the external entity. In Figure 4.16, the yellow colour marked value is user-defined. Once the dialog box is filled, CORE will send back the filled values to the external entity.

Lastly, when the session stops, it is reached to Shutdown State from Datacollect State which is demonstrate in Figure 4.17. The “LINK” message contains the flags=2 which means to remove the link between the nodes. Then “NODE” message shows to delete the nodes and “CONF” message indicates to reset (cflags=0x3) the configuration for a new session emulation. The CORE Daemon logfile is also showing the session has stopped and it is in Shutdown state.

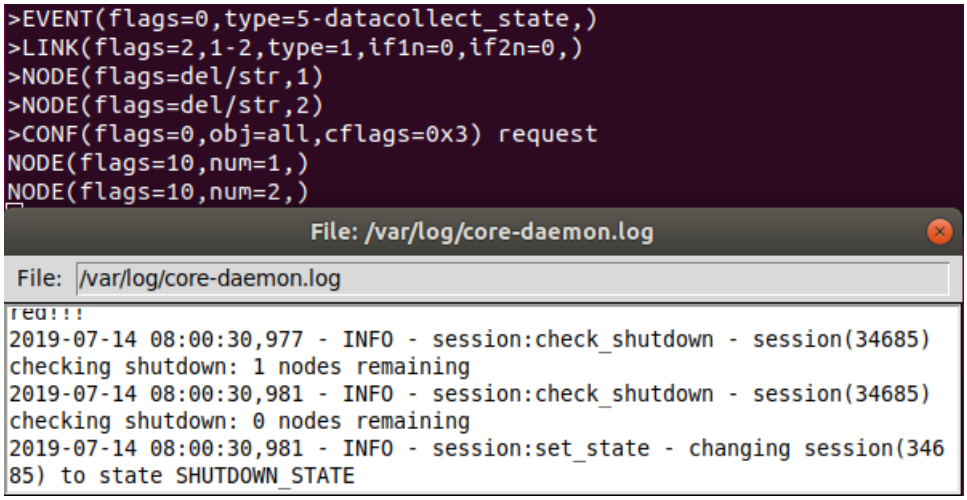

Figure 4.17 Session Termination and Logfile of CORE Daemon

Based on this above investigation of CORE GUI and CORE Daemon via CORE API message, the targeted approach will be, to extend the current CORE emulator implementation for integration of LXC container into CORE network emulator. As CORE uses network namespace to create virtual nodes, so a new node type is required to support LXC container into CORE emulator. Therefore, in CORE GUI there will be an implementation for new node type called “LXC” node type. To work with LXC container inside CORE, there will be some implementation inside CORE Daemon. Additionally, there will be a development of separate configuration manager to retrieve possible configuration parameters for LXC node. This configuration manager will be developed based on the concept of CORE session configuration part. As CORE uses opaque data to transfer configuration data, so it will be convenient way to apply on LXC node regarding transferring LXC specific configuration information. In case of

connecting to other nodes, it is possible to extend the functionality of CORE nodes to network connection. All these implementations will be discussed on later chapters.

4.3 Integration of a Selectable LXC Node Type into CORE GUI

In this section, there will be a detail description on how the new node type LXC has been implemented into the CORE emulator. The implementation for LXC node type took part in both components respectively CORE GUI and CORE Daemon. First, the discussion will be regarding the integration of a selectable LXC node type in CORE GUI.

As mentioned before in the requirement analysis of this thesis work that Tcl-Tk programming language has been used for the implementation part of CORE GUI. In current CORE implementation, CORE GUI has several files for creating different node types with different names and icons.

To set the icon for LXC node `nodes.tcl` provides the support for some built-in icons for the node types which can be used in this implementation. There is an array of built-in node types in that file and LXC has been added to that array. Below code snippet is showing that for implementing LXC node host.gif icon has been chosen:

```
Array set g_node_types_default {

      ....

      OVS landswitch.gif landswitch.gif {}DefaultRoute SSH OvsService} OVS {} }

      lxc host.gif host.gif {} lxc {} }

}
```

Then it is required to setup the machine type for LXC node like the below line in the same file `nodes.tcl`

```
set MACHINE_TYPES "netns physical OVS lxc"
```

After that, the following line should be added at the end of the file.

```
proc lxc.layer {} { return NETWORK }
```

Basically, LXC node type creation is generated from the concept of OVS switch implementation. So, at first Figure 4.18 shows the procedure to draw node's icon on CORE GUI which will be taken place in CORE `editor.tcl` file. In this file, there are different notation for different node types such as router, pc, host, OVS. So, to draw the node's icon, it is required to add the LXC notation first. Then, a condition needs to be checked whether the node type is LXC or other type. If it is true, then it will set the image for LXC node otherwise not.

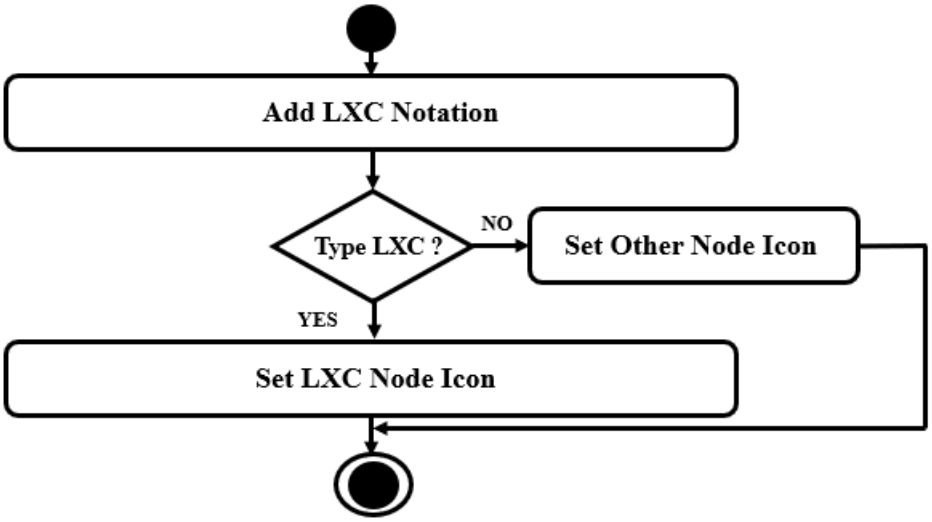

Figure 4.18 UML Activity Diagram of Drawing LXC Node Icon

Figure 4.19 shows that the icon for LXC node has been added to the toolbar of CORE emulator. After connecting CORE GUI to CORE Daemon, from the left side toolbar LXC node type can be selected like this.

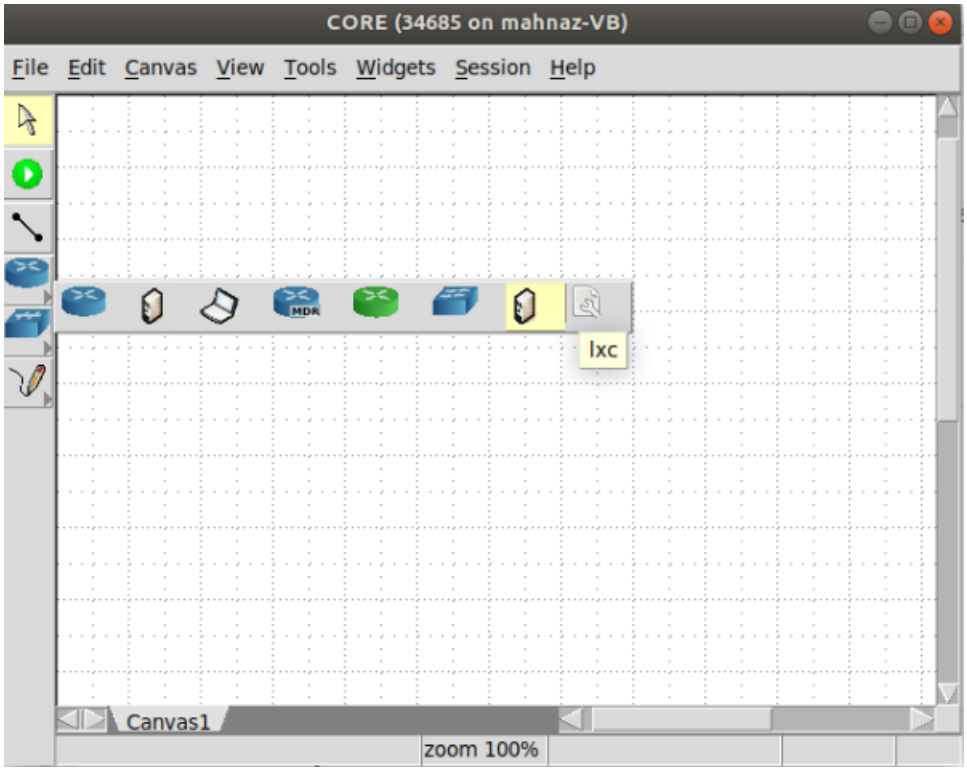

Figure 4.19 LXC Node Type in CORE Toolbar

The next procedure is called when the button is pressed on the canvas to create a new node. Figure 4.20 shows the UML activity diagram of creating the node on canvas. To enable the button for LXC node, first the LXC notation has to be added. Then a condition is checked that the activetool variable has the name “LXC” or not. If it is true then when the button is pressed on the canvas, it will create the new node LXC otherwise other node.

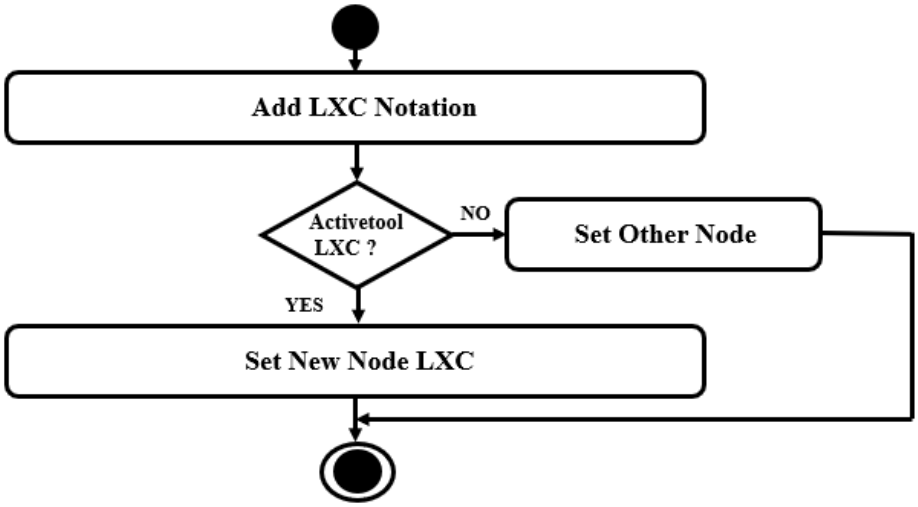

Figure 4.20 UML Activity Diagram for Creating Node on the Canvas

Therefore, this is the first step to create the LXC nodes's icon and image. In the next chapter, the implementation of popup configuration box, and the creation of the Configure button will be explained first for giving further support to the configuration of LXC node type.

4.4 Provision of User-definable Configuration Options for an LXC Node Type via the CORE GUI

In this section, there will be a detail discussion regarding the implementation parts which are needed to realise the "Configuraion" button for LXC node. Since CORE nodes use network namespace virtualisation, the new node type is required to support LXC container in CORE. And therefore, to work around with the LXC container into CORE emulator, the container needs to be configured. In this regard, several changes are made inside CORE Daemon python modules as well as in CORE GUI. Before starting to explain the actual implementation of LXC node, it is necessary to describe some additional implementation of user-definable configuration options for LXC node first.

4.4.1 Create the button for Configuration in CORE GUI

After adding the node in the CORE canvas, it is required to create the button for configuration of LXC node. This implementation took place in the same CORE GUI `editor.tcl` file. Figure 4.21 presents the procedure in UML activity diagram. After creating the node, the next procedure is called to create a popup configuration dialog box if the node type is LXC. The configuration dialog box will be opened by double-clicking on the node. In this configuration box, it is possible to set some text and also a window size is defined. Then, a button for configuration has been created.

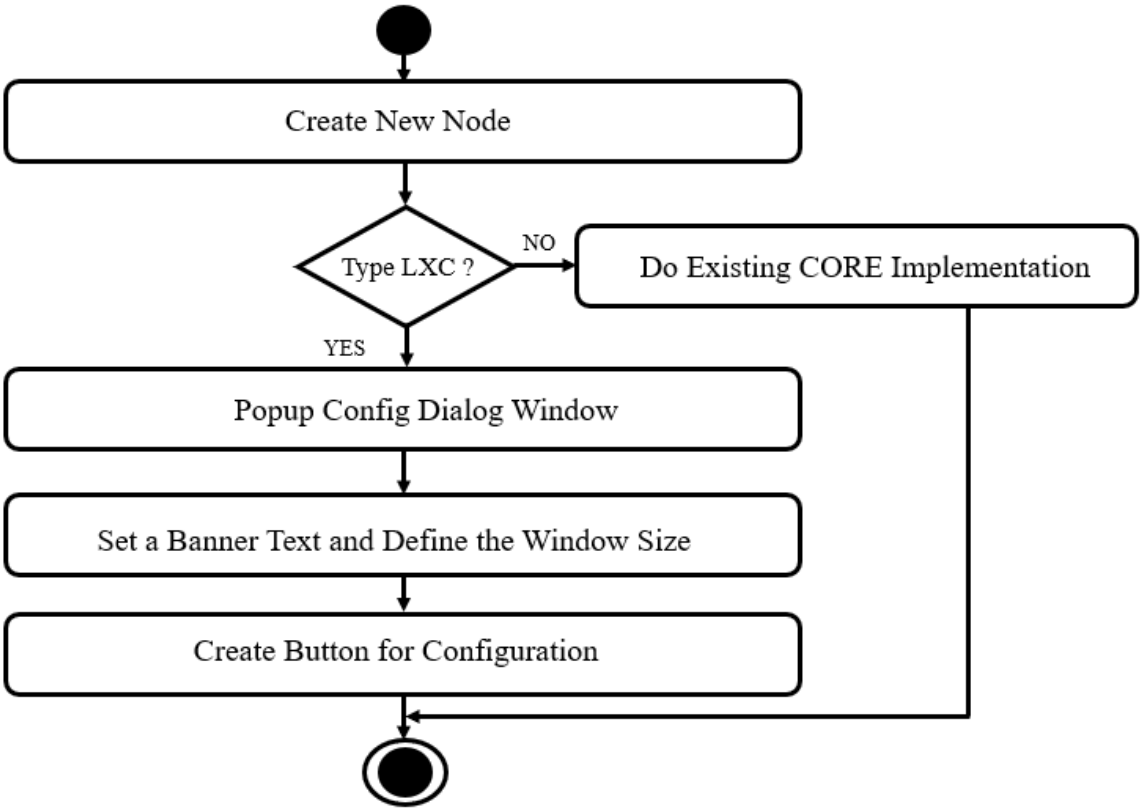

Figure 4.21 UML Activity Diagram for Opening Configuration Dialog Window

Figure 4.22 presents that user can observe this by right-click on the n1 node and select the option Configure or double-lick on the node. Then, a popup configuration dialog box will be opened to configure LXC node.

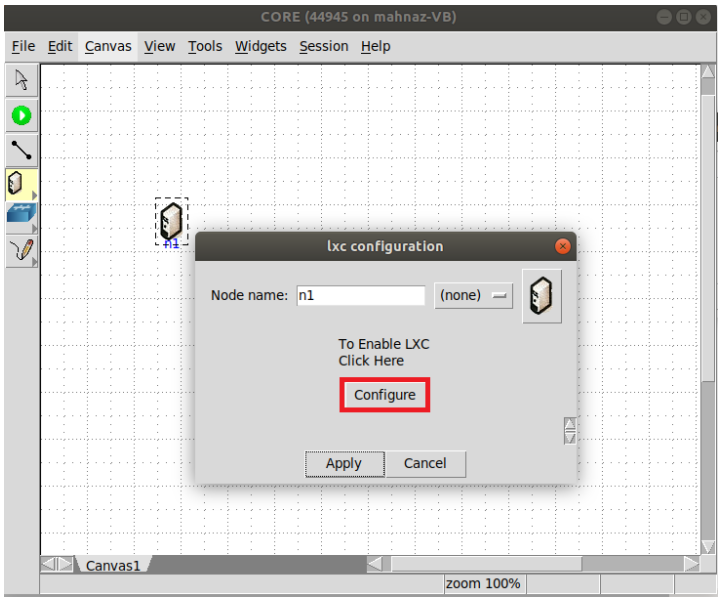

Figure 4.22 Popup Configuration Window for LXC Node

4.4.2 Implementation of LXC Configuration Manager

As mentioned before in chapter 4.2 that it is possible to adapt the implementation of session configuration for new node type LXC. For the implementation of session configuration, CORE has developed the classes respectively, `ConfigurableOptions` and `ConfigurableManager`. The class `ConfigurableOptions` provides the base class for specifying the configuration options for CORE node and the class `ConfigurableManager` helps to store and retrieve the configuration options for CORE nodes. Therefore, a new instance called `Lxc-ConfigManager` will be generated which will inherit the implementations of above-mentioned classes. This new instance will be used for specifying and retrieving all possible configurable aspects for LXC node type.

For this implementation, a new class called `LxcConfigManager` has been created which will extend the functionality of CORE configuration manager classes (`ConfigurableOptions, ConfigurableManager`) to store and retrieve configuration options for LXC node type. Figure 4.23 shows the procedure in an UML activity diagram. At first, an object name has been defined so that it can be dedicated only for LXC configuration. Then, a list has been declared for the configuration options. This is the base for all configurable options. After that, the constructor for `LxcConfigManager` has been initialised to set the configuration values such as image selection, usage limit for CPU/RAM, mount directory and start-up command. This will help to append the configurable aspects to the option list. To avoid the duplication of the configuration parameters, the `del()` method will be invoked every time, whenever the constructor is being called and CORE GUI restarts. It will clear the option list.

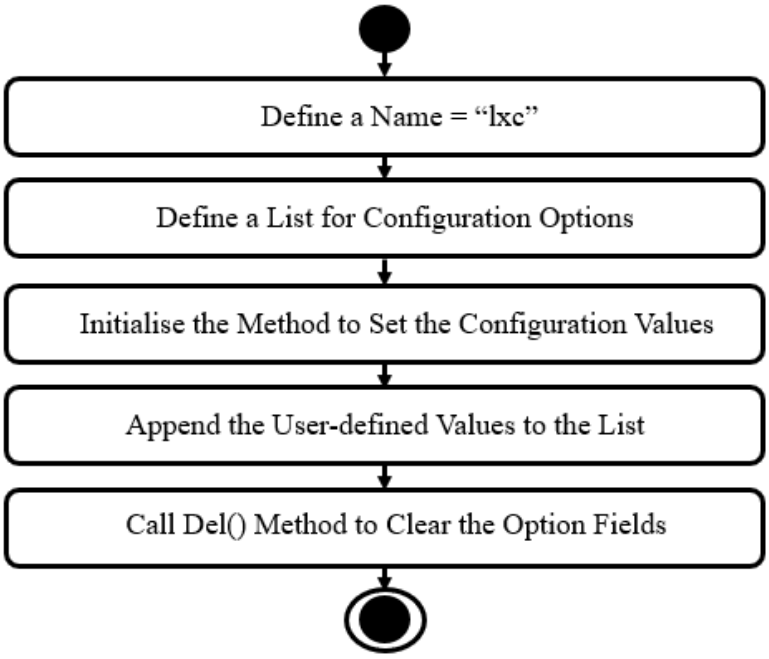

Figure 4.23 UML Activity Diagram for LXC Configuration Options

So, this is the procedure to retrieve possible configuration aspects for LXC node. Figure 4.24 demonstrates the configuration window for LXC node type. As soon as the Configure button is pressed, this window will popup for configuration. In order to use the LXC container into CORE emulator, the configurable options are as follows, Container-Image - to select template image for LXC container, CPUSet and Limit_Memory respectively, to limit the usage for CPU and RAM, mount-directory - to mount a directory from host machine, startup-command – to execute a command during runtime of a container.

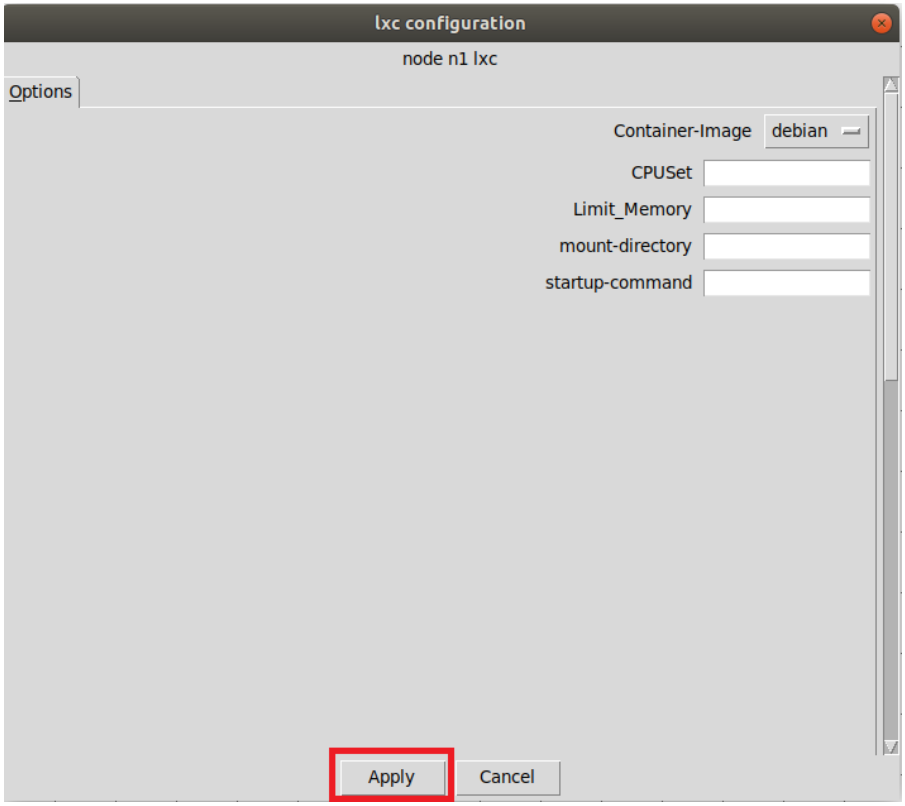

Figure 4.24 LXC Configuration Window

To manage the session for LXC node, the `LxcConfigManager` method needs to be initialised in CORE Daemon Session class. This method invokes right after CORE Daemon starts. This implementation has been shown in the following code snippet below.

```
class Session(object):
      def __init__():
            .....
            self.sdt = Sdt(session=self)
            self.lxc = LxcConfigManager(session=self)
```

Figure 4.25 depicts that the session has been initialised for LXC Config Manager and then the session has created.

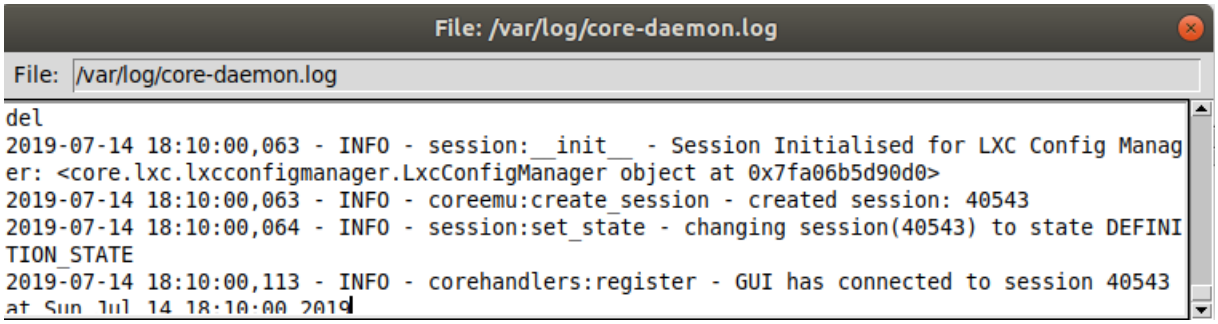

Figure 4.25 CORE Daemon Log for Session Initiation

4.4.3 Container Image Allocation

LXC container is an isolated container from host system which can run full-fledged operating system though LXC shares the same kernel as host system's kernel. There are different kinds of templates which is used to create various operating system containers. The templates provided in LXC are scripts to bootstrap particular operating system. Each operating system that is supported by LXC installation has a dedicated script for it. In general, all the images are available in the LXC containers image server which can be used by selecting "lxc-download" template. This generic script named as "download" can install many different Linux operating system distributions. But downloading the images at the staring of a session in CORE will consume a lot of time so, the alternate way to get the image from server is to use another template called "local". This "local" template can create images and can be used with distrobuilder tool.

There is a configuration manager to get the opportunity to select the image template from CORE GUI. Additionally, the configuration manager will be residing on CORE Daemon. User can be able to select the specified Linux distribution of choice. So, to perform this action at first there should be a predefined directory on the local host machine. User can define any path as desired. In this implementation, the image file path has been defined in the local directory "`/home/mahnaz/ContainerImages`". "`ContainerImages`" is the folder, where the template images have been stored. Then the method of `os` module, `os.chdir()` has been executed to change the current working directory to the given path and `os.getcwd()` has been called to return the working directory. After that `os.listdir()` method has been used to retrieve all the files and folders name from the given directory. In order to keep the folders in a place, an array has been declared. **Error! Reference source not found.** demonstrates the process of defining the path for the image folder and how to get the subfolders name from the current folder in an UML Activity Diagram.

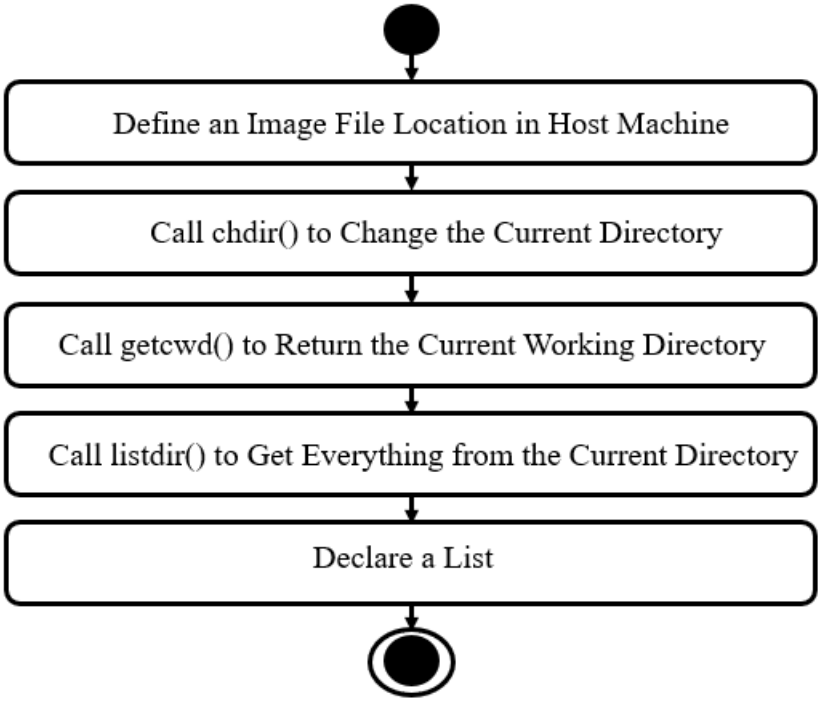

Figure 4.26 UML Activity Diagram of Container Image Allocation

After declaring the array list for the container images, the next task is to append the name of the folders to the Options array which is in the CORE GUI. Figure 4.27 shows the process of allocating the image from the local

predefined directory to the LXC node‘s configuration options. Here, two template/image are on the container image list.

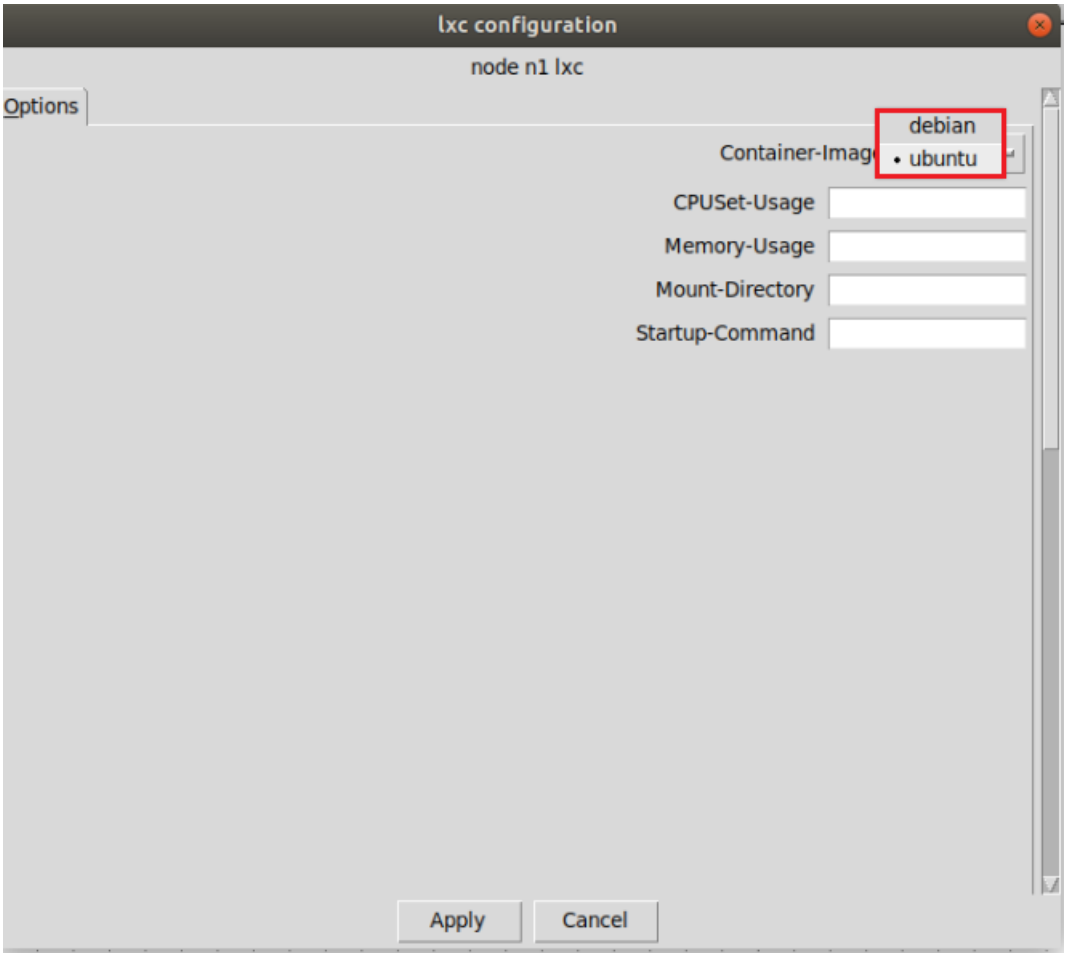

Figure 4.27 Dynamically Image Allocation to the CORE GUI Config Options

In this way, many image templates can be added to the configuarion option as desired. The whole image allocation process has been done dynamically. Based on the template images, LXC containers are being created and started inside CORE emulator network.

4.4.4 Implementation for LXC Node in CORE Handler

This section will explain various changes which is required for exchanging configuration information between CORE GUI and CORE Daemon. Before doing that, it is necessary to get some insight on current CORE handler functionality on CORE node.

CORE handler module of CORE is responsible for handling different situation in a CORE session. The first message with red colour marker in Figure 4.28 is defining that when GUI is connected to the running session it returns a message which means the current GUI is registered with current session. Then, to modify the session state, it returns the session message. If any message is not handled during the session, it throws a warning to handle the event message. And lastly, CORE handlers handle node’s location coordinate reference point as well. After setting the node in the CORE canvas, it measures the cartesian coordinate of a node and stores the measurement in the log file.

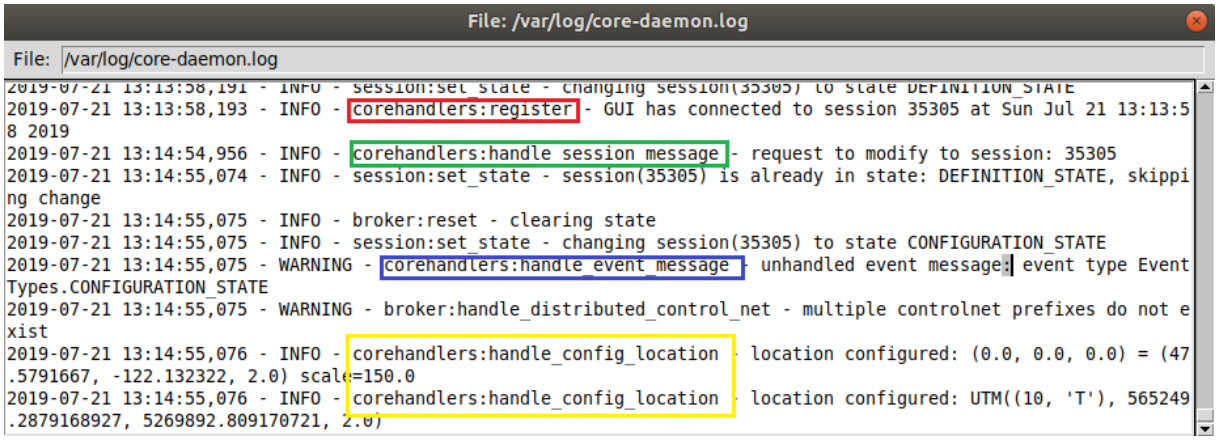

Figure 4.28 Log File of Current CORE Handler Message

So, this CORE handlers module of CORE Daemon can be implemented for transferring configuration information between CORE GUI and CORE Daemon for LXC node.

At first, the implementation will take place in CORE handler's `register()` method. It is required for registering TLV so that when the GUI is connected, it will trigger a notification. This addition to the method is showing through a code snippet below:

```
def register(self):

  …

  tlv_data += coreapi.CoreRegisterTlv.pack(self.session.lxc.config_type, self.session.lxc.name)

  return coreapi.CoreRegMessage.pack(MessageFlags.ADD.value, tlv_data)
```

the next section describes the condition to identify the incoming configuration request from CORE GUI in a code snippet below:

```
def handle_config_message(self, message):

        …

        elif config_data.object == self.session.lxc.name:

                replies = self.handle_config_lxc(message_type, config_data)

        else:

              raise Exception("no handler for configuration: %s", config_data.object)

        …
```

And, lastly there will be an additional implementation for returning the configuration information provided by the user in LXC configuration option fields. this method invokes if the message type is request then it requests for the configuration data and return the response with user-defined information

```
def handle_config_lxc(self, message_type, config_data):

        replies = []

        node_id = config_data.node

        object_name = config_data.object
```

```
        logging.info("Received config message for LXC - message_type=%s - config_data=%s", mes-
sage_type, config_data)

        if message_type == ConfigFlags.REQUEST:

                config = self.session.lxc.get_configs()

                logging.info("Returning configs - %s", config)

                config_response     =     ConfigShim.config_data(0,     config_data.node,     Con-
        figFlags.NONE.value, self.session.lxc, config)

                replies.append(config_response)

                return replies
```

Figure 4.29 shows the log file for CORE handlers message. The `register()` method triggers when the GUI is connected to the session. Then it recognised the request message from GUI for the object type is LXC and query about the configuration values (marked in red). After received the information from user, it returns the configuration parameters such as Container-image, CPUSet, Memory Usage, Mount-directory and Start-up-command (marked in green).

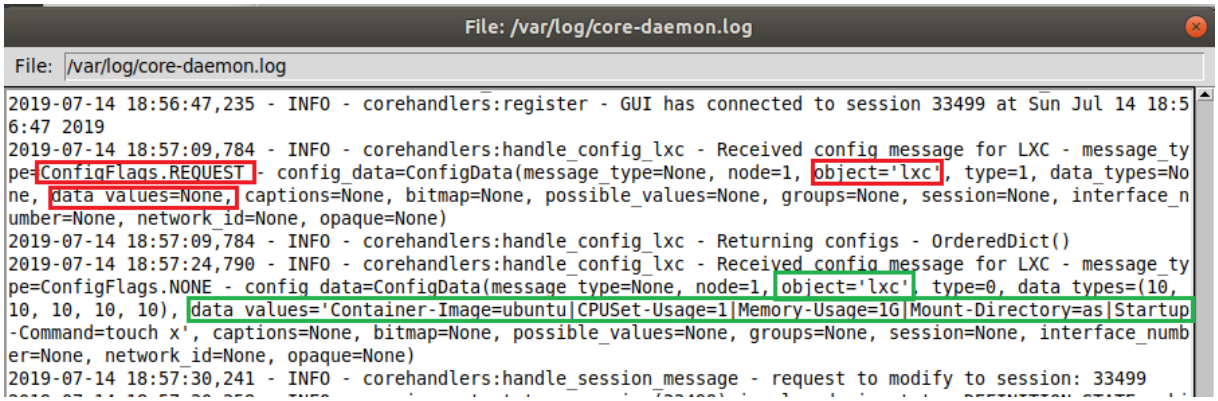

Figure 4.29 CORE Daemon Log of CORE Handlers Message for LXC Node

4.4.5 CORE API Message Implementation for LXC Node

In this part, there will be a discussion on CORE API message communication between CORE Daemon and CORE GUI based on current CORE API message communication. To provide the communication interface between CORE GUI and CORE Daemon for LXC node, it extends the current CORE API implementation.

At the beginning, the LXC node type implementation has been started by setting up the node type so that the API message realise the message flow for LXC node. In CORE `api.tcl` file, it is required to add the node type for LXC in both decimal and Hex format respectively, 70 and 0x46 to interpret the recieved API message for LXC node type. LXC notation has been added in decimal format below:

```
array set nodetypes { 0 def 1 phys 2 tbd 3 tbd 4 lanswitch 5 hub 6 wlan 7 rj45 8 tunnel \

           9 ktunnel 10 emane 70 lxc}
```

In Hex format:

```
proc getNodeTypeAPI { node } {

     ....

     switch -exact -- $type {
```

```
    router { return 0x0 }
    ....
    lxc { return 0x46 }
    default { return 0x0 }
    }
}
```

Here, Figure 4.30 shows the sequence diagram of message flow between CORE GUI and different components of CORE Daemon. CORE Daemon is distinguished from CORE GUI by drawing a dotted line in grey colour. This sequence diagram is describing the message flow for node selection, configuration and the corresponding message to do that. At first, the LXC node type has been selected from the left side toolbar and added to the CORE canvas. Then click on the node for configuration. This will check a condition whether the configuration is for LXC or not. If the object type is LXC then it will request for configuration. Here, CORE handlers are used to exchange the message communication between CORE GUI and CORE Daemon. CORE handlers forward the request to LXC Config Manager to ask about the configuration options. LXC Config Manager contains all the configuration options which is required for LXC. From LXC Config Manager, CORE handlers retrieved the configuration entries or options and show those to CORE GUI via CORE API message. On the other hand, if the object type is not LXC then it will not request for configuration because the configuration is only dedicated for LXC node type. Now, once it identified the object type is LXC, it will open the configuration window. User will enter the data in the configuration fields and click apply to map the information for LXC node. Once the "Apply" button is clicked, it will show a message to accept the configuration information for LXC node.

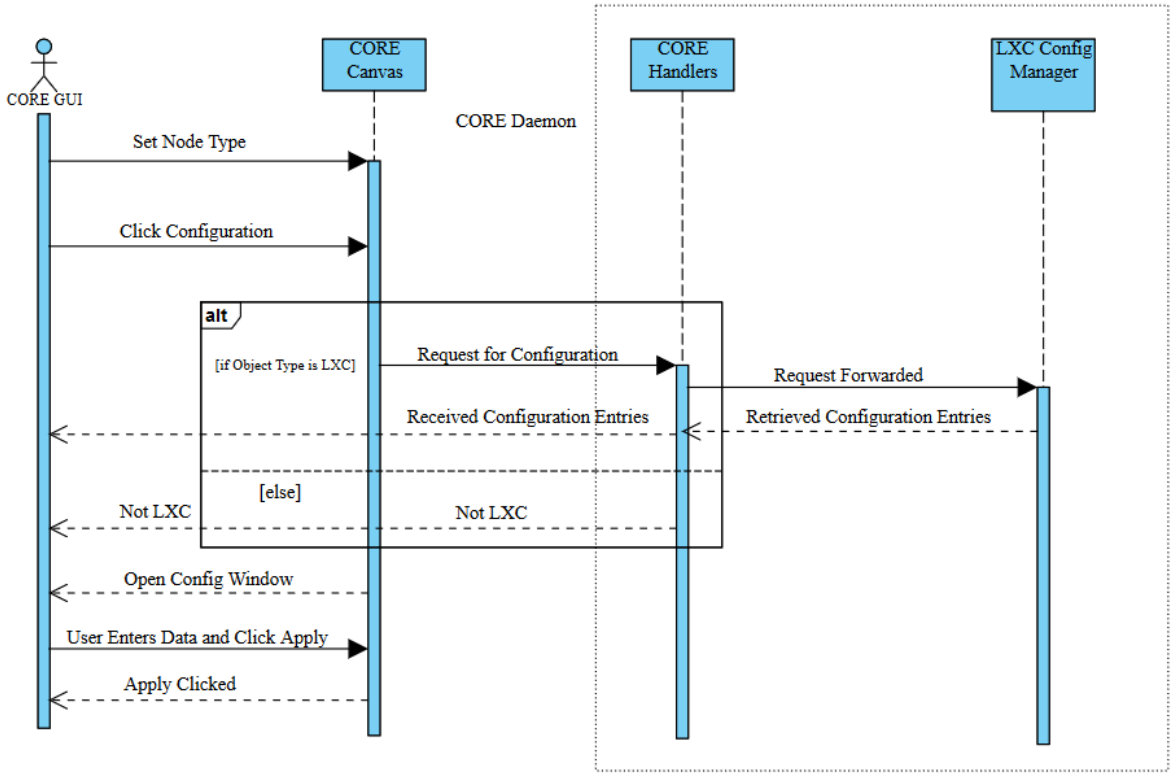

Figure 4.30 Sequence Diagram of CORE API Message Communication between CORE GUI and CORE Daemon

The following Figure 4.31 shows the configuration API message for LXC node. This message appears when the "Configuration" button is pressed. The CONF request contains flags, number of node = 1, object type = lxc and the flag = 0x1 to indcate the request message. Configuration message contains various TLVs. It must includes configuration object TLV to verify the object has been configured.

The next CONF message brings the configuration entries from LXC Configuration manager options. This message containes the object type lxc, the number of node information. The values TLV can contain more than one string. Each string is considered as single value and the values as well as the captions are separated by a pipe ("|") symbol. These values are the user-defined configuration entries.

```
>CONF(flags=0,node=1,obj=lxc,cflags=0x1) request
CONF(flags=0,node=1/obj=lxc,cflags=0,types=10/10/10/10/10/,vals=Container-Image= |CPU
Set=|Limit_Memory=|mount-directory=|startup-command=,capt=Container-Image|CPUSet|Limi
t_Memory|mount-directory|startup-command,pvals=debian,ubuntu||||,groups=Options:1-5,)
```

Figure 4.31 Screenshot of CORE API Configuration Request and Response Message

The following Figure 4.32 shows the response when the user enters the data for configuration and click the "Apply" button. It contains the configuration data that is supplied by the configuration manager window. This implementation part has been done in CORE GUI `plugins.tcl` file. This message will apear when the "Apply" button is pressed to apply the configuration.

```
Apply Clicked!
node: n1
model: lxc
types: 10 10 10 10 10
vals: Container-Image=ubuntu CPUSet=1 Limit_Memory=1G mount-directory= startup-c
ommand=
>CONF(flags=0,node=1,obj=lxc,cflags=0types=<10 10 10 10 10>,values=<Container-Im
age=ubuntu CPUSet=1 Limit_Memory=1G mount-directory= startup-command=>) reply
```

Figure 4.32 User-defined Configuration Parameters

The message flow for CORE handlers are described in the previous chapter.

4.5 Integration of a Lifecycle Management for LXC Node Type into CORE Daemon

This section will be used to discuss the actual LXC node implementation and observe the lifecycle management of LXC container inside CORE Daemon. Moreover, there will be some clarification of the configurable aspects of LXC container.

4.5.1 Implementation of LXC Node Type in CORE Daemon

To realise the LXC node type by CORE Daemon, the node type has been added in CORE Daemon `enumeration.py` file in decimal type as CORE GUI. For better understanding, this part is shown in small code snippet below:

```
class NodeTypes(Enum):

        DEFAULT = 0

        ...
```

```
EMANE_NET = 14

LXC = 70
```

So, the LXC notation has been added in decimal format to the file. This is necessary to interpret received API message from CORE GUI for LXC node.

4.5.2 Integration to Node Maps

CORE Daemon python module `nodemaps.py` provides the possibility to support LXC node. So, LXC node type has been added to the list of node types in CORE Daemon. To show the change, a small code snippet has given below:

```
NODES = {

NodeTypes.DEFAULT: nodes.CoreNode,

.....

NodeTypes.CONTROL_NET: nodes.CtrlNet,

NodeTypes.LXC: reallxcnode.RealLxcNodeType

}
```

Therefore, the reason behind adding this line to this file is, to ensure the usage of correct implementation for received node type. If the received node type is LXC, then it will call the implementation for `RealLxcNodeType` as this is the actual implementation of LXC node type for this thesis work. To avoid the naming conflict with CORE implementation, in this implementation the name is given as `RealLxcNodeType` because CORE has already used the name `LxcNodeType`.

4.5.3 Actual Implementation for LXC Node

This chapter contains the actual implementation part for LXC node. The new node type has been created and the configuration options part has been created as well. Therefore, it is required to extend the current CORE Daemon implementation to support LXC container as node type. In the current CORE emulator implementation, `SimpleLxcNode` class implements the network namespace virtual node. As CORE emulator has programmed in a structured way so, it is possible to omit network namespace node and inherit the functionality of `SimpleLxcNode` class for LXC node implementation.

Figure 4.33 shows briefly the implementation process of LXC node into the CORE emulator. The process begins with starting the session. Add the node from CORE GUI and the node has extended from `SimpleLxcNode` which is the base class for CORE node. Then the node has initiated to provide the functionality for LXC node. Therefore, it fetched the configuration parameters to instantiate the LXC node. The configuration parameters are stored in a separate configuration manager. Based on the information the container has inialized and after that it has created and then started. If user stops the session, the shutdown message will trigger to shutdown the nodes and links. Then, the container will be stopped and immediately destroyed from the container list.

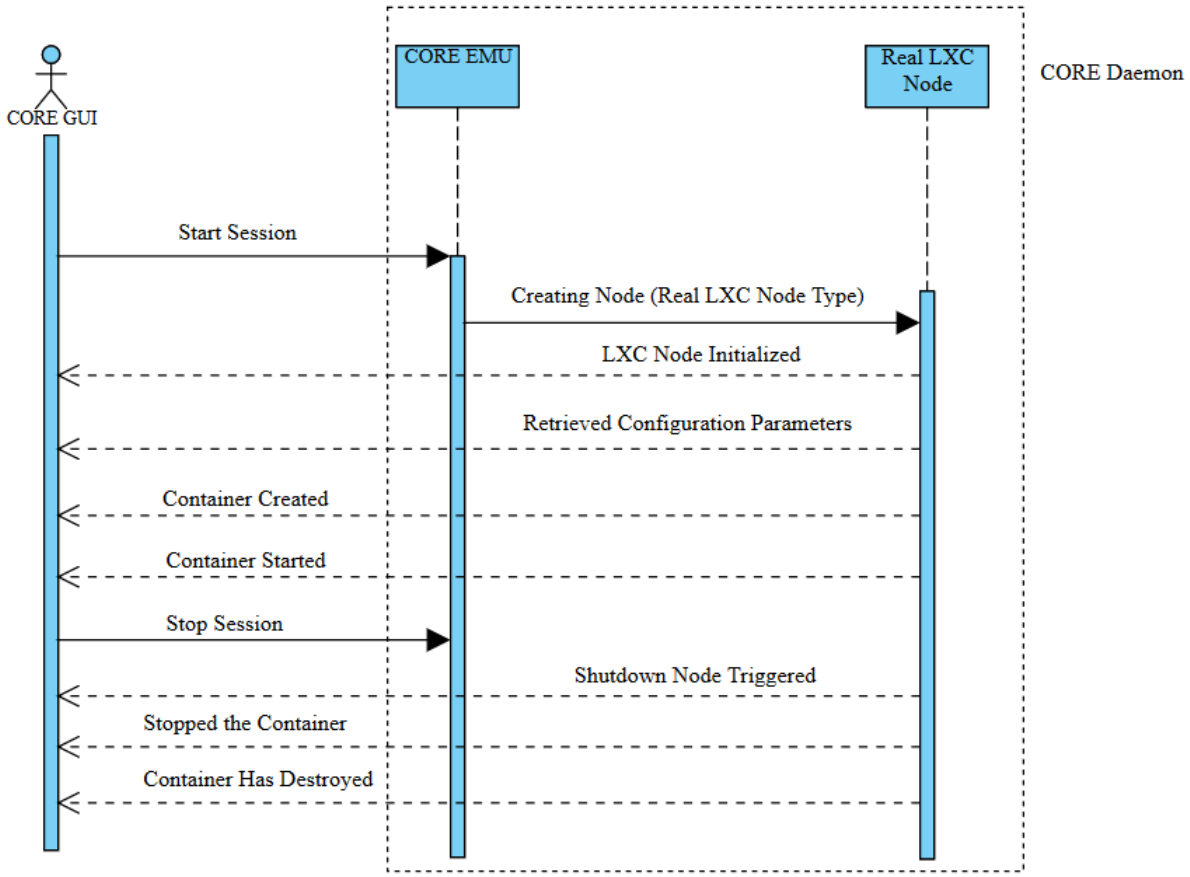

Figure 4.33 Sequence Diagram of Actual Implementation of LXC Node

Figure 4.34 demonstrate the log file of LXC node creation inside CORE Daemon.

File: /var/log/core-daemon.log

File: /var/log/core-daemon.log

```
2019-07-14 18:57:30,360 - INFO - coreemu:add_node - creating node(RealLxcNodeType) id(1) name(n1) start(Tr
ue)
2019-07-14 18:57:30,360 - INFO - reallxcnode:__init__ - Init triggered!
2019-07-14 18:57:30,360 - INFO - reallxcnode:__init__ - opaque = None
2019-07-14 18:57:30,360 - INFO - reallxcnode:initialiseLXC - initialiseLXC triggered!
2019-07-14 18:57:30,360 - INFO - reallxcnode:initialiseLXC - opaque = {"Container-Image": "ubuntu"
,"CPUSet-Usage": "1"
,"Memory-Usage": "1G"
,"Mount-Directory": "as"
,"Startup-Command": "touch x"
}
2019-07-14 18:57:30,360 - INFO - reallxcnode:initialiseLXC - session = <core.emulator.coreemu.EmuSession o
bject at 0x7f5d10f07bd0>
2019-07-14 18:57:30,360 - INFO - reallxcnode:initialiseLXC - name = n1
2019-07-14 18:57:30,360 - INFO - reallxcnode:initialiseLXC - objid = 1
2019-07-14 18:57:30,360 - INFO - reallxcnode:initialiseLXC - retrieved container-image: ubuntu
2019-07-14 18:57:30,361 - INFO - reallxcnode:initialiseLXC - retrieved cpu: 1
2019-07-14 18:57:30,361 - INFO - reallxcnode:initialiseLXC - retrieved ram usage: 1G
Unpacking the rootfs

---
You just created an Ubuntu bionic amd64 (20190714_1302) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
2019-07-14 18:58:08,664 - INFO - reallxcnode:initialiseLXC - container created - n1
2019-07-14 18:58:08,752 - INFO - reallxcnode:initialiseLXC - container started - n1
```

Figure 4.34 CORE Daemon Log for LXC Node Creation

4.5.4 CORE API Message Flow Between CORE GUI and CORE Daemon after LXC Node Creation

The Figure 4.35 presents the further message flow of CORE API message between CORE GUI and CORE Daemon. After the configuration is done, then "Start" button is pressed to start the session. The session will be modified from Definition state to Configuration state. Then, CORE Daemon's module CORE emu creates the LXC node. This message contains the flag for creating the node, the type = 0x46 which indicates LXC node type and opaque data for passing configuration information. Once the node type is detected, it will respond with a message that LXC type is found. After that, the Link message defines the connection between two nodes. Once the node and link configuration are sent, it will reach to instantiate state.

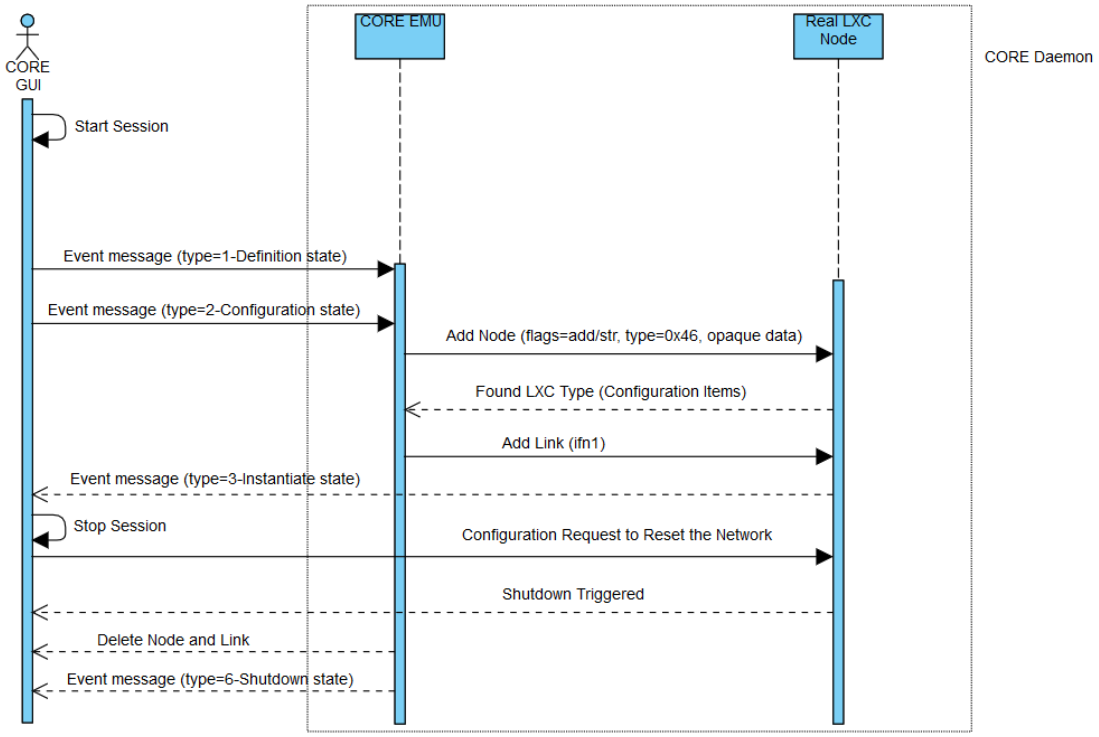

Figure 4.35 Sequence Diagram of CORE API Message Flow for Node Creation and Deletion

Now, if the "Stop" button is pressed, CORE GUI sends request to reset the configuration. At this point, LXC container shutdown has triggered. Eventually, the nodes and links will be deleted and thus, it will reach to the shutdown state.

4.5.5 Opaque Data Separation

In this thesis task, opaque data is being used to pass any type of information between CORE GUI and CORE Daemon. For LXC node type implementation, several configuration parameters are required to use. As a consequence, this might create some redundant issue to handle different information on same data type. Therefore, in order to get rid of the problem, the configuration parameters need to be split before execution. So, when the "Apply" button has been clicked all the configuration parameters are fetching by `getCustomConfig`. At this point, to parse the information JSON format will be used to separate the configuration parameters. The JSON package delivers a TCL library for parsing the JSON.

To use JSON, install TCL library on host system

```
$ sudo apt-get install tcllib
```

To parse in JSON formatted string, iterate a loop. The configuration options which is on the left-side of the equal (=) sign are denoted as key and right-side elements are considered as value. Based on the equal (=) sign the strings has been separated and the JSON package has returned the data as Tcl dict. This means, multiple JSON entities contains list of dictionaries and has returned the list. So, in CORE Daemon, user gives input of the configuration options. JSON loads the information and parse it to split into multiple strings.

4.5.6 Limiting Maximum CPU/RAM Usage

Control groups or cgroups is one of the prime Linux kernel features which limits, allocates and isolates system resources such as CPU, memory, disk I/O and network usage of one or many processes.

lxc-python bindings are capable to specify the usage for a container. Using `set_cgroup_item()` and `get_cgroup_item()` methods, can conveniently change the configuration options and later on query them on a running container. In case of limiting the usage of RAM in container, `get_cgroup_item()` method is used to retrieve the maximum usage of RAM in bytes and then `set_cgroup_item()` method specifies the user-defined value to limit the usage. After that, `get_cgroup_item()` method has called to return the value of memory usage in bytes. Figure 4.36 presents that user has specified to limit the memory usage to 1 Gigabytes (G) in the n1 LXC container in CORE GUI.

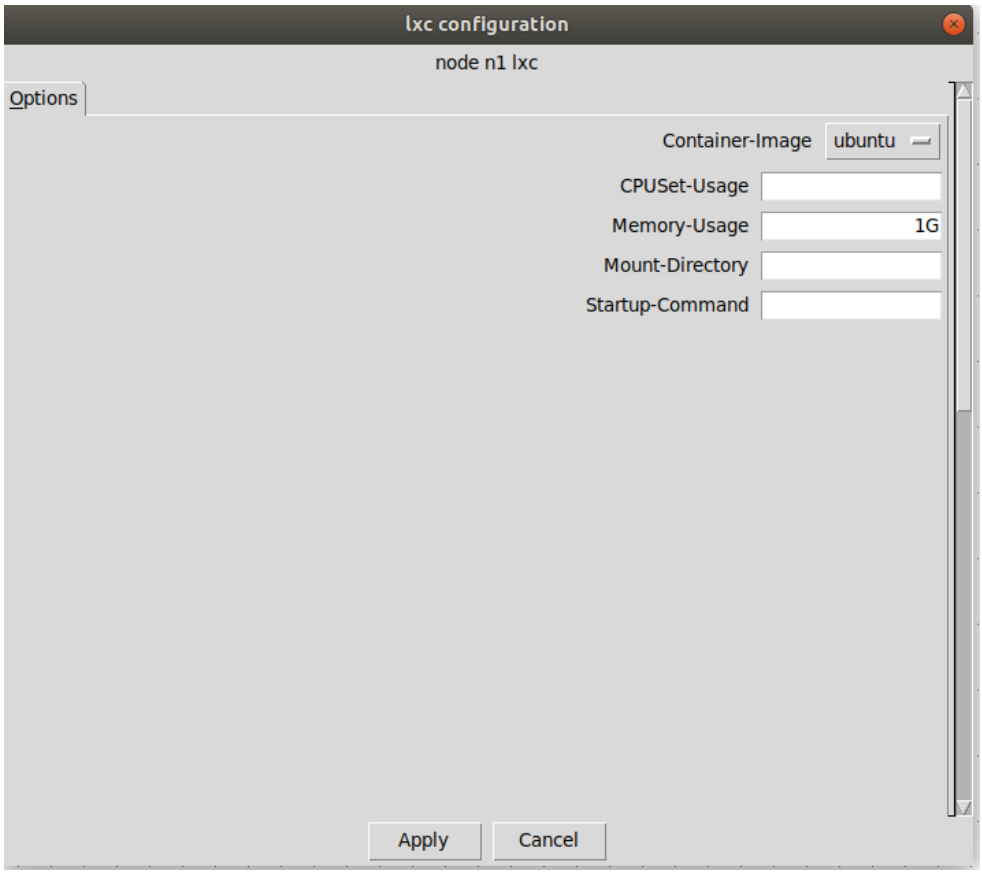

Figure 4.36 User-defined Memory Usage in CORE-GUI

Figure 4.37 shows that user can navigate to `/sys/fs/cgroup/memory/lxc/n1` this directory and check the value in bytes.

```
mahnaz@mahnaz-VB:~$ cd /sys/fs/cgroup/memory/lxc/n1/
mahnaz@mahnaz-VB:/sys/fs/cgroup/memory/lxc/n1$ cat memory.limit_in_bytes
1073741824
mahnaz@mahnaz-VB:/sys/fs/cgroup/memory/lxc/n1$
```

Figure 4.37 Screenshot of Memory Usage in Bytes

For the second aspect, user can define the maximum CPU usage in CORE GUI using again `set_cgroup_item()` method to enter the number of core user wants for each container. Then `get_cgroup_item()` returns the number of cores for the containers. Figure 4.38 shows that user has defined 2 core for n1 LXC container.

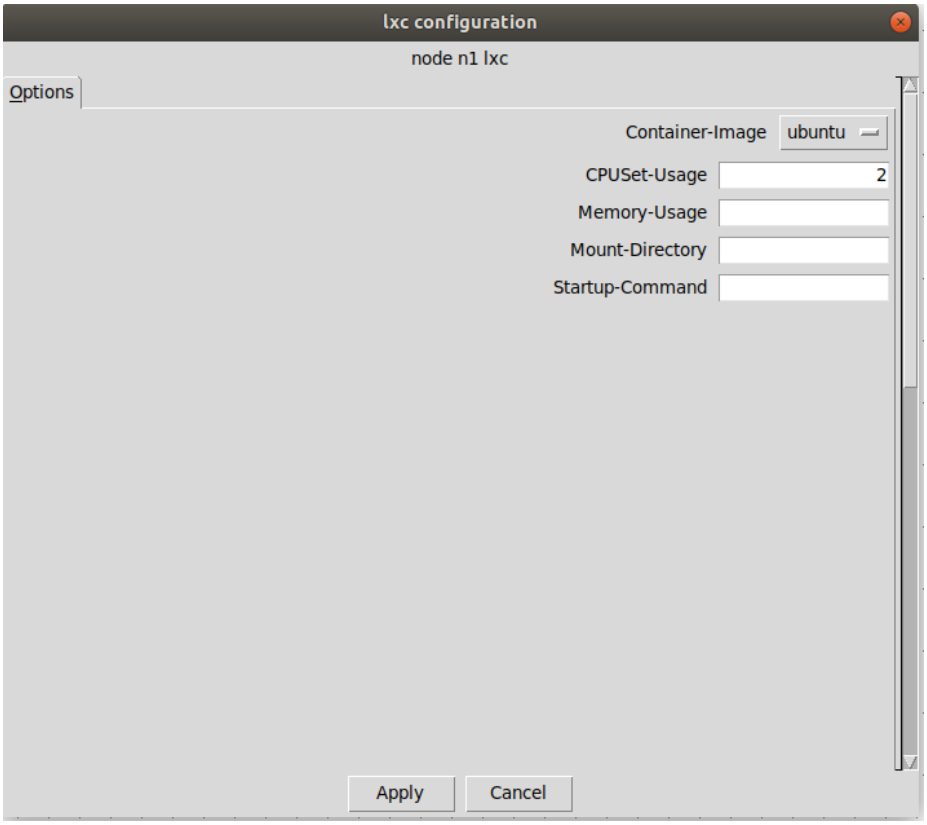

Figure 4.38 User-defined CPU Usage in CORE GUI

Figure 4.39 demostrates that user can navigate to `/sys/fs/cgroup/cpuset/lxc/n1` this directory and validate the result.

```
mahnaz@mahnaz-VB:~$ cd /sys/fs/cgroup/cpuset/lxc/n1/
mahnaz@mahnaz-VB:/sys/fs/cgroup/cpuset/lxc/n1$ cat cpuset.cpus
2
mahnaz@mahnaz-VB:/sys/fs/cgroup/cpuset/lxc/n1$
```

Figure 4.39 Screenshot of CPU Usage in CORE GUI

4.5.7 Network Implementation

In this section, there will be a discussion on the network setup for LXC node type to the corresponding CORE bridge network. At this moment, CORE uses Linux network namespace virtualization technique to create virtual nodes and connects them via Linux Ethernet Bridge network. Based on this concept, there is a possibility to setup the LXC node to network implementation.

In order to apply connection between CORE nodes, the parameters need to be defined to create the network interfaces for the container. A pair of virtual ethernet (veth) device will be specified for the implementation. Only one part of the virtual ethernet pair should be set inside the LXC container and the other one will be assigned to the corresponding CORE bridge network. Figure 4.40 depicts that one of the pair is attached into the container and the other side of the pair is attached to the bridge network that means it should be outside of the container.

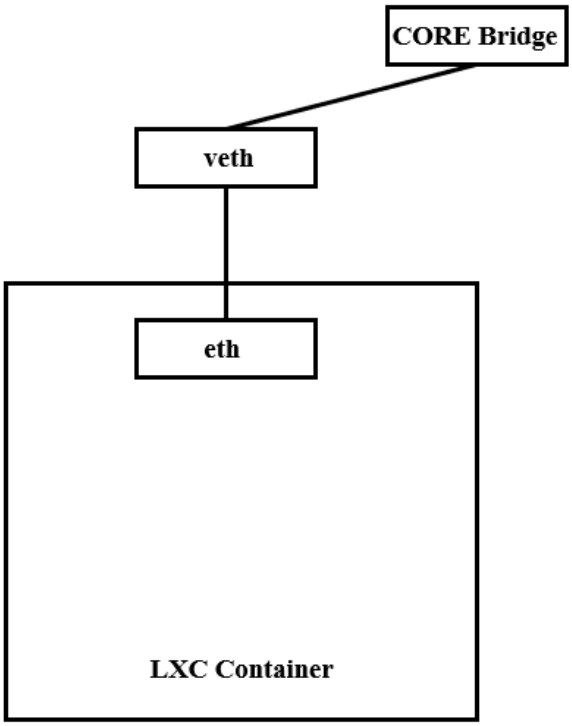

Figure 4.40 Virtual Ethernet Pair Assignment to CORE Bridge Network

In order to adapt the implementation of CORE emulator network for LXC node type implementation, Figure 4.41 shows the UML activity diagram of LXC node to network implementation in brief. While adding two nodes in CORE GUI, in the emulation the nodes are connecting with the CORE bridge network in the backend. To add a node to the network, the network interface will be created. After that, it will add one pair of virtual ethernet pair to the container ethernet interface. To do that, it will retrieve the container pid and set the virual ethernet device name to the container interface. Similarly, it will attach the remaining part to the corresponding CORE bridge network. Once this is done, then the interfaces will be up and ready for emulation. If the interface is up then it will return the interface information. User can query the information using "`ifconfig`" or "`ip address show`" these commands.

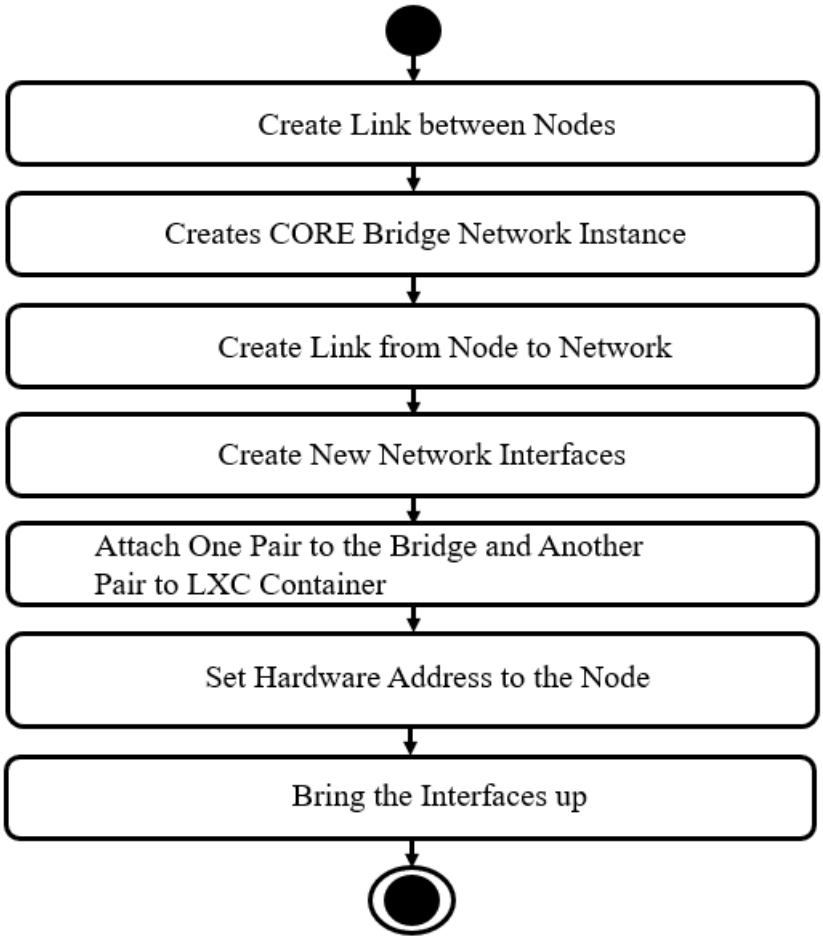

Figure 4.41 UML Activity Diagram of Network Implementation for LXC Node

Figure 4.42 presents the log file for LXC node to network implementation. After creating the link between nodes n1 and n2, it gets connected to the CORE bridge network. Here, n1 is the LXC node and n2 is the network namespace based CORE node. The bridge object ID is 27289. In every new CORE session, this ID will change.

File: /var/log/core-daemon.log

File: /var/log/core-daemon.log

```
2019-07-25 11:01:28,367 - INFO - service:add_services - adding service to node(n2): IPForward
2019-07-25 11:01:28,368 - INFO - coreemu:add_link - adding link for peer to peer nodes: n1 - n2
2019-07-25 11:01:28,414 - INFO - vnet:startup - bridge name triggered: b.27289.29
2019-07-25 11:01:28,414 - INFO - coreemu:add_link - adding link from node to network: n1 - 27289
2019-07-25 11:01:28,415 - INFO - reallxcnode:newnetif - newnetif triggered - net=<core.netns.nodes.
PtpNet object at 0x7fdf20334c50> - addrlist=['10.0.0.1/24', '2001::1/64'] - hwaddr=00:00:00:aa:00:0
0 - ifindex=0 - ifname=None
2019-07-25 11:01:28,415 - INFO - reallxcnode:newveth - newnetifindex triggered, ifindex = 0 - ifnam
e = None - net = <core.netns.nodes.PtpNet object at 0x7fdf20334c50>
2019-07-25 11:01:28,639 - INFO - coreemu:add_link - adding link from network to node: n2 - 27289
2019-07-25 11:01:28,640 - INFO - vnode:newnetif - newnetif triggered - net=<core.netns.nodes.PtpNet
 object at 0x7fdf20334c50> - addrlist=['10.0.0.2/24', '2001::2/64'] - hwaddr=00:00:00:aa:00:01 - if
index=0 - ifname=None
2019-07-25 11:01:28,640 - INFO - vnode:newveth - newnetifindex triggered, ifindex = 0 - ifname = No
ne - net = <core.netns.nodes.PtpNet object at 0x7fdf20334c50>
```

Figure 4.42 CORE Daemon Log for LXC Node to Network Implementation

Here, Figure 4.43 shows the available Ethernet Bridge and the network interfaces. From the previously shown log file, it is clear that the virtual ethernet interface pair (veth) is connected to the corresponding bridge network which is b.27289.29. the interface veth1.0.29 indicates the network device veth1 is connected to the LXC node n1 ethernet interface eth0 and the interface veth2.0.29 indicates that veth2 is connected to CORE node n2 ethernet interface eth0.

```
mahnaz@mahnaz-VB:~$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
b.27289.29              8000.3ac8a3fdfe92       no              veth1.0.29
                                                                veth2.0.29
lxcbr0          8000.00163e000000       no
```

Figure 4.43 Available CORE Bride network and Interfaces

To verify the connection is implemented correctly, Figure 4.44 illustrates that LXC node n1 is connected to CORE node n2 and n1 node can ping to n2 node.

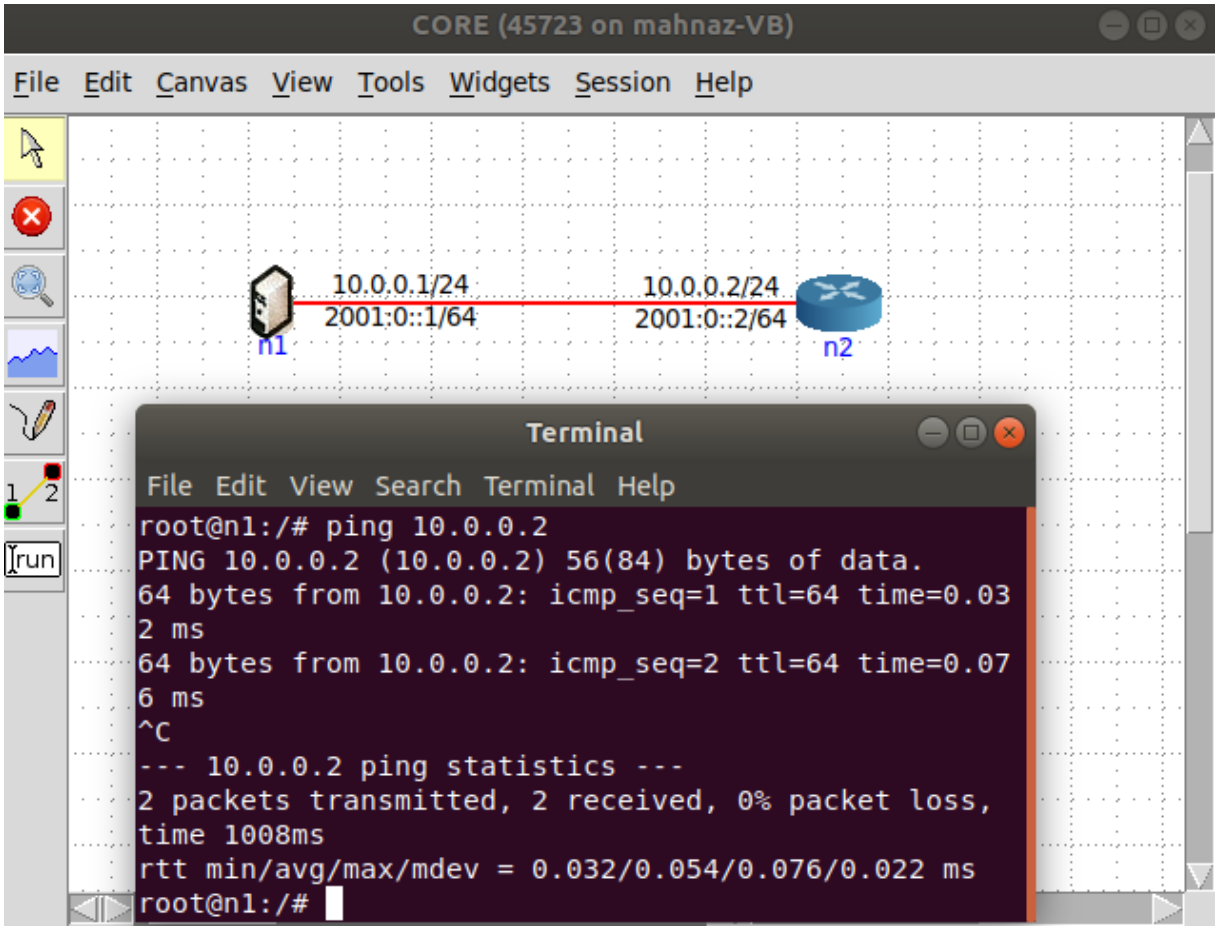

Figure 4.44 Verification of LXC Network Implementation

If LXC node is not connected to other LXC or CORE node then it should have only loopback interface. To do that, edit the `/etc/lxc/default.conf` file and specify the network type empty like this `lxc.net.0.type = empty`. This will make sure that the containers will not connect to the default lxcbr0 bridge.

Figure 4.45 displays the information about available network interface of n1 LXC container in the system.

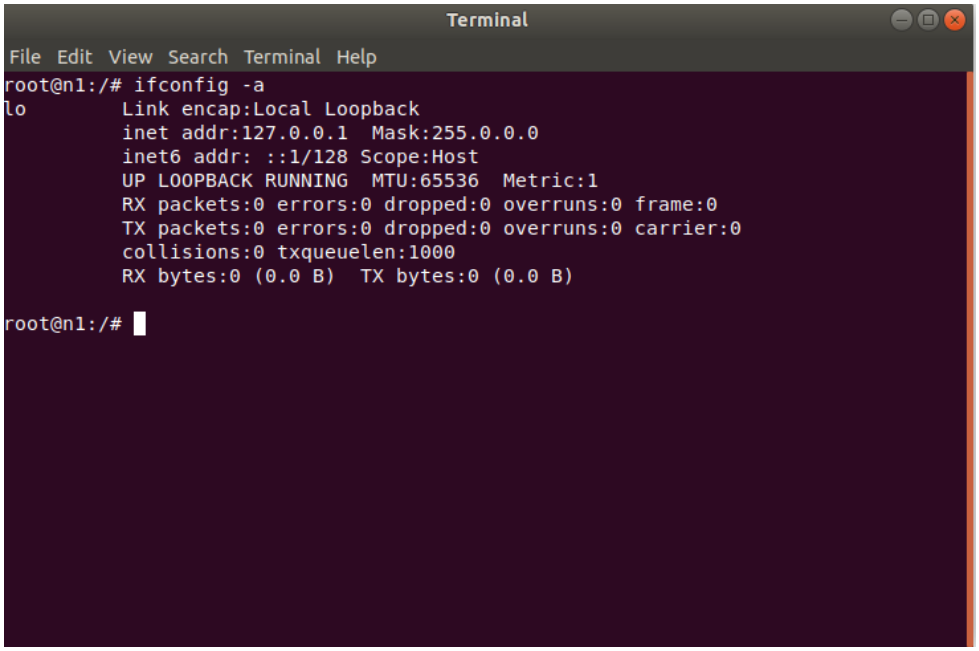

Figure 4.45 Screenshot of Viewing Network Interface

4.5.8 Enable Observer Widgets on LXC Node

Observer widget is a tool to display information about a node in the CORE network emulator. It enables the widgets by hovering over the node and it shows the specific information about the node.

CORE has developed a program called "`vcmd`" which connects `vnoded` daemon in a Linux network namespace and runs commands in the namespace. Current CORE implementation implements a separate class for CORE node named as "`vnodedClient`" for executing commands over a network namespace CORE node type. This implementation has two required arguments. They are namely as control channel and command line channel which is used to pass the commands to the running `vnoded` process within a network namespace. By calling `vcmd` python modules, the control channel of the `vnodedClient` can be accessed. But, in LXC node type implementation it is not possible to extend the functionality of CORE node widget implementation due to the incompatibility issue. LXC does not support vcmd program and therefore, it cannot be used to implement Observer Widget tool to LXC node. Hence, theoretically it is possible to execute command inside the container and then fetch the result to display.

CORE has implemented the method `cmd_output()` to execute command on CORE node which can be adapted to implement observer widget for LXC node. The method `cmd_output()` is used to run commands on the CORE node and return a tuple containing exit status and the output string. Similarly, widget implementation for LXC node, the same method has given the support to achieve the same functionality as CORE node. In this regard, a string type variable has been declared to hold the commands. Then the commands need to be split out into smaller chunks so that no whitespace will be counted by default. Another list of arrays has been created to attach the command to the LXC Container. After that, two array-list have been merged and has been executed using `subprocess` python module. This method will return the status and the output result. Figure 4.46 shows the UML activity diagram of the process step by step.

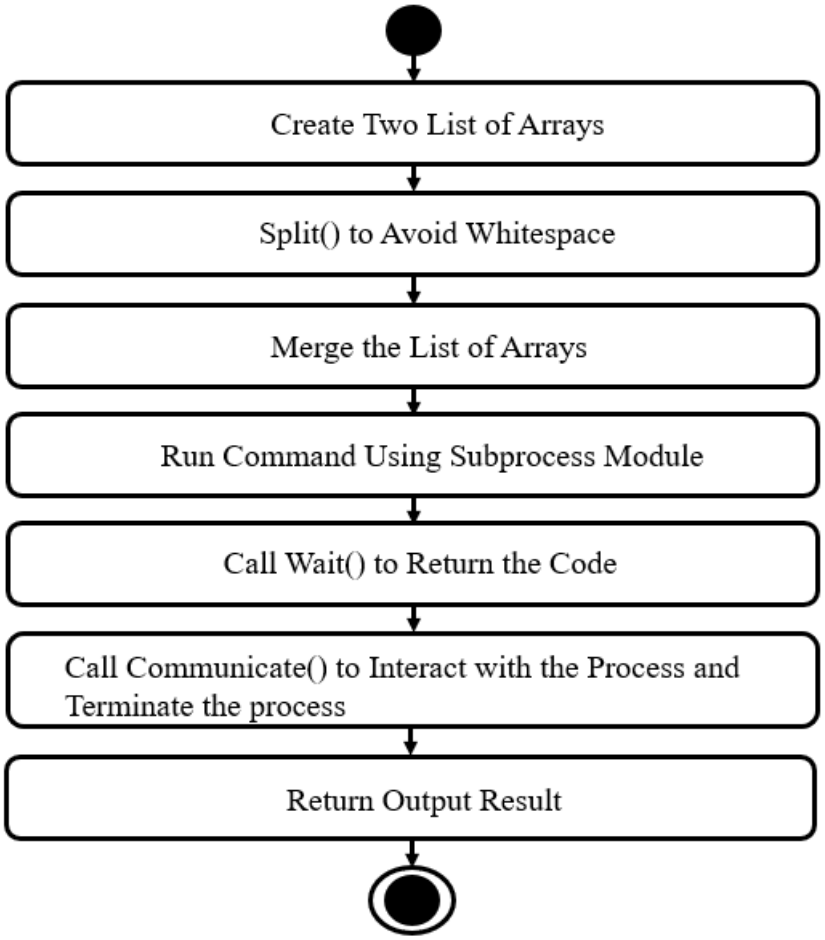

Figure 4.46 UML Activity Diagram for Observer Widget Implementation

4.5.9 Enable Popup Shell Window on LXC Node

During a running network emulation, `vcmd` program offers to open a terminal for a CORE node. Double-click on a node opens the shell window by invoking the `vcmd` shell command such as:

```
gnome-terminal -e vcmd -c /tmp/pycore.50160/n1 -- bash
```

Here, 50160 indicates the session ID and n1 is the node name. As discussed before in sub-section 4.5.8 that `vcmd` does not provide support for LXC node, so implementation for LXC node will take different approach.

To extend the functionality of opening shell window on LXC node, `termcmdstring()` method will be invoked and adapted to the LXC node type implementation. This method will create a terminal command string and attach a shell to execute command in it. The command which will open the shell window for LXC node is given below.

```
gnome-terminal -x lxc-attach -n self.name
```

here, self.name is the variable for LXC node type.

4.5.10 Mount Directory from Host OS and Exploring the Filesystem

The root filesystem of LXC containers are accessible from host operating system as a regular directory tree. LXC offers the possibility of manipulating the files in an already running container by a simple change in the directory.

LXC also allows mounting directory from host machine inside container using bind mount. A bind mount is a different view of the directory tree which can be obtained by replicating the existing directory tree under a different mount point.

To implement this use-case, Figure 4.47 shows that, at first to add this specific configuration aspect to the LXC node, open the configuration option window and enter a directory path in the "Mount-directory" field in CORE GUI. This directory can be any directory location from the host system. User can attach any directory which is desired to examine during container running state. After adding the option for configuration then, the session will be started in CORE GUI. The container will be created with specified template image. After creating the container, it will check the condition in CORE Daemon that whether the "Mount-directory" configuration option is filled or empty. If the option contains any directory location, then the configuration option called "`lxc.mount.entry`" has been added to the configuration file of the newly created container. This will tell LXC to target which directory should be bind mount from the host machine and the mount point inside the container filesystem to bind to. After that, the container will be started. On the contrary, if the "Mount-directory" option is empty then it will directly start the container after creation. Hence, it will not able to attach any directory from host system to LXC container.

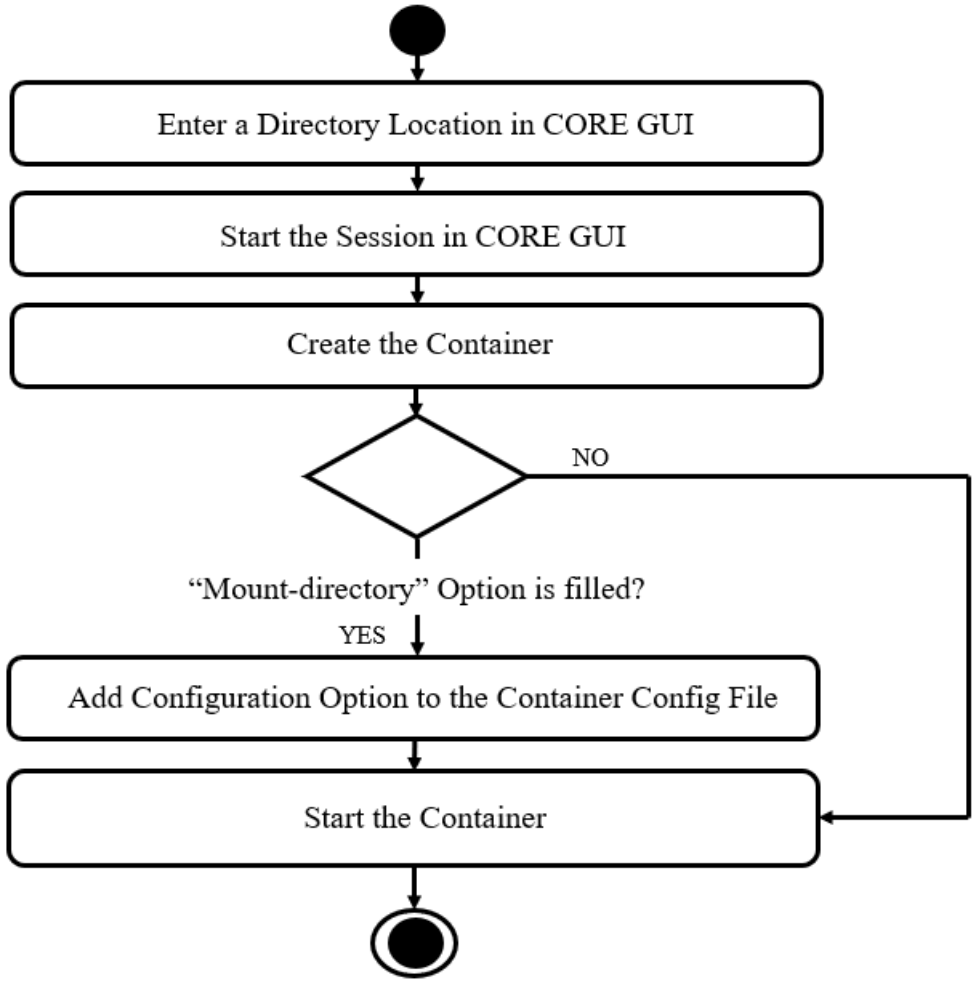

Figure 4.47 UML Activity Diagram of Mount Directory from Host Machine to LXC Container

The configuration file of a container can be found under "`/var/lib/lxc/container_name/config`" this path. The mount point inside the container is "`/mnt`". Figure 4.48 presents that the configuration option has been

added to the config file of n1 LXC container. It is also showing that the directory "`/home/mahnaz/hook`" has been attached from the host machine which is entered by the user in CORE GUI.

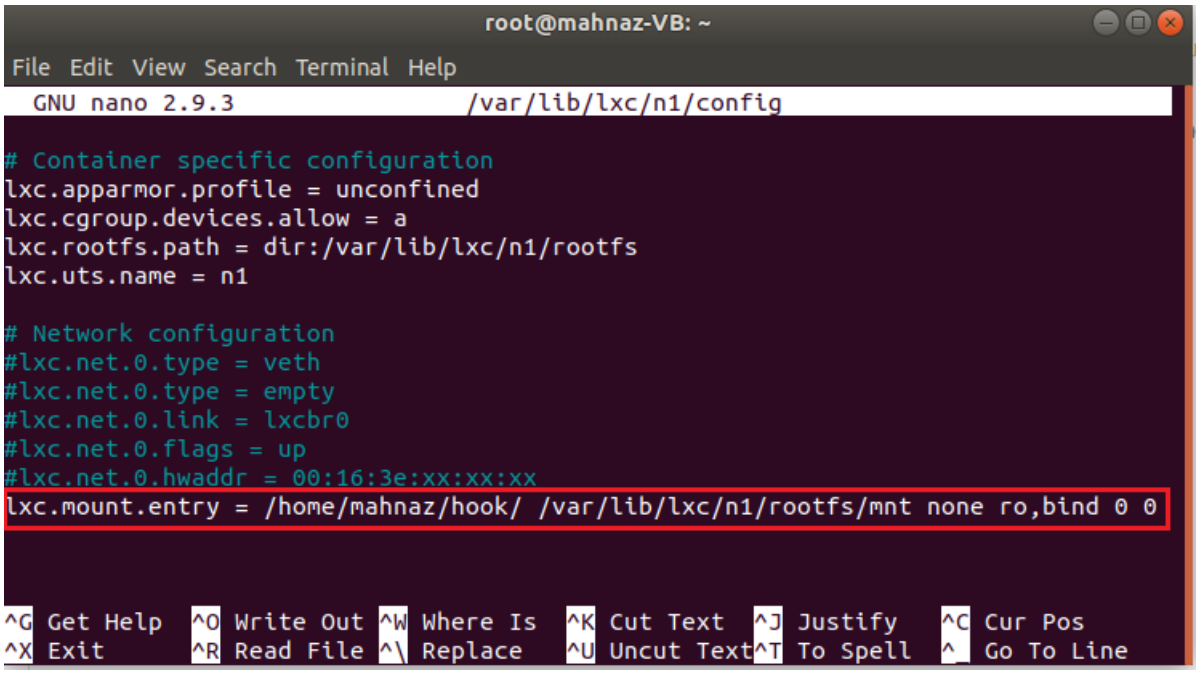

Figure 4.48 Addition of Configuration Option of n1 LXC Container

Before starting the container, these steps need to be performed. Once the container has been created, with this configuration the container will be started.

Figure 4.49 presents that once the container has started, it is possible to enter inside the container and observe under /mnt folder as the user-defined directory is mounted into the container. As the directory is mounted to the container so now it is possible to view the content of the folder.

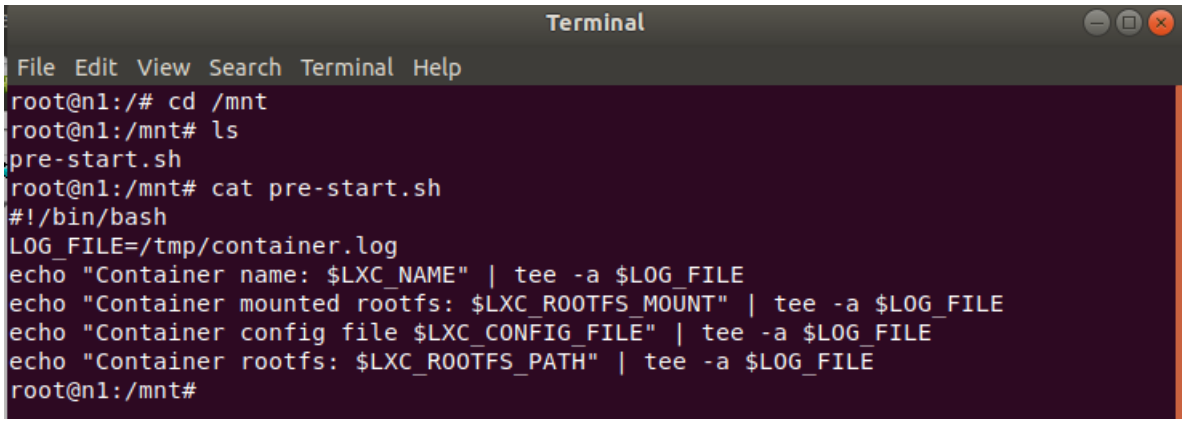

Figure 4.49 File Mounted Inside Container

In a running container it is possible to access into some files from the /proc directory in the host machine. To explore the running directory of a container, it is required to obtain the container PID. Figure 4.50 portrays the information of n1 container and collect the PID of the container.

```
root@mahnaz-VB:~# lxc-info -n n1
Name:           n1
State:          RUNNING
PID:            3040
IP:             172.17.0.1
CPU use:        11.04 seconds
BlkIO use:      12.00 KiB
Memory use:     72.80 MiB
KMem use:       10.22 MiB
Link:           lxcbr0
 TX bytes:      0 bytes
 RX bytes:      0 bytes
 Total bytes:   0 bytes
root@mahnaz-VB:~#
```

Figure 4.50 Retrieve Information of Specific Container

Figure 4.51 shows that after collecting the PID of the container it is possible to explore the running directory of the container. The command for examining the running directory of the container is given below

```
$ ls -la /proc/3040/root/run
```

Here, the PID of the container will be changing at each time new container creation.

```
root@mahnaz-VB:~# ls -la /proc/3040/root/run
total 20
drwxr-xr-x 14 root root    480 Jul  5 12:01 .
drwxr-xr-x 21 root root   4096 Jul  5 12:01 ..
-rw-------  1 root root      0 Jul  5 12:01 agetty.reload
drwxr-xr-x  2 root root     60 Jul  5 12:01 console-setup
drwx--x--x  4 root root    100 Jul  5 12:01 containerd
-rw-r--r--  1 root root      3 Jul  5 12:01 crond.pid
----------  1 root root      0 Jul  5 12:01 crond.reboot
drwxr-xr-x  2 root root     60 Jul  5 12:01 dbus
drwx------  5 root root    120 Jul  5 12:01 docker
-rw-r--r--  1 root root      2 Jul  5 12:01 docker.pid
srw-rw----  1 root netdev    0 Jul  5 12:01 docker.sock
lrwxrwxrwx  1 root root     25 Jul  5 12:01 initctl -> /run/systemd/initctl/fifo
drwxrwxrwt  3 root root     60 Jul  5 12:01 lock
drwxr-xr-x  2 root root     40 Jul  5 12:01 log
drwxr-xr-x  2 root root     60 Jul  5 12:01 mount
drwxr-xr-x  3 root root     60 Jul  5 12:01 NetworkManager
-rw-r--r--  1 root root      2 Jul  5 12:01 rsyslogd.pid
drwxr-xr-x  2 root root     40 Jul  5 12:01 sendsigs.omit.d
lrwxrwxrwx  1 root root      8 Jul  5 12:01 shm -> /dev/shm
drwx--x--x  3 root root     60 Jul  5 12:01 sudo
drwxr-xr-x 18 root root    440 Jul  5 12:01 systemd
drwxr-xr-x  2 root root     40 Jul  5 12:01 user
-rw-rw-r--  1 root utmp   2688 Jul  5 12:01 utmp
-rw-------  1 root root      0 Jul  5 12:01 xtables.lock
root@mahnaz-VB:~#
```

Figure 4.51 Running Directory of a Container

4.5.11 Implement Start-up Script

In this task, the LXC configuration manager provides a user-configurable field called “Start-up Command” where user can be able to give input of a command. This command will be executed inside the container after it has been created and started. Basically, this option offers the possibility to observe a file creation during the LXC’s lifetime.

Figure 4.52 presents the configuration option for start-up command. In this field, user gets the possibility to enter a command “`touch file`” which creates a file called “file”. This is an example command which has been executed inside container.

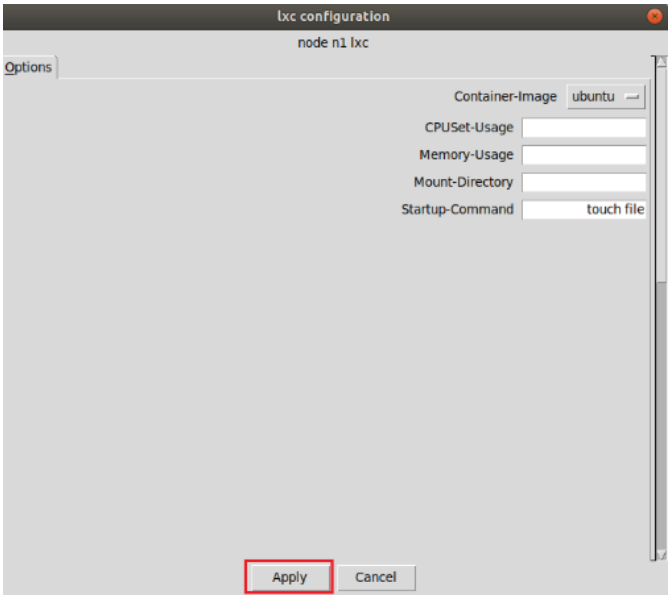

Figure 4.52 Configuration Option for Startup Command

Once the button “Apply” is pressed, user starts the session and observes that a file has been created inside the container. Figure 4.53 indicates that user-defined startup-command has been executed correctly.

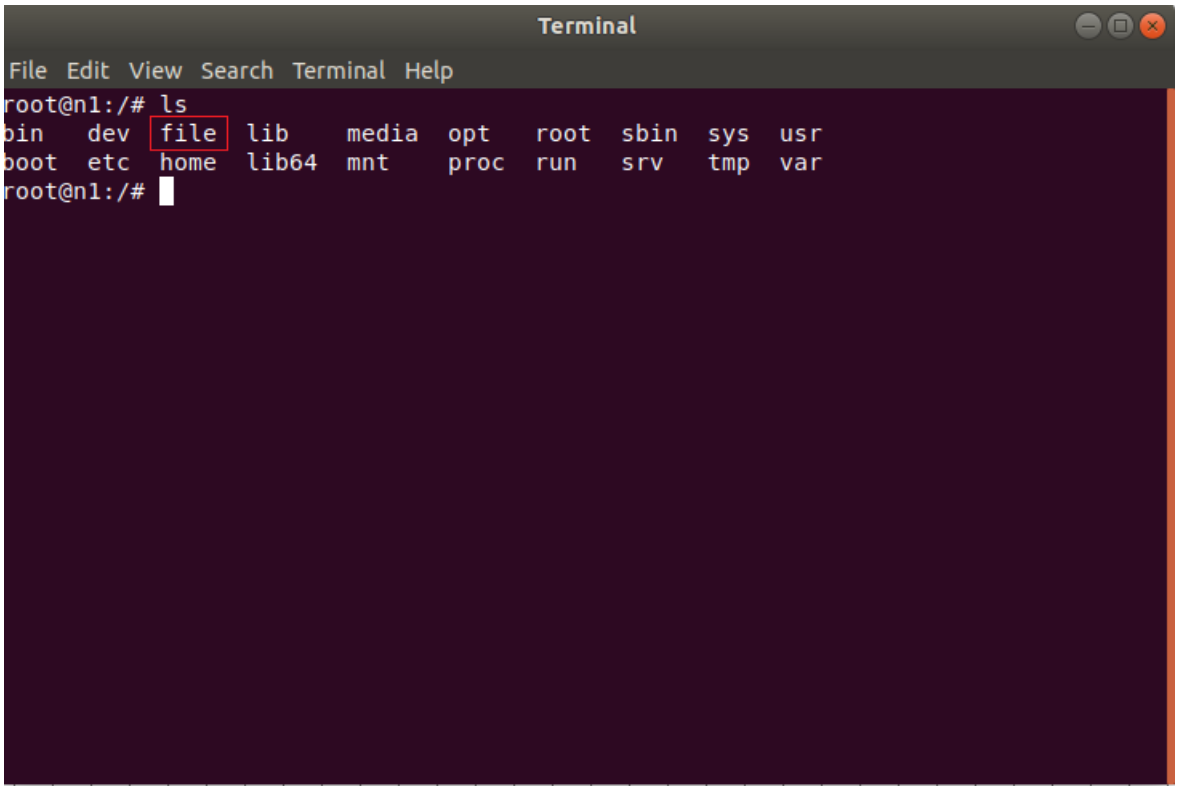

Figure 4.53 File Creation inside LXC Container

4.5.12 LXC Security Features

Security is one of the major concerns in terms of using container. LXC supports various technologies to alleviate many security concerns. Capabilities is a security feature which has been included into LXC container to drop a capability in the container. This feature is useful when the container can be vulnerable due to not having root permission. So, `lxc.cap.drop` is a configurable option which specifies the capabilities to be dropped before staring of the container to do permission check.

LXC-python bindings provide the possibility to apply this configurable option to the LXC node. The `get_config_item()` and `set_config_item()` methods are required to drop the capabilities to a container. These methods take two arguments. One is the configuration option entry "`lxc.cap.drop`" and the other one is the list of value which means the number of capabilities the user wants to drop in a container. If no value has been given that means LXC will clear out any specified dropped capability in the container. Figure 4.54 shows all the capabilities dropped in a specific container.

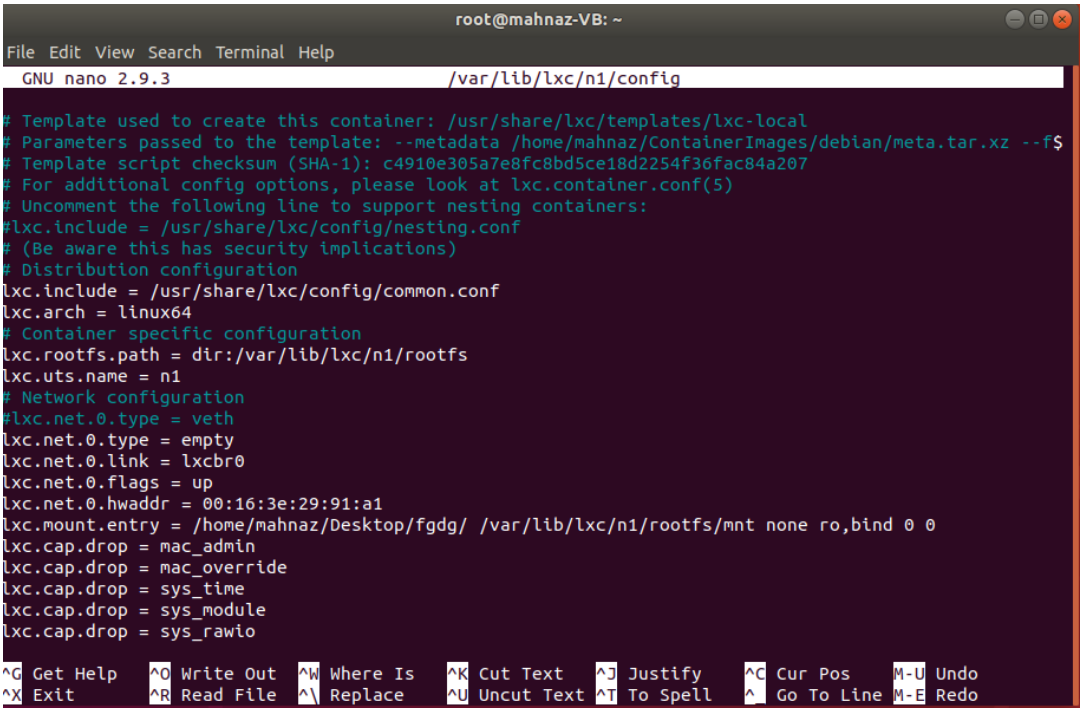

Figure 4.54 Screenshot of LXC Capability Drop

Here, a list of capabilities are dropped in a container. The first capability `mac_admin` allows MAC configuration or state changes. Then `mac_override` to override MAC. The next is, `sys_time` to set the system clock. After that, `sys_module` to load and unload kernel modules and the last one `sys_rawio` to perform I/O operation.

4.6 Evaluation of LXC Container Integration into CORE

The goal of this chapter is to demonstrate and validate the overall workflow of LXC node into CORE emulator. In this regard, Figure 4.55 illustrates a network topology as a test basis. Here, n1 and n3 are two LXC node type and n2 is a network namespace based CORE node. They are linked together via CORE bridge network.

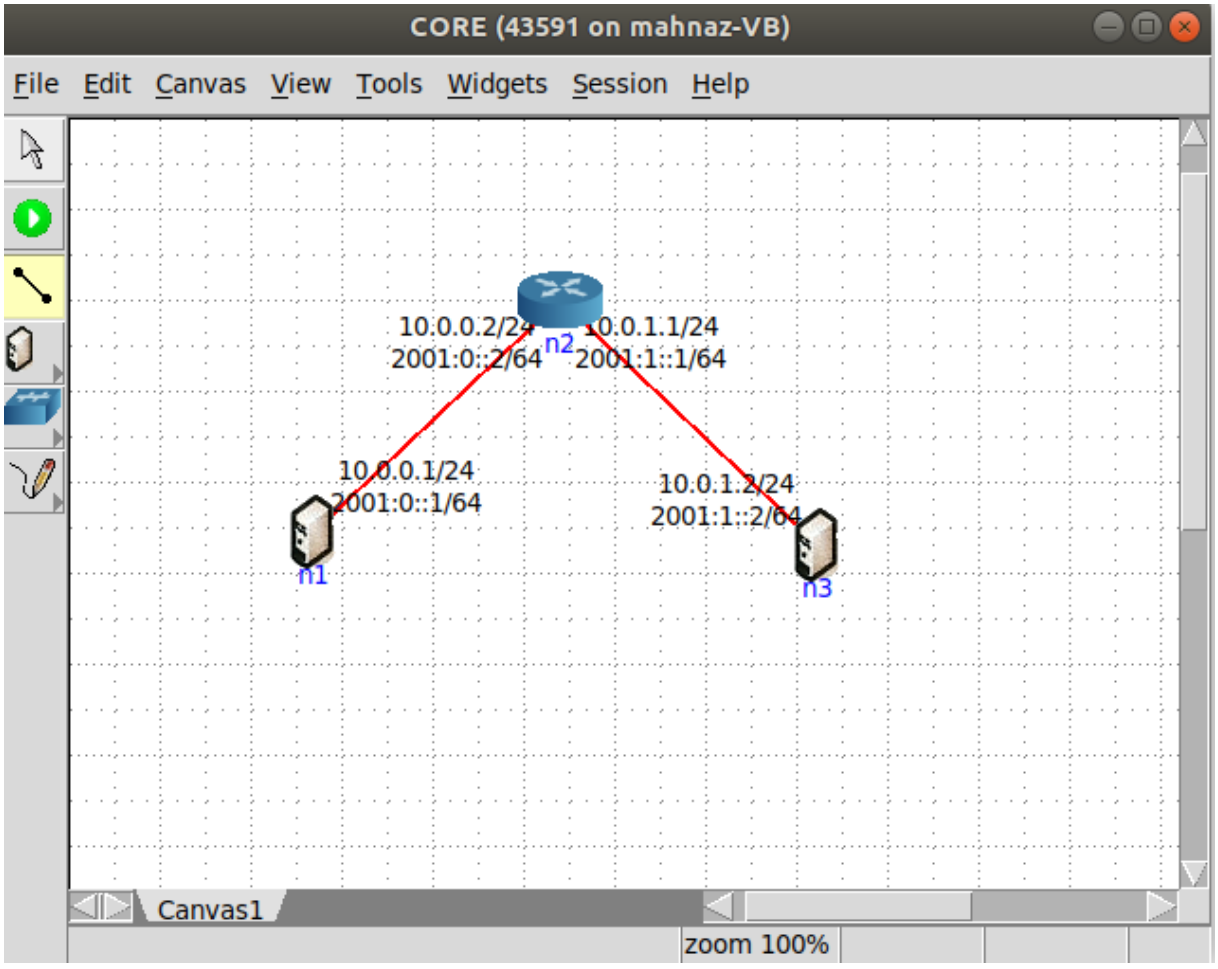

Figure 4.55 Example Network Scenario for LXC Implementation

4.6.1 LXC Node Configuration

After adding the nodes to the CORE canvas, now these nodes need to be configured. In this regard, the LXC nodes has been configured with two different configurations. To configure a node, double-click on the node, it will open a popup configuration box. Then, in the configuration box, click on the “Configure” button. This will open a configuration option window to specify the configuration parameters for the LXC nodes.

Figure 4.56 depicts the parameters for LXC node n1 configuration. Here, the LXC template image has been selected as ubuntu, CPU usage has been defined to 1 core and RAM usage has been specified as 2 Gigabytes. Moreover, the directory /home/mahnaz/hook/ from host system has been chosen to attach to the container. Lastly, there is a startup-command called touch x which will create a file named "x" inside the container after the container creation. Now, to enable the configurable parameters for LXC, click on “Apply”.

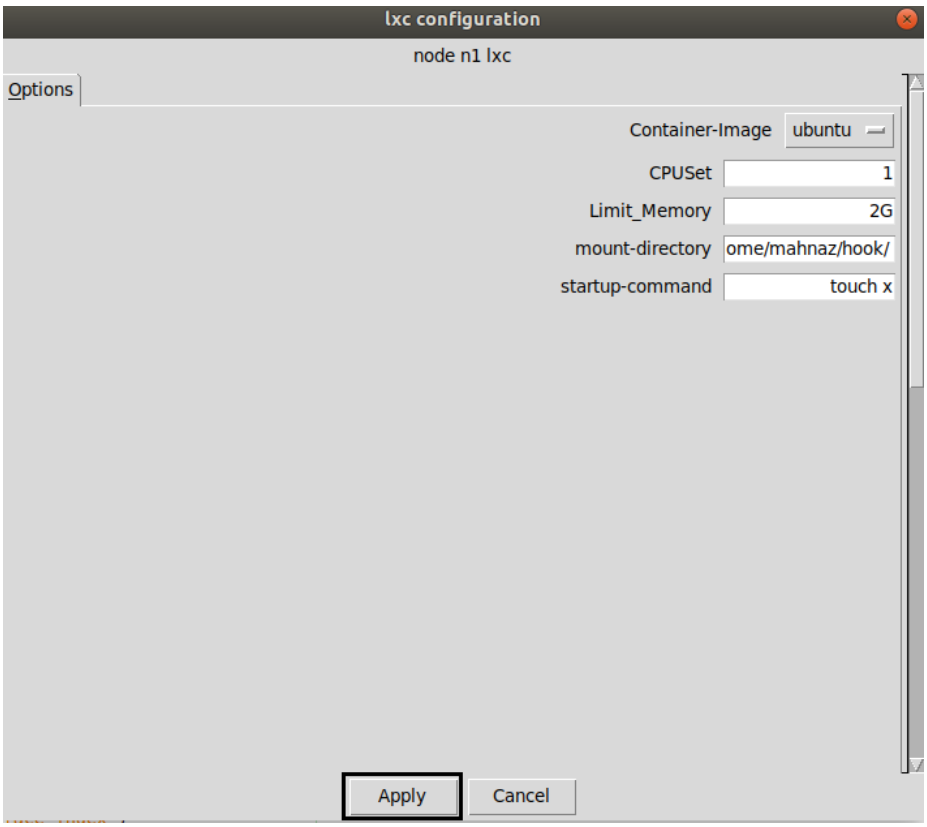

Figure 4.56 LXC Node n1 Configuration

Now, to configure the second LXC node n3, Figure 4.57 shows the configration. Here, container image templae is as same as n1 node, CPU usage has been defined as 2 core and RAM usage has been defined as 4 Gigabytes. In the “mount-directory” field, the direction location has been and “startup-command” field is also filled. After entering all data, click “Apply”.

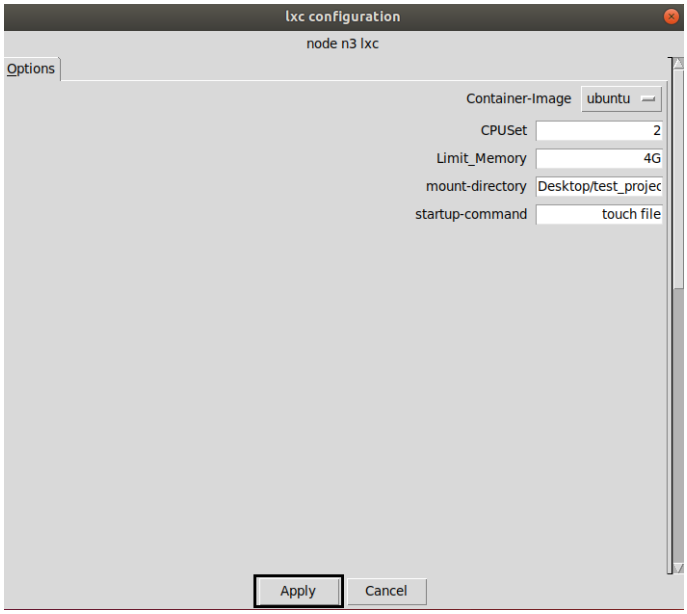

Figure 4.57 LXC Node n3 Configuration

4.6.2 CORE API Message Flow for Configuration

When the “Configure” button is pressed to proceed for configuration, it will send a request for configuration and get configuration parameters as a response when user defines the configuration in the GUI.

The overall message flow for configuration has been presented in Figure 4.58 using CORE API message. The first “CONF” message is for n1 LXC node. To distinguish the “CONF” messages for n1 and n3 node, the figure is marked with different colours. The “CONF” request and reply message for n1 node are shown in red and green colour. So, for LXC node type, a request has been sent for configuration. The next message contains all the configuration entries which are retrieved from the LXC configuration manager. Then, user has entered the required data in the configuration fields and click “Apply”. As a result, the “Apply Clicked” message is popping to confirm that the configuration is done, and it has received the parameters which is displayed with green colour.

The second “CONF” message is for n3 LXC node and the request-response message for n3 node configuration are presented in orange and yellow coloured rectangular box. This message flow is also similar to “CONF” message for n1 node.

```
root@mahnaz-VB:~# core-gui
Connecting to "core-daemon" (127.0.0.1:4038)...connected.
>CONF(flags=0,node=1,obj=lxc,cflags=0x1) request
CONF(flags=0,node=1/obj=lxc,cflags=0,types=10/10/10/10/10/,vals=Container-Image= |C
PUSet=|Limit_Memory=|mount-directory=|startup-command=,capt=Container-Image|CPUSet|
Limit_Memory|mount-directory|startup-command,pvals=debian,ubuntu||||,groups=Options
:1-5,)
Apply Clicked!
node: n1
model: lxc
types: 10 10 10 10 10
vals: Container-Image=ubuntu CPUSet=1 Limit_Memory=2G mount-directory=/home/mahnaz/
hook/ {startup-command=touch x}
>CONF(flags=0,node=1,obj=lxc,cflags=0types=<10 10 10 10 10>,values=<Container-Image
=ubuntu CPUSet=1 Limit_Memory=2G mount-directory=/home/mahnaz/hook/ {startup-comman
d=touch x}>) reply
>CONF(flags=0,node=3,obj=lxc,cflags=0x1) request
CONF(flags=0,node=3/obj=lxc,cflags=0,types=10/10/10/10/10/,vals=Container-Image= |C
PUSet=|Limit_Memory=|mount-directory=|startup-command=,capt=Container-Image|CPUSet|
Limit_Memory|mount-directory|startup-command,pvals=debian,ubuntu||||,groups=Options
:1-5,)
Apply Clicked!
node: n3
model: lxc
types: 10 10 10 10 10
vals: Container-Image=ubuntu CPUSet=2 Limit_Memory=4G mount-directory=/home/Desktop
/test_project/ {startup-command=touch file}
>CONF(flags=0,node=3,obj=lxc,cflags=0types=<10 10 10 10 10>,values=<Container-Image
=ubuntu CPUSet=2 Limit_Memory=4G mount-directory=/home/Desktop/test_project/ {start
up-command=touch file}>) reply
```

Figure 4.58 CORE API Message Flow for LXC Configuration

4.6.3 LXC Node Creation

The configuration for the LXC nodes is done, now the next step is, to initiate the session by pressing the “Start” button. When the “Start” button is pressed, it will initialise the LXC nodes and it will take some time to unpack the root filesystem of the container and then the container will be created. The container image has been prepared with distrobuilder tool, which has been explained at the beginning of the “Realisation” part.

The LXC node n1 and CORE node n2 creation has been outlined in Figure 4.59. As soon as the “Start” button is pressed, CORE shows a message that the LXC node type is found. Along with the configuration parameters, the LXC node type gets initialised, created and started the LXC container into CORE. To identify the LXC node, the red coloured box is portraying the LXC node type (0x46). Type = 0x0 defines the default network namespace CORE node. The “NODE” message for LXC includes the add flag, node type, name, object type, position and opaque data which contains the configuration information. On the other hand, CORE “NODE” messages contain add flag, name, type, object type and position.

```
found lxc type
cfg: {custom-config-id lxc} {custom-command {10 10 10 10 10}} {config {Container-Im
age=ubuntu CPUSet=1 Limit_Memory=2G mount-directory=/home/mahnaz/hook/ {startup-com
mand=touch x}}}
config: Container-Image=ubuntu CPUSet=1 Limit_Memory=2G mount-directory=/home/mahna
z/hook/ {startup-command=touch x}
json: {"Container-Image": "ubuntu"
,"CPUSet": "1"
,"Limit_Memory": "2G"
,"mount-directory": "/home/mahnaz/hook/"
,"startup-command": "touch x"
}
>NODE(flags=add/str,n1,type=0x46,n1,m=lxc,x=129,y=218,opaque={"Container-Image": "u
buntu"
,"CPUSet": "1"
,"Limit_Memory": "2G"
,"mount-directory": "/home/mahnaz/hook/"
,"startup-command": "touch x"
})
>NODE(flags=add/str,n2,type=0x0,n2,m=router,x=259,y=96)
```

Figure 4.59 LXC node n1 and CORE Node n2 Creation Showing Via CORE API Message

For further verification, user can check the CORE Daemon log message. Here, only one container which is n1 is showing in Figure 4.60.

File: /var/log/core-daemon.log

File: /var/log/core-daemon.log

```
2019-07-25 16:54:06,645 - INFO - coreemu:add_node - creating node(RealLxcNodeType) id(1) n
ame(n1) start(True)
2019-07-25 16:54:06,645 - INFO - reallxcnode:__init__ - Init triggered!
2019-07-25 16:54:06,645 - INFO - reallxcnode:__init__ - opaque = None
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - initialiseLXC triggered!
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - opaque = {"Container-Image":
"ubuntu"
,"CPUSet": "1"
,"Limit_Memory": "2G"
,"mount-directory": "/home/mahnaz/hook/"
,"startup-command": "touch x"
}
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - session = <core.emulator.core
emu.EmuSession object at 0x7fe157469e50>
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - name = n1
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - objid = 1
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - retrieved container-image: ub
untu
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - retrieved cpu: 1
2019-07-25 16:54:06,646 - INFO - reallxcnode:initialiseLXC - retrieved ram usage: 2G
Unpacking the rootfs

---
You just created an Ubuntu bionic amd64 (20190725_1340) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
2019-07-25 16:54:26,944 - INFO - reallxcnode:initialiseLXC - container created - n1
2019-07-25 16:54:26,985 - INFO - reallxcnode:initialiseLXC - container started - n1
```

Figure 4.60 CORE Daemon Log File to Validate LXC Container Creation

Figure 4.61 shows the second LXC node n3 creation. This “NODE” message contains the same message except the different configuration parameters.

```
found lxc type
cfg: {custom-config-id lxc} {custom-command {10 10 10 10 10}} {config {Container-Im
age=ubuntu CPUSet=2 Limit_Memory=4G mount-directory=/home/Desktop/test_project/ {st
artup-command=touch file}}}
config: Container-Image=ubuntu CPUSet=2 Limit_Memory=4G mount-directory=/home/Deskt
op/test_project/ {startup-command=touch file}
json: {"Container-Image": "ubuntu"
,"CPUSet": "2"
,"Limit_Memory": "4G"
,"mount-directory": "/home/Desktop/test_project/"
,"startup-command": "touch file"
}
>NODE(flags=add/str,n3,type=0x46 n3,m=lxc,x=395,y=225,opaque={"Container-Image": "u
buntu"
,"CPUSet": "2"
,"Limit_Memory": "4G"
,"mount-directory": "/home/Desktop/test_project/"
,"startup-command": "touch file"
})
```

Figure 4.61 LXC node n3 Creation Showing Via CORE API Message

The next log file Figure 4.62 shows the LXC node n3 has created and started inside CORE Daemon with different configuration parameters.

File: /var/log/core-daemon.log

File: /var/log/core-daemon.log

```
2019-07-25 16:54:27,032 - INFO - coreemu:add_node - creating node(RealLxcNodeType) id(3) n
ame(n3) start(True)
2019-07-25 16:54:27,032 - INFO - reallxcnode:__init__ - Init triggered!
2019-07-25 16:54:27,032 - INFO - reallxcnode:__init__ - opaque = None
2019-07-25 16:54:27,032 - INFO - reallxcnode:initialiseLXC - initialiseLXC triggered!
2019-07-25 16:54:27,032 - INFO - reallxcnode:initialiseLXC - opaque = {"Container-Image":
"ubuntu"
,"CPUSet": "2"
,"Limit_Memory": "4G"
,"mount-directory": "/home/Desktop/test_project/"
,"startup-command": "touch file"
}
2019-07-25 16:54:27,033 - INFO - reallxcnode:initialiseLXC - session = <core.emulator.core
emu.EmuSession object at 0x7fe157469e50>
2019-07-25 16:54:27,033 - INFO - reallxcnode:initialiseLXC - name = n3
2019-07-25 16:54:27,033 - INFO - reallxcnode:initialiseLXC - objid = 3
2019-07-25 16:54:27,033 - INFO - reallxcnode:initialiseLXC - retrieved container-image: ub
untu
2019-07-25 16:54:27,033 - INFO - reallxcnode:initialiseLXC - retrieved cpu: 2
2019-07-25 16:54:27,033 - INFO - reallxcnode:initialiseLXC - retrieved ram usage: 4G
Unpacking the rootfs

---
You just created an Ubuntu bionic amd64 (20190725_1340) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
2019-07-25 16:54:48,600 - INFO - reallxcnode:initialiseLXC - container created - n3
2019-07-25 16:54:48,649 - INFO - reallxcnode:initialiseLXC - container started - n3
```

Figure 4.62 CORE Daemon Log for n3 LXC Container Creation

It is possible to check whether the LXC container is created or not using the following command

```
$ sudo lxc-ls --fancy
```

Figure 4.63 presents the list of containers and it is visible that the LXC node n1 and n3 has created and already in running state.

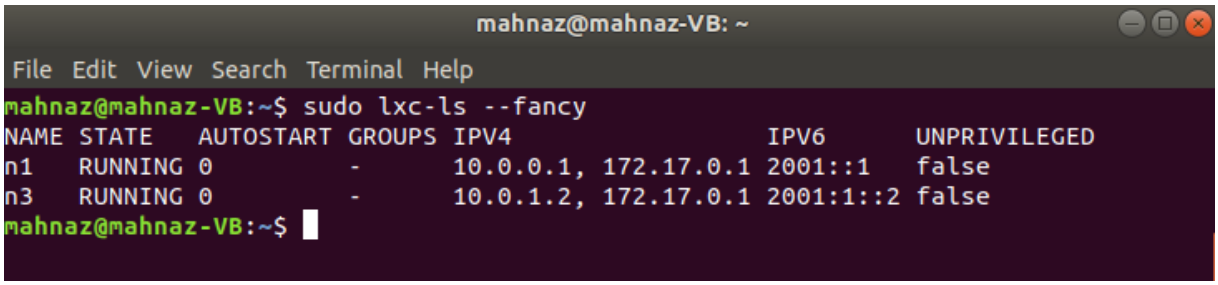

Figure 4.63 List of Containers

4.6.4 Network Implementation

As it is observed from Figure 4.55 that n1 and n3 two LXC nodes are connected to CORE node n2. When the nodes are connected to each other, they connect through Bridge network. In Figure 4.64, it is seen that the first "LINK" message is the connection between n1 and n2. It also contains the interfaces eth0 for n1 node and eth0 for n2 node as well. The second "LINK" message signals the link between n2 and n3 and the interfaces eth1 for n2 and eth0 for n3. After creation of the nodes and links, then the session reached to instantiation state.

```
>LINK(flags=1,1-2,0,0,0,0,0,type=1,if1n=0,10.0.0.1/24,2001:0::1/64,00:00:00:aa:00:0
0,if2n=0,10.0.0.2/24,2001:0::2/64,00:00:00:aa:00:01,)
>LINK(flags=1,2-3,0,0,0,0,0,type=1,if1n=1,10.0.1.1/24,2001:1::1/64,00:00:00:aa:00:0
2,if2n=0,10.0.1.2/24,2001:1::2/64,00:00:00:aa:00:03,)
>CONF(flags=0,obj=metadata,cflags=0types=<10 >,values=<{canvas c1={name {Canvas1}}}
>) reply
>CONF(flags=0,obj=metadata,cflags=0types=<10>,values=<{global_options=interface_nam
es=no ip_addresses=yes ipv6_addresses=yes node_labels=yes link_labels=yes show_api=
yes background_images=no annotations=yes grid=yes traffic_start=0}>) reply
>EVENT(flags=0,type=3-instantiation_state,)
EVENT(flags=0,type=15,)
EVENT(flags=0,type=15,)
EVENT(flags=0,type=4-runtime_state,time=1564066489.88,)
NODE(flags=9,num=1,emuid=1,)
NODE(flags=9,num=2,emuid=2,)
NODE(flags=9,num=3,emuid=3,)
```

Figure 4.64 Link Creation Showing Through CORE API Message

To check the available bridge network, Figure 4.65 is outlined. Node n1 is connected to n2 via the bridge b.35683.ed and n2 and n3 have made connection with bridge b.16381.ed. The virtual ethernet pair veth1 is linked to eth0 of n1 node and veth2 is linked to eth0 of n2 CORE node. Then veth2 is connected to eth1 of n2 node while, veth3 is connected to eth0 of n3 LXC node.

```
mahnaz@mahnaz-VB:~$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
b.16381.ed              8000.3211041d5774       no              veth2.1.ed
                                                        veth3.0.ed
b.35683.ed              8000.4e9cb1a194e0       no              veth1.0.ed
                                                        veth2.0.ed
lxcbr0          8000.00163e000000       no
```

Figure 4.65 Showing Available Bridge Network and Interfaces

To verify the connectivity between the nodes, ping is the best way to use. Here, in Figure 4.66 it is clear that, n1 node can ping to n2 node and n2 node can ping to n3 node. In Figure 4.55, the node's IP addresses are given.

```
                                    Terminal
 File  Edit  View  Search  Terminal  Help
root@n1:/# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.070 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.073 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.034 ms
^C
--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2051ms
rtt min/avg/max/mdev = 0.034/0.059/0.073/0.017 ms
                              root@n2: /tmp/pycore.43591/n2.conf
 File  Edit  View  Search  Terminal  Help
root@n2:/tmp/pycore.43591/n2.conf# ping 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=0.055 ms
64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=0.079 ms
^C
--- 10.0.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2050ms
rtt min/avg/max/mdev = 0.055/0.070/0.079/0.014 ms
root@n2:/tmp/pycore.43591/n2.conf#
```

Figure 4.66 Verification of LXC Node's Network Implementation

4.6.5 Limit CPU Usage

The usage for CPU in LXC node n1, has been defined as 1 core in the configuration box (see Figure 4.56). first navigate to this `/sys/fs/cgroup/cpuset/lxc/n1` directory and then to view the content of this cgroup item, the command is

```
cat cpuset.cpus
```

Figure 4.67 demonstrate the CPU usage of n1. The usage has been limited to 1 core only.

```
root@mahnaz-VB:~# cd /sys/fs/cgroup/cpuset/lxc/n1
root@mahnaz-VB:/sys/fs/cgroup/cpuset/lxc/n1# cat cpuset.cpus
1
root@mahnaz-VB:/sys/fs/cgroup/cpuset/lxc/n1#
```

Figure 4.67 Validation of CPU Usage in n1 LXC node

Then the next is, to check the cpu usage of n3 node, navigate to the directory as the same way like n1 node and view the content of n3 cpuset. During the configuration for n3 node, the CPU usage has been defined as 2 core which is presented in Figure 4.68.

```
root@mahnaz-VB: /sys/fs/cgroup/cpuset/lxc/n3
File  Edit  View  Search  Terminal  Help
root@mahnaz-VB:~# cd /sys/fs/cgroup/cpuset/lxc/n3
root@mahnaz-VB:/sys/fs/cgroup/cpuset/lxc/n3# cat cpuset.cpus
2
root@mahnaz-VB:/sys/fs/cgroup/cpuset/lxc/n3#
```

Figure 4.68 Validation of CPU Usage in n3 LXC node

4.6.6 Limit Memory Usage

During the configuration of n1 node, the memory usage has been limited to 2G (see Figure 4.56). to validate the implementation, navigate to `/sys/fs/cgroup/memory/lxc/n1` and view the limied usage of bytes. The following command to view the memory allocation.

```
cat memory.limit_in_bytes
```

Figure 4.69 shows the limited memory usage in bytes.

```
root@mahnaz-VB: /sys/fs/cgroup/memory/lxc/n1
File  Edit  View  Search  Terminal  Help
root@mahnaz-VB:~# cd /sys/fs/cgroup/memory/lxc/n1
root@mahnaz-VB:/sys/fs/cgroup/memory/lxc/n1# cat memory.limit_in_bytes
2147483648
root@mahnaz-VB:/sys/fs/cgroup/memory/lxc/n1#
```

Figure 4.69 Validation of Memory Usage in n1 LXC Container

Then next is to check the memory usage of n3 LXC node. To do this, navifate to the specific directory of n3 and view the memory allocation. Figure 4.70 shows that the allocated memory for n3 is 4G which is showing in bytes.

```
root@mahnaz-VB:~# cd /sys/fs/cgroup/memory/lxc/n3
root@mahnaz-VB:/sys/fs/cgroup/memory/lxc/n3# cat memory.limit_in_bytes
4294967296
root@mahnaz-VB:/sys/fs/cgroup/memory/lxc/n3#
```

Figure 4.70 Validation of Memory Usage in n3 LXC Container

4.6.7 Validation of Mount Directory

According to Figure 4.56, the "mount-directory" option has this "`/home/mahnaz/hook/`" directory to be mounted in n1 LXC container. So, Figure 4.71 portrays that inside `/mnt` this, it contains the file named "`pre-start.sh`" which is belong to `/home/mahnaz/hook/` folder on the host system previously.

```
Terminal
File  Edit  View  Search  Terminal  Help
root@n1:/# cd /mnt
root@n1:/mnt# ls
pre-start.sh
root@n1:/mnt# cat pre-start.sh
#!/bin/bash
LOG_FILE=/tmp/container.log
echo "Container name: $LXC_NAME" | tee -a $LOG_FILE
echo "Container mounted rootfs: $LXC_ROOTFS_MOUNT" | tee -a $LOG_FILE
echo "Container config file $LXC_CONFIG_FILE" | tee -a $LOG_FILE
echo "Container rootfs: $LXC_ROOTFS_PATH" | tee -a $LOG_FILE
root@n1:/mnt#
```

Figure 4.71 Mount directory in n1 LXC Container

Similarly, the "mount-directory" option has different path location "`/home/mahnaz/Desktop/test_project/`" for n3 LXC configuration (see Figure 4.57). Now, Figure 4.72 illustrates that inside `/mnt`, it contains the files which were in this "`/home/mahnaz/Desktop/test_project/`" location of host earlier.

```
Terminal
File  Edit  View  Search  Terminal  Help
root@n3:/# cd /mnt
root@n3:/mnt# ls
testLxc.py  testLxc.pyc
root@n3:/mnt# cat testLxc.py
import lxc
import sys
import subprocess

def lxc_test(name):
        c=lxc.Container(name)
        if c.defined:
```

Figure 4.72 Mount Directory in n3 LXC Container

4.6.8 Validation of Start-up Command

A start-up command will be executed inside the LXC container, although it will be defined in the LXC configuration window. For n1 LXC container, user-defined command was to create a file named “x” using `touch` command

To verify the implementation, Figure 4.73 is outlined. Here it is shown that the file called “x” has been created inside n1 LXC container.

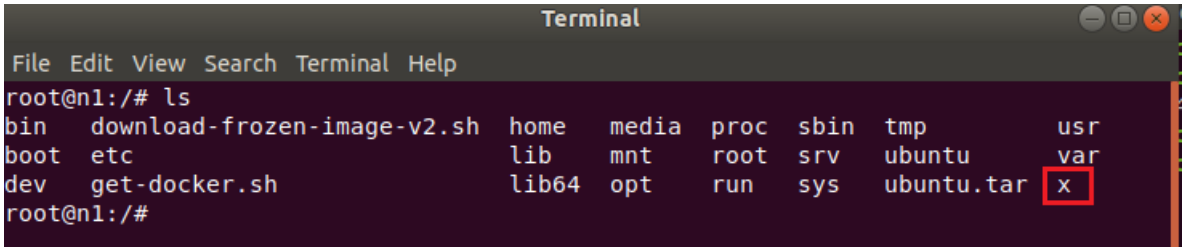

Figure 4.73 Startup Command Executed in n1 LXC Container

Figure 4.74 presents the start-up command implementation inside n3 LXC container. Here, “file” has been created in n3 LXC container.

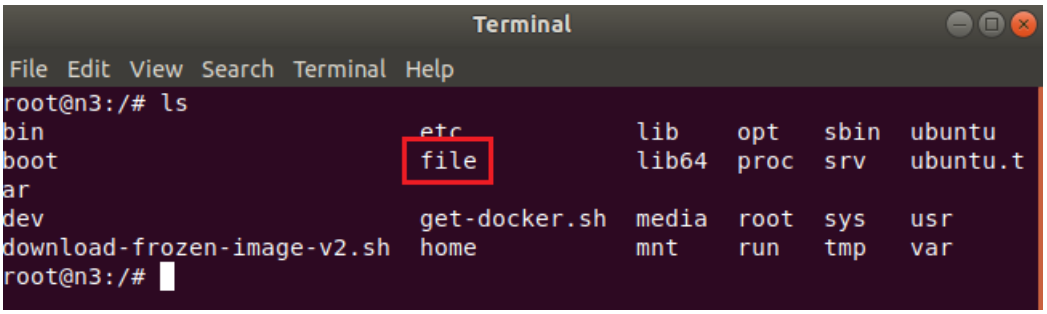

Figure 4.74 Startup Command Executed in n3 LXC Container

Both the commands have been defined before the container has been created and started.

4.6.9 Validation of Observer Widget on LXC Node

Observer widget is used to display the information about a node in CORE emulator. To enable this tool for LXC node, first select "Widgets" from the menubar of CORE canvas. Then, navigate to observer widgets. This will open a list of widgets. In this case, "ifconfig" is selected to observe the result (Figure 4.75).

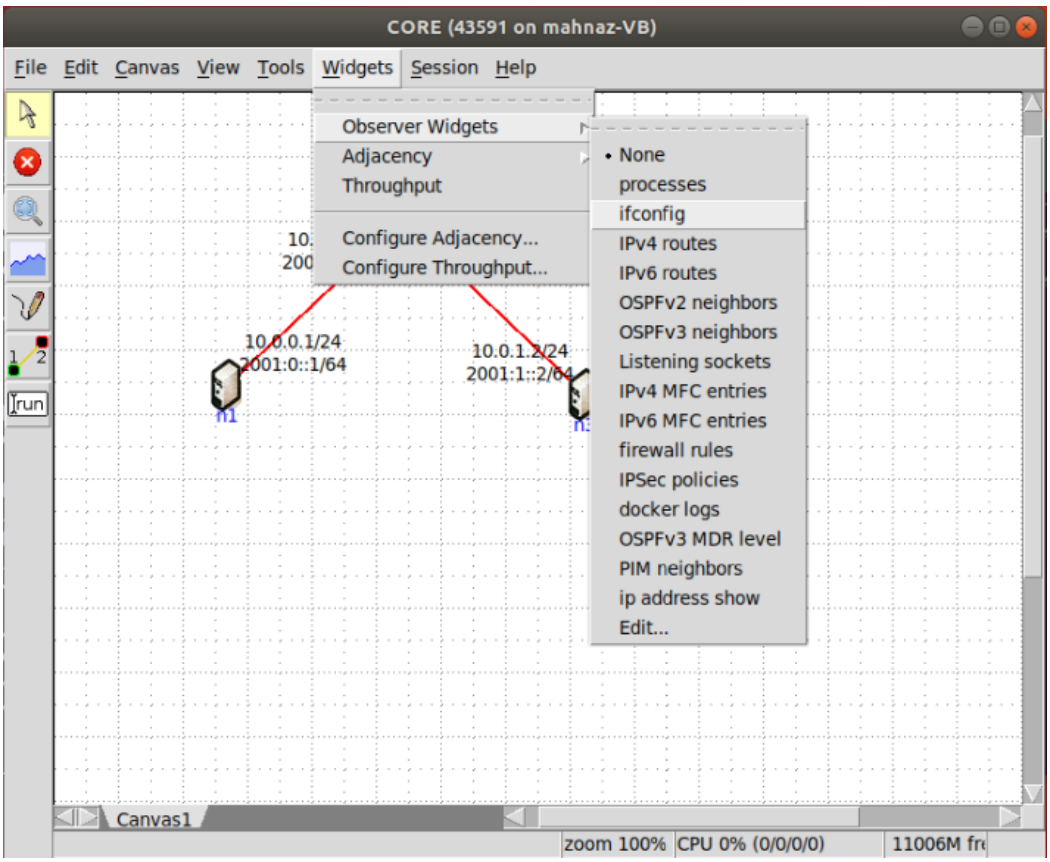

Figure 4.75 Validation for Observer Widget for LXC Node

After selecting the widget option from the list, the Edit mode of CORE GUI has changed to Execute mode. Figure 4.76 demonstrates that when user hovers over the LXC node, it is executing the shell command "ifconfig" on LXC node and return the result.

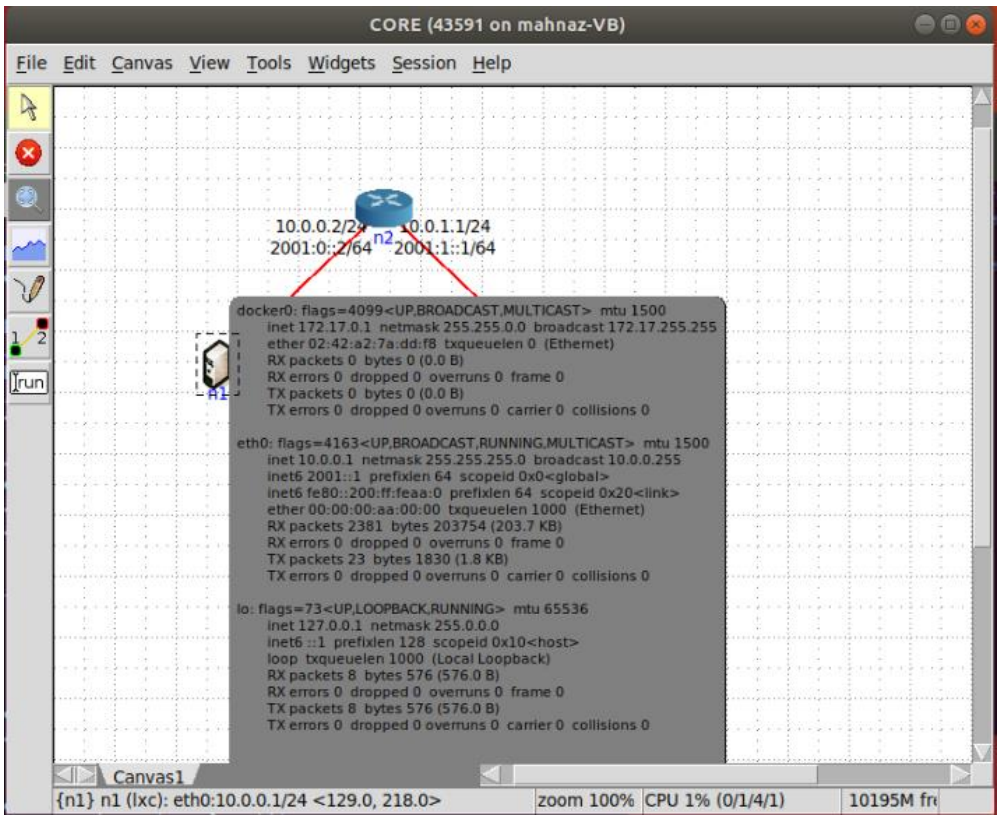

Figure 4.76 Observer Widget Reutn the Output Result

In order to demonstrate the rest of the widgets on LXC node, it is presenting through CORE API message in Figure 4.77. for all three nodes (n1, n2, n3), different command have been executed.

```
>EXEC(flags=0x30,n1,n=116,cmd='ps -e')
EXEC(flags=0,node=1/exec=116,cmd=ps -e,res=(606 bytes),status=0,)
>EXEC(flags=0x30,n1,n=117,cmd='ps -e')
EXEC(flags=0,node=1/exec=117,cmd=ps -e,res=(606 bytes),status=0,)
>EXEC(flags=0x30,n2,n=118,cmd='/sbin/ifconfig')
EXEC(flags=0,node=2/exec=118,cmd=/sbin/ifconfig,res=(1438 bytes),status=0,)
>EXEC(flags=0x30,n2,n=119,cmd='/sbin/ifconfig')
EXEC(flags=0,node=2/exec=119,cmd=/sbin/ifconfig,res=(1438 bytes),status=0,)
>EXEC(flags=0x30,n2,n=120,cmd='/sbin/ifconfig')
EXEC(flags=0,node=2/exec=120,cmd=/sbin/ifconfig,res=(1438 bytes),status=0,)
>EXEC(flags=0x30,n3,n=121,cmd='ip address show')
EXEC(flags=0,node=3/exec=121,cmd=ip address show,res=(1023 bytes),status=0,)
>EXEC(flags=0x30,n3,n=122,cmd='ip address show')
EXEC(flags=0,node=3/exec=122,cmd=ip address show,res=(1023 bytes),status=0,)
```

Figure 4.77 Observer Widget Implementation on LXC Node Validation

4.6.10 Stress Test on LXC Container and CORE Node

Stress is used to impose load on and stress test systems. It is a useful thing to test the performance of Linux hardware components for example, CPU, RAM, disk devices and so on. In order to test and monitor the performance of a LXC container, first the package for `stress` and for monitoring `htop` need to be installed in any template image using distrobuilder. Then, it is required to compile the image using "`build-lxc`" command.

After that, container will be created and started inside CORE and open the shell window for LXC container. Now, to stress test the CPU, the command should be given like below:

```
$ stress -c 4
```

Here, “c” refers to CPU. This means, the CPU utilization will be increased to 4 core. Initially, the CPU usage was set to 1 core for n1 node. Figure 4.78 demonstrates that the CPU usage via `htop` and it is using 1 core completely.

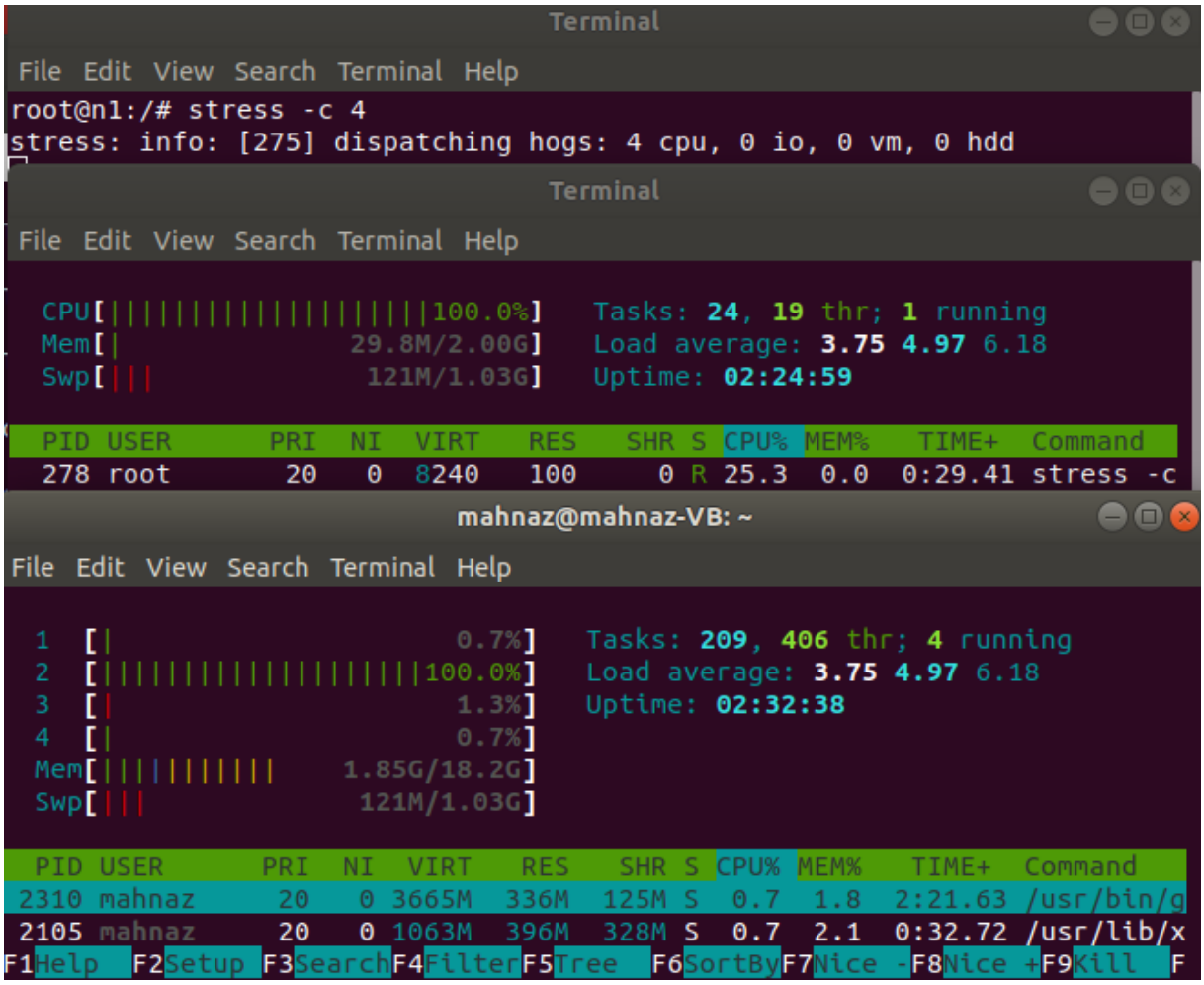

Figure 4.78 Stress Test to Monitor CPU Utilization of LXC Node

Now, the same test will be conducted on CORE node to validate the fact. Figure 4.79 presents the stress test result. In n2 CORE node, 4 core has been imposed and it is utilizing all core completely 100%. The system resources are not limited in CORE.

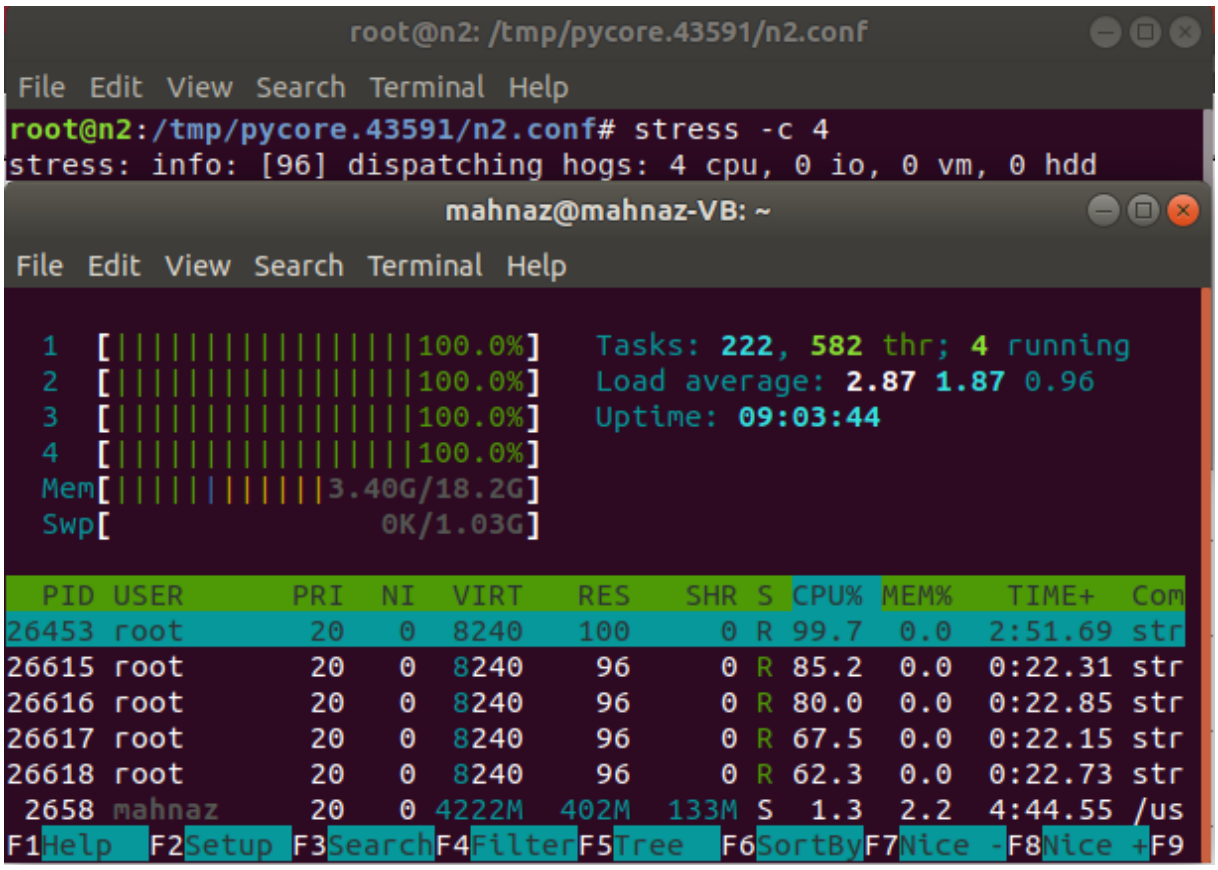

Figure 4.79 Stress Test to Monitor CPU Utilization of CORE Node

The next task is, stress test to memory. In order to stress test memory, the amount of memory needs to be defined to fill up memory and observe how much it has filled up. For example, in Figure 4.80 it is seen that, the maximum usage limit was 2G in n1 LXC node and after stress test it is filling up almost.

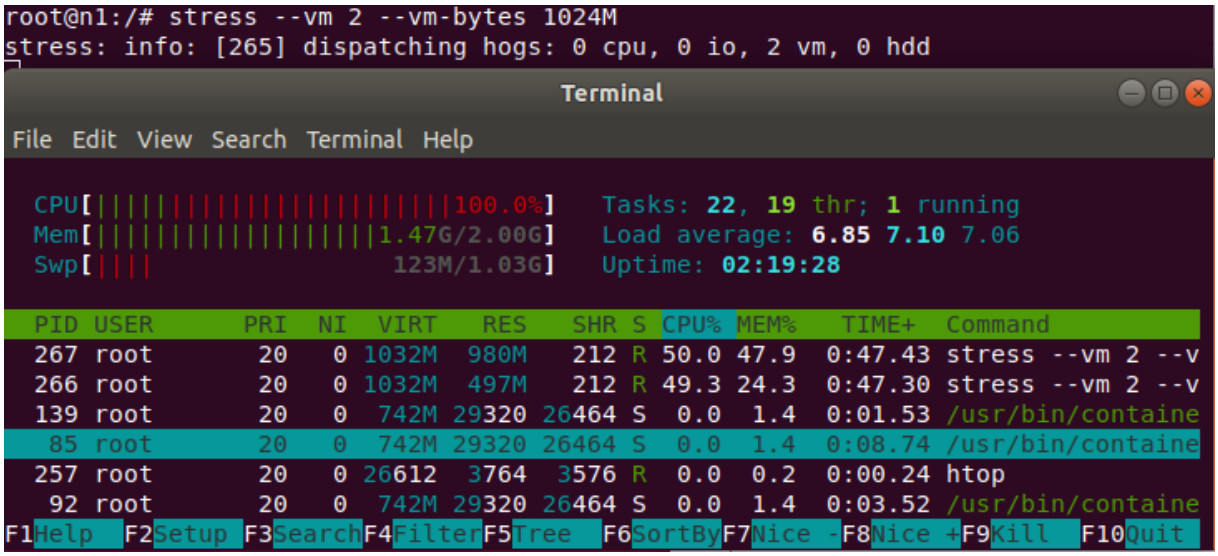

Figure 4.80 Stress Test to Monitor Memory Allocation on LXC Node

And lastly, the same test has been done on n2 CORE node and Figure 4.81 shows that the memory filled up more that 4G.

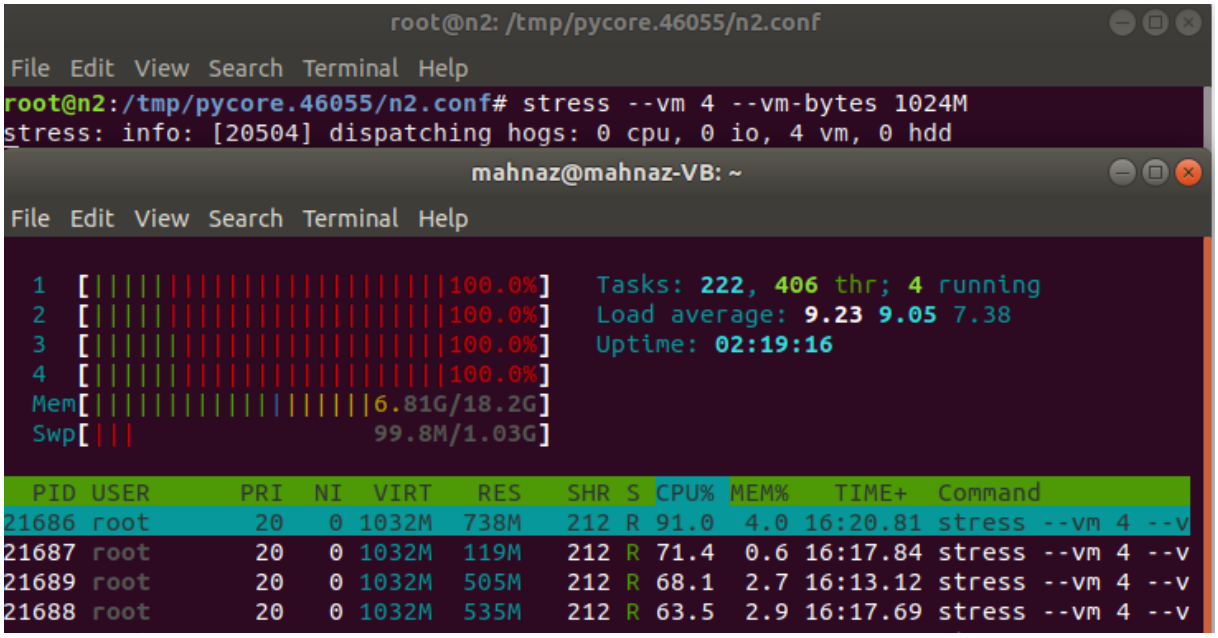

Figure 4.81 Stress Test to Monitor Memory Allocation on CORE Node

Hence, this stress test validates the fact that the system resources are not limited in CORE, whereas in LXC container, the usage of resources are limited.

5 Summary and Perspectives

In this chapter, summary of the implementation and some proposal of future perspectives will be described, there will be a suggestion to extend the present work in future.

5.1 Summary

In this thesis, the obejctive was to integrate an LXC-based node type into the CORE emulator network to support LXC container as node in CORE emulator. To develop the LXC node type into CORE emulator Tcl/Tk for CORE GUI , python for CORE Daemon and CORE emulator network tool has been used.

To enable the usage of LXC container into CORE emulator, the LXC container has been developed as a node type in CORE emulator. In this work, for developing the LXC node type, the current CORE emulator implementation has been extended. Moreover, to work around into the CORE emulator, LXC related some configurable aspects such as container template image selection, limiting the usage of CPU and RAM, mounting directory from host machine and a startup command has also been developed.

At first, it was required to investigate the current CORE emulator network implementation to identify the phase for creating network namespace and to identify which method is responsible for this. To do this, the CORE API message flow and the CORE Daemon logfile has been examined and identified that in Configuration state the network namespace is creating and the default network namespace based CORE node is being used. Additionaly, there was an another investigation regarding the configuration part as CORE provides the possibility to configure a session. Based on all these research, the approach to develop the LXC node type has been outlined. Since CORE is using default network namespace based node type so, a new node type is required to support LXC container into the CORE emulator. Then, to configure the LXC container, the implementation for current CORE session configuration manager has been extended and some configurable options such as container-image list, limit CPUSet, limit memory usage, mount directory and startup-command has been defined for LXC configuration.

The required communication between CORE GUI and CORE Daemon for exchanging the configuration information for LXC node type has been developed using CORE API message. To provide the communication interface between CORE GUI and CORE Daemon for LXC node, it extends the current CORE API impemntation.

In CORE GUI, the implementation has taken place into several modules of GUI. Before creating the LXC node type, the LXC node image icon, the popup configuration window and button for configuration has been developed in CORE GUI.

To create the LXC container image, distrobuilder has been used. This tool helps to create the system container image and it gives the possibility of storing the images in local directory of a host machine. As a result, it can minimize the loading time during the booting time of a container inside CORE.

LXC-Python bindings provides all the necessay methods to limit the cgroup items like limiting CPU and Ram usage. In this thesis work, CPU and Ram usage has been controlled using this LXC-python bindings. Moreover, this LXC-Python bindings has been used to create, start, stop and destroy the container inside CORE.

To implement the network connection between the nodes, the existing CORE implementaion has been extended for LXC node type. Since CORE nodes connect to each other via Linux Ethernet bridge, so on the basis of this concept, the convenient approach is to adapt the current implementation for LXC node to network connection. To apply the connection between the nodes, a pair of virtual ethernet (veth) has been specified for the implementation. One side of the virtual ethernet (veth) pair has been attached to the LXC container and the other side of the veth pair has been assigned to the corresponding CORE bridge network.

For the configurable options mount directory and startup-command, python's OS module has been used in this implementation.

Finally, the evaluation has been done regarding analysing the configurable aspects CPU and RAM usage for both LXC and CORE nodes. From the analysis, it was visible that LXC nodes are successful to limit and control the usage of CPU and memory rather than CORE nodes. CORE nodes does not provide the possibility to manage the resource control but integrating LXC node into CORE emulator, it is simple and effective to manage the system resources.

5.2 Future Scope and Conclusion

From the above discussion, it is seen that LXC container can be configured as per user requirements. So, this thesis work creates some scope which can be done in further reseach work.

There can be possibility to make a container persistent. At the moment, the current implementation does not provide this because when the session stops, the containers are destroyed. Therefore, the further development can be, to make the container reusable for another CORE session.

Apart from this, there is another possibility to enable container nesting virtualisation as a future work of this thesis work. This will make run an LXC container within another LXC container.

The target of this thesis work is to integrate LXC container as a nodetype into CORE emulator network so that it can provide the possibility to limit the usage of system resources. In the theoretical part, the background concept and some technical aspects of these technologies have been discussed. In the realisation part, the new node type has been defined and some configurable aspects has been developed based on the use-cases which eventually helped to evaluate the performance based on CPU and RAM usage of LXC container as node type. In the summary, there is a discussion about the implemented approach. Additionaly, some further work scope has been defined which may develop in future.

6 Abbreviations

A

API	Application Programming Interface

B

Blkio	Block I/O

C

CFQ	Completely Fair Queuing
CFS	Completely Fair Schedulers
CGROUPS	Control Groups
Chroot	Change Root
CORE	Common Open Research Emulator
COW	Copy-on-write
CPU	Central Processing Unit

G

GUI	Graphical User Interface

H

HAL	Hardware Abstraction Layer

I

I/O	Input/output
IP	Internet Protocol
IPC	Inter Process Communication

L

LXC	Linux Containers

M

MAC	Media Access Control

O

OS	Operating System
OSPF	Open Shortest Path First
OVS	Open vSwitch

P

PC	Personal Computer
PID	Process ID

R

RAM	Random Access Memory
RFC	Request for Comments
Rootfs	Root File System

T

TCL	Tool Command Language
TCP	Transmission Control Protocol
TLV	Type, Length, Value

U

UML	Unified Modelling Language
UTS	Unix Time Sharing

V

VM	Virtual Machine
Veth	Virtual Ethernet

7 References

1. Ahrenholz, J., Danilov, C., Henderson, T. And Kim, J. (2008): *CORE: A real-time network emulator,* [online] https://ieeexplore.ieee.org/document/4753614 [accessed 03 May 2019]
2. Archlinux (2019): *Linux Containers,* [online] https://wiki.archlinux.org/index.php/Linux_Containers [accessed 15 February 2019]
3. Bregman, A. (2016): *Linux: Network Namespace,* [online] https://devinpractice.com/2016/09/29/linux-network-namespace/ [accessed 10 March 2019]
4. Carvalho, A. (2017): *Using cgroups to limit I/O,* [online] https://andrestc.com/post/cgroups-io/ [accessed 20 July 2019]
5. Cgroups (2019): *cgroups - Linux control groups,* [online] http://man7.org/linux/man-pages/man7/cgroups.7.html#top_of_page [accessed 21 March 2019]
6. CORE (2015): *CORE Documentation Release 4.8*, [online] https://downloads.pf.itd.nrl.navy.mil/docs/core/core_manual.pdf [accessed 08 February 2019]
7. Core (2019): *Using the CORE GUI*, [online] http://coreemu.github.io/core/usage.html [accessed 15 July 2019]
8. coreemu/core (2019): *Common Open Research Emulator*, [online] https://github.com/coreemu/core [accessed 08 February 2019]
9. CORE API (2019): *CORE IPC API Version 1.23*, [online] https://downloads.pf.itd.nrl.navy.mil/docs/core/core_api.pdf [accessed 07 March 2019]
10. Eder, M. (2016): *Hypervisor- vs. Container-based Virtualization*, [online] https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf [accessed 22 June 2019]
11. Garcia, D. (2016): *Network Namespaces,* [online] https://blogs.igalia.com/dpino/2016/04/10/network-namespaces/ [accessed 11 March 2019]
12. GeeksforGeeks (2019): *OS Module in Python with Examples,* [online] https://www.geeksforgeeks.org/os-module-python-examples/ [accessed 30 March 2019]
13. Ivanov, K. (2017): *Containarization with LXC,* Birmingham: Packt Publishing Ltd.
14. Kelly, D. (2016): *A History of Container Technology*, [online] https://blog.container-ship.io/a-history-of-container-technology [accessed 10 July 2019]
15. Kumaran, S. (2017): *Practical LXC and LXD*, Berkeley: Apress
16. Linux containers (2019): *LXC: Security,* [online] https://linuxcontainers.org/lxc/security/ [accessed 01 march 2019]
17. Linux containers(2019): *What's LXC?* [online] https://linuxcontainers.org/lxc/introduction/# [accessed 12 February 2019]
18. Linux Journal (2018): *Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers (LXC)* [online] https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lxc [accessed by 25 February 2019]

19. Linux containers (2019): lxc.container.conf [online] https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html [accessed 10 February 2019]
20. lxc/distrobuilder (2019): *System container image builder for LXC and LXD*, [online] https://github.com/lxc/distrobuilder [accessed 12 April 2019]
21. lxc/python3-lxc (2019): *Python 3.x binding for liblxc,* [online] https://github.com/lxc/python3-lxc [accessed 27 February]
22. lxc (2019): *lxc-python2 0.1,* [online] https://pypi.org/project/lxc-python2/ [accessed 27 February]
23. OS (2019): *os — Miscellaneous operating system interfaces*, [online]https://docs.python.org/2/library/os.html [accessed 04 April 2019]
24. PyMOTW (2019): *subprocess – Work with additional processes,* [online] https://pymotw.com/2/subprocess/ [accessed 05 April 2019]
25. Red Hat (2019): *Chapter 3.Subsystems and Tunable Parameters*, [online] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch-subsystems_and_tunable_parameters [accessed 25 July 2019]
26. Sikeridis, D., Papapanagiotou, I., Devetsikiotis, M. and Rimal, B. (2017): *A Comparative Taxonomy and Survey of Public Cloud Infrustructure Vendors*, [online] https://www.researchgate.net/publication/320223370_A_Comparative_Taxonomy_and_Survey_of_Public_Cloud_Infrastructure_Vendors [accessed 20 June 2019]
27. Subprocess (2019): *subprocess – Subprocess management,* [online] https://docs.python.org/2/library/subprocess.html [accessed 05 April 2019]
28. Tcl Developer Xchange (2019): *Welcome to the Tcl Developer Xchange!,* [online] https://www.tcl.tk/ [07 March 2019]
29. Ubuntu documentation (2019): *LXC,* [online] https://help.ubuntu.com/lts/serverguide/lxc.html [accessed 15 February 2019]
30. Xenitellis, S. (2018a): *Using distrobuilder to create container images for LXC and LXD*, [online] https://blog.simos.info/using-distrobuilder-to-create-container-images-for-lxc-and-lxd/ [accessed 12 April 2019]
31. Xenitellis, S. (2018b): *How to create a minimal container image for LXC/LXD with distrobuilder*, [online] https://blog.simos.info/how-to-create-a-minimal-container-image-for-lxc-lxd-with-distrobuilder/ [accessed 12 April 2019]

8 Appendix

A CD in attachment which contain following

1. This thesis in PDF format
2. This thesis in DOC format
3. Source Code