Container Network Topology Modelling

João Miguel Soares de Matos Neves

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor: Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Fernando Manuel Valente Ramos

November 2018

Acknowledgments

First of all, I would like to express my gratitude to Professor Miguel Matos for the opportunity to work on this subject, and for the countless hours invested in this work.

Furthermore I would like to thank Professor Valerio Schiavoni from the University of Neuchâtel, for all the useful insights, and for granting me access to the cluster where this work was evaluated.

Finally I would like to thank Professor João Leitão and Pedro Fouto from the Universidade Nova de Lisboa for the discussions on Cassandra, Professor Alysson Bessani and João Sousa from the Faculdade de Ciências da Universidade de Lisboa for the help regarding BFT-SMaRt, and Professor Hughes Mercier from the University of Neuchâtel for the help regarding the jitter model.

This work was partially supported by Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa and by Fundação para a Ciência e Tecnologia (FCT) through projects with reference UID/CEC/50021/2013 and LISBOA-01-0145-FEDER-031456.

Resumo

As aplicações distribuídas são fortemente influenciadas pelas propriedades da rede subjacente, por exemplo, largura de banda, latência e perda de pacotes. É por isso importante que os investigadores e engenheiros sejam capazes de avaliar o impacto destas propriedades no desempenho do sistema e no seu comportamento correcto. Estas propriedades de rede têm efeitos observáveis e mensuráveis directamente nas aplicações, enquanto que o comportamento interno dos elementos de rede subjacentes, como comutadores e roteadores, é mais difícil de capturar da perspectiva da aplicação. Esta observação permite explorar novos desenhos de emuladores que apenas emulam o comportamento macro de topologias complexas em vez dos detalhes internos. Apresentamos o NEED, um emulador de topologias de rede descentralizado que explora esta ideia. O NEED emula uma topologia de rede sob aplicações não modificadas em contentores, é agnóstico relativamente à linguagem de programação e protocolo de transporte e pode escalar para milhares de contentores bastando para isso adicionar mais nós a um cluster de máquinas. A nossa avaliação mostra que a precisão da emulação do NEED está ao mesmo nível de outros sistemas modernos que emulam o estado completo da rede. Mostramos também que o NEED pode ser usado não só para reproduzir resultados anteriores presentes na literatura, mas também para avaliar o comportamento de sistemas geodistribuídos como se fossem colocados a funcionar numa WAN real.

Palavras-chave: Emulação de Redes, Contentores, Descentralizado, Sistemas Distribuídos, Reproducibilidade

Abstract

Distributed applications are heavily influenced by the properties of the underlying network, *i.e.*, bandwidth, link latency, packet loss rate and jitter. Nevertheless, researchers, practitioners and engineers should be able to assess the impact of those properties on the system's performance and correctness. Those network properties have direct observable and measurable effects on the application, while the internal behavior of the underlying network elements, such as switches and routers, is harder to capture from an application perspective. This observation leads us to explore designs that only emulate the macro behavior of complex topologies rather than its internal details. We present NEED, a decentralized network topology emulator that explores this idea. It emulates a network topology beneath unmodified containerized applications, it is agnostic of the application language or transport protocol and can scale to thousands of containers by adding more nodes to a cluster of commodity machines. Our evaluation shows that NEED emulation accuracy is on-par with state-of-the-art systems that emulate the full state of the network. We show that it can be used not only to reproduce previous results from the literature but also to assess the behavior of geo-distributed systems *as if* they were deployed on a WAN testbed or production network.

Keywords: Network Emulation, Containers, Distributed Systems, Decentralized, Reproducibility

Contents

List of Tables

List of Figures

List of Acronyms

API	Application Programming Interface
CNI	Container Network Interface
CNM	Container Network Model
GUI	Graphical User Interface
IPAM	IP Address Management
LAN	Local Area Network
RTT	Round-Trip Time
SDN	Software Defined Networking
SLO	Service Level Objectives
VLAN	Virtual Local Area Network
WAN	Wide Area Network

Chapter 1

Introduction

The development and testing of distributed systems is a complex task mainly because distributed systems must execute on a variety of different scenarios, including different hardware and network capabilities, as well as different configurations. A given distributed system might appear to be working correctly when tested on a single machine or even on a Local Area Network (LAN), but then behave incorrectly when deployed on a Wide Area Network (WAN). Furthermore the underlying network topology can have a drastic impact on performance metrics such as application throughput and latency. With the increasingly popular deployment of geographically-distributed applications operating at a global scale, assessing the impact of geo-distribution, and hence network topology, is fundamental to build and tune applications that perform correctly and meet the desired Service Level Objectives (SLO).

Unfortunately it is hard and costly to perform these tests during development. On one hand, the deployment of such geographically distributed systems was made simpler thanks to the increasing popularity of container technology (*e.g.* Docker [Mer14], Linux LXC [lxc]). These technologies have been adopted by big IT players in their commercial offerings (*e.g.* Amazon Elastic Container Service [amab]) and became the de-facto standard to deploy large-scale applications. On the other hand, evaluating the performance of such systems once deployed in production networks, especially at the early stages of prototyping, is a particularly daunting, time-consuming and expensive task. This is mainly due to the inherent variability of network conditions (*i.e.* failures, contention and reconfigurations). Hence, it is hard (if possible at all) to systemically reproduce a specific system behavior, such as observing a performance bottleneck, or force pathological cases, such as a high packet loss in a particular network segment, as these events are out the of control of the system deployers.

To overcome these difficulties both network simulation and network emulation systems have been developed. According to [BRNR15], in simulation 'a model of an application is tested on a model of the platform', while in emulation 'a real application is tested on a model of the platform'. Conclusions obtained from simulation are completely reproducible, however it can be argued that the results are not obtained in real world conditions and can therefore be unrealistic [BRNR15]. Conclusions obtained from emulation are considered more realistic since the environment is closer to real world conditions, however, results can be harder to reproduce due to the less stable conditions of real software and networks.

Being able to obtain the same results from running an experiment more than once is a crucial feature for a network experimentation system. According to [BRNR15] an experimentation system should achieve replicability, if it has the ability to produce the same results when running the same experiment multiple times under the same environment (same testbed, same researcher). Achieving reproducibility however requires that the system can produce the same results even if the same experiment is executed in different environments (different testbed, different researcher). Although reproducibility is more desirable than replicability, achieving reproducibility places further requirements on the network experimentation system, such as having awareness of the limits in the testbed. If this knowledge is not taken into consideration when deploying an experiment, and the limit of the available resources is reached, then results will be skewed, and can vary when executed on different testbeds.

Furthermore, there have been two approaches to the design of network experimentation systems: dedicated testbeds and tools. Dedicated testbeds require dedicated hardware resources and customized software (like the underlying operating systems) to be configured solely for the purpose of executing experiments. An example of this is Emulab [HRS+08b], where virtual nodes, must execute on a physical node dedicated only to the execution of experiments. Since they require significant effort to set up, the same testbed is usually shared between multiple researchers. Network experimentation tools are more flexible by allowing experiments to be run under broader conditions and are also generally easier to setup. An example of this is Mininet [LHM10], where the goal is to have a network emulation system that can execute on any machine.

One of the biggest challenges to the general adoption of experimentation systems is the high entry barrier. A researcher must spend considerable time learning how the chosen system works and specifying the desired experiment. We argue that some of the most popular emulation systems are more focused on the internal behavior of the network, while distributed systems engineers are more interested in how the macro aspects of the network - namely, end-to-end latency, jitter, available link bandwidth and packet loss - affect the system. Also some other systems require the user to specify their experiments using languages or frameworks that are not commonly used.

Under this perspective, we propose NEED, an end-to-end decentralized network topology emulation tool. NEED can emulate an arbitrary network topology without having to model network elements such as routers or switches. This results in a simple, fully distributed design, where the key idea is to collapse the full network topology to virtual links that retain the same end-to-end network properties. Furthermore we make use of Docker containers as the deployment unit, using terminology and workflows that are already familiar to Docker users. The use of Docker containers also aids in achieving replicability of experiments, thanks to the portable nature of containers which limits the variability of the environment available to the applications.

1.1 Objectives

The objective of this work is to design and evaluate an architecture and implementation for a network emulation system. The network characteristics to be emulated are latency, jitter, packet loss and bandwidth. The proposed network emulation system is targeted at distributed systems developers, as such the system should allow for large scale deployments of resource intensive applications. It should also abstract away the details of network specific elements, that are not relevant from an application point of view. Furthermore the proposed system should at least achieve replicability of experiments.

1.2 Outline

The remainder of this document is divided into the following sections. In Section 2 we provide some background by covering the technologies used by NEED. In Section 3 we discuss existing network experimentation systems described in literature. In Section 4 we explain the design and architecture of NEED and discuss the decisions and alternative solutions. In Section 5 we describe in detail how NEED is implemented. In Section 6 we evaluate NEED by showing experiment results and discussing them. In Section 7 we conclude with a brief overview of the features offered by NEED, and a discussion of future work possibilities.

Chapter 2

Background

The fundamental building blocks for a network emulation system are the mechanisms it uses to deploy applications, and to enforce network properties. In this section we will cover the technologies on which we will build upon to implement the proposed network emulation system.

First, we will cover container technologies, namely the Docker platform. We choose to use containers, and Docker in particular, due to the their agility of deployment and networking stack isolation between instances.

Second, we will cover the methods the Linux kernel provides to perform traffic shaping. We decided to use this technology due to its good performance and accuracy, and also because its management is greatly simplified when performed inside containers, due to the isolation of the networking stack.

2.1 Containers

Docker[Mer14][Doc] is a widely used container platform that leverages lightweight virtualization techniques provided by the Linux kernel to run processes in an isolated environment. Furthermore, Docker provides utilities to facilitate the processes of development and deployment of applications, as well as utilities to manage clusters of deployed containers. To achieve process isolation Docker leverages Linux kernel functionality such as kernel namespaces[nam17]. Of particular interest to this work are the mechanisms Docker uses to both isolate the networking stack available inside containers, and to connect containers together in networks, even when containers are deployed across separated physical hosts. Through the rest of this section we will refer to machines running the Docker daemon as hosts, and will refer to hosts belonging to a cluster as nodes.

In the following subsections we will briefly explain how Docker provides network isolation and connectivity between the containers, how management of Docker clusters can be performed, and also how traffic shaping can be performed on Linux.

Network isolation

Docker container networking is managed by **Container Network Model (CNM)** plugins. It is the responsibility of the chosen plugin to setup the networking inside the container as well as setup any necessary routing information on the host. Docker plugins work by creating a virtual Ethernet pair (two virtual interfaces connected together). Docker then moves one end of the pair to a unique network namespace and assigns the processes running inside the container to that network namespace [Hau16]. All plugins perform this setup step with the exception of the **host** plugin, which gives the container direct access to the network interfaces of the host. This operation gives the container a unique network interface together with an independent IP stack, routing tables, firewall rules and other kernel networking data structures, and isolates the container from other existing interfaces (both on the host and on other running containers).

Network connectivity

The default plugin used by Docker on single host deployments is the **bridge** plugin. The bridge plugin creates a bridge interface on the host[Hem12], and creates a virtual Ethernet pair for each container that is then attached to the host's bridge (See Figure 2.1).

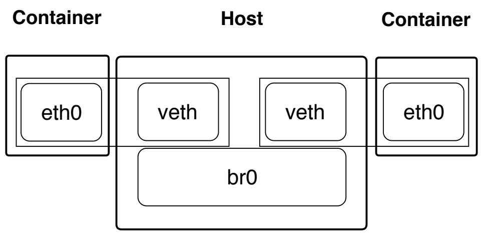


Figure 2.1: Illustration of the bridge driver.

This allows individual containers to have separated IP stacks, and to communicate with each other through the bridge. Since the bridge exists only inside a single host this approach works only for single host deployments.

For greater scalability, we are however interested in connecting together containers, that are running across a cluster of physical machines executing the Docker daemon. We will name such a deployment scenario a multi host deployment for the rest of this document.

The default plugin used by Docker on multi host deployments is the **overlay** plugin. The overlay plugin extends the bridge plugin and achieves connectivity between nodes through the use of **VXLAN tunnels**[MDD$^+$14].

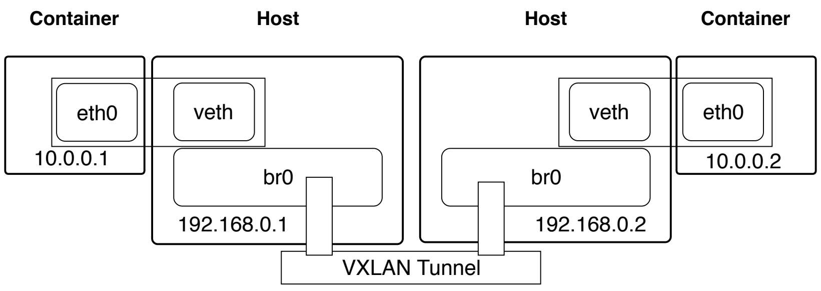


Figure 2.2: Illustration of the overlay driver.

VXLAN tunnels encapsulate data between endpoints that are created at each node in the cluster. The tunnels encapsulate data from layer two and above, generated by the containers connected to this network, into a layer three packet on the underlying infrastructure.

This allows for routing through the underlying physical network infrastructure to be done transparently, and requires traffic to be encapsulated/de-encapsulated only at the tunnel endpoints (See Figure 2.2).

A different way to achieve multi host container networking is through the **macvlan** plugin. The macvlan plugin leverages the Linux kernel ability to create virtual child interfaces, that get assigned a different MAC address, but otherwise share the same network connection as the parent interface [Lit12] (See Figure 2.3).

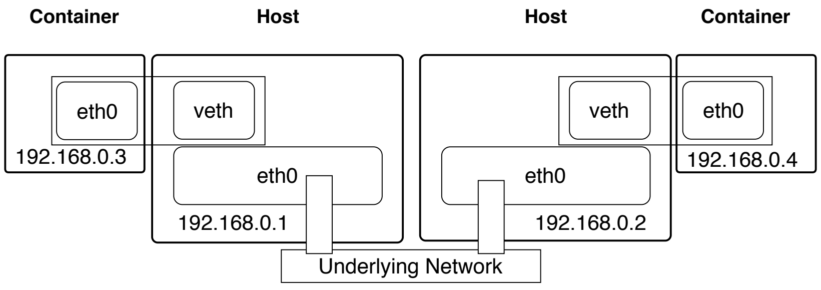


Figure 2.3: Illustration of the macvlan/ipvlan driver.

This allows for two typical setups. In the first one we use a physical network interface as the parent and create virtual interfaces that behave as if the host had multiple physical interfaces sharing identical connectivity. This setup allows the containers to share the same underlying physical networking infrastructure as the host, while maintaining networking stack isolation between virtual interfaces.

The second typical setup involves using a Virtual Local Area Network (VLAN) (802.1q) [IEE06] parent interface. This setup shares the same properties as the first setup but allows to create further isolation from other networks in the cluster. The **macvlan** plugin requires that the parent interface be placed in promiscuous mode. In some situations, such as when Virtual Machines are used, this might not always be possible.

Another plugin that allows for multiple host container networking is the **ipvlan** plugin. The ipvlan plugin is similar to the macvlan plugin, but instead of assigning a different MAC address to each sub-interface they share the same address MAC address, but use different IP addresses. Routing of packets is performed by the kernel in either layer 2 or layer 3 mode, assuring that incoming/outgoing packets are correctly delivered to/from all the sub-interfaces. Unlike the macvlan plugin, the ipvlan plugin does not need to set the parent interface in promiscuous mode, which allows for deployment in scenarios where macvlan can not be used.

Beyond the plugins presented above, there are several third-party plugins such as Calico, Flannel and Weave [Hau16]. These plugins also allow multi host networking based on mechanisms similar to the ones used by the overlay plugin, but provide features that give greater control to the user configuring the network, namely allowing for complex policies to be defined. These plugins are not available in the default Docker installation and must be manually installed by the user.

Cluster management

When creating any IP network, the interfaces connected to it must be assigned unique IP addresses. This assignment can be done manually, however automatic **IP Address Management (IPAM)** is desirable to reduce the effort of setting up container networks. Such a system should be able to automatically manage IP address pools belonging to each of the networks used, and assign containers a unique address upon start. Docker has a built-in IPAM system that can automatically manage IP pools even when deploying on clusters. However, to date, centralized IPAM only works when using the overlay driver [mac]. When using other plugins like **macvlan** there is no IPAM cooperation between the nodes, which forces IPAM to be performed on a node basis. It is therefore up to the user to guarantee correct configuration, by manually allocating IP pools on each node. Third-party drivers typically provide their own IPAM systems.

When deploying Docker containers on clusters, orchestration tools are essential to automate configurations and manage the resources in a fair and efficient way.

Docker comes with its own **Swarm mode** [swab] cluster management tool, which allows the user to deploy **services**. A service can be a single container, or a collection of replicated containers that execute the same image. In Swarm mode, Docker allows the user to group together related services into **stacks**. When in Swarm mode, nodes can be either workers or managers. Managers are responsible for maintaining status information about deployed services and stacks, as well as coordinating with each other on service life-cycle decisions. To facilitate deployment of **stacks**, these can be described as Docker **compose files**. Compose files allow the user to specify the services that need to be launched, together with extra information such as where services should get their configurations from, what networks they should be attached to, what computational resources must be guaranteed, and even on what specific nodes they should be deployed.

Kubernetes [Kub] is a container orchestration tool designed to facilitate deployment of containers on a cluster. Kubernetes is highly modular and is not limited to using Docker as its underlying container platform. Kubernetes makes use of the **Container Network Interface (CNI)** container network plugin architecture instead of Dockers default CNM architecture. For most CNM plugins there are equivalent CNI ones, however Kubernetes limits the way these plugins can be used. Namely each node in the cluster can only have a single CNI configuration for all the containers deployed on that machine, forcing all containers that execute in that node to belong to the same network. This fact complicates deploying containers in multiple isolated networks.

There are other orchestration tools available that support deploying Docker containers on clusters like **Apache Mesos** [Mes] or **Hashicorp Nomad** [Nom]. These tools however are general purpose cluster schedulers and require time consuming and complex configuration before being able to deploy Docker containers with non-standard networking requirements. In Docker Swarm mode it is possible to easily setup networks through client commands and in Kubernetes it is possible to setup networks through simple CNI configuration files.

2.2 Traffic shaping

The Linux kernel provides several techniques to control network traffic at a low level. When socket calls are made, packets are placed into queues for later dispatching to the network interface. The Linux kernel allows the user to modify the default queues, named **qdiscs** (queueing discipline), therefore modifying their behaviour. These functionalities can be accessed through the **tc** command [Hub01]. **qdiscs** can either be simple or **classful**. Classful qdiscs have the advantage of allowing other qdiscs to be attached to them, so that when a packet is dequeued it must traverse the corresponding qdisc hierarchy. Qdiscs are attached to a specific network interface and their scope is limited to that interface.

Several qdiscs are provided in the default Linux kernel, which allow to perform traffic shaping. Important to this work are the Hierarchy Token Bucket (**htb**) [Ber02], Network Emulator (**netem**) [Ste11, Hem05] and the **prio** [Ale01] qdiscs.

htb is an implementation of the token bucket algorithm which allows the user to limit the outgoing bandwidth. It has many advanced features that allow to implement complex

QoS (Quality of Service) rules, however, in this work, we are only interested in using its basic functionality as a token bucket implementation to enforce specific maximum output bandwidths. **netem** is a qdisc that allows to implement several traffic control policies, in this work we are only interested in its abilities to perform packet delay and randomized packet dropping. **Prio** is a classfull qdisc that dequeues packets from multiple classes in the order of their priority. This allows for certain traffic, assigned to a higher priority class, to always be dequeued before traffic on lower priority classes.

Since a single network interface can have multiple qdiscs attached to it, its necessary for the kernel to have a method of matching specific traffic to a specific qdisc (or class in a classful qdisc). In order to achieve that, the kernel offers several **filters** that can be configured through the **tc** command. Important to this work is the **u32** filter [Hub15], which allows traffic to be matched against any field in a network packet, namely the destination IP address. Another important feature of u32 is that its implementation relies on hash tables, that allow matching of traffic against several specific qdiscs to be performed in constant time.

Chapter 3

Related Work

Network experimentation tools and testbeds described in literature can be categorized as either simulators or emulators. In this section we will only cover network emulation systems, since NEED is a network emulation system. Network emulation systems can be distinguished by their operational mode. Some solutions can be characterized as tools while others are dedicated testbeds. Furthermore, some solutions make use of code running in kernel space, while others operate entirely on userspace. More recently, solutions have been created that make use of Linux containers.

			Concurrent	Path	Capabilities				Any	Deployment
Name	**Mode**	**Orchestration**	**deployments**	**congestion**	**B**	**D**	**P**	**J**	**Language**	**unit**
DelayLine [IP94]	User	Centralized	✗	✗	✗	✓	✓	✗	✓	Process
ModelNet [VYW+02]	Kernel	Centralized	✗	✓	✓	✓	✓	✗	✓	Process
Nist NET [CS03]	Kernel	Centralized	✗	✗	✓	✓	✓	✓	✓	Process
NetEm [Hem05]	Kernel	*(N/A: single link emulation only)*			✓	✓	✓	✓	✓	Process
Trickle [Eri05]	User	*(N/A: single link emulation only)*			✓	✓	✗	✗	✓	Process
EmuSocket [AV06]	User	*(N/A: single link emulation only)*			✓	✓	✗	✗	✓	Process
Dummynet [CR09]	Kernel	*(N/A: single link emulation only)*			✓	✓	✓	✓	✓	Process
ACIM/FlexLab [RDS+07]	Kernel	Centralized	✗	✓	✓	✓	✓	✓	✓	VM
NCTUns [WCL07]	Kernel	Centralized	✗	✓	✓	✓	✓	✓	✓	Process
Emulab [HRS+08a]	Kernel	Centralized	✗	✓	✓	✓	✓	✓	✓	VM
IMUNES [PPM08]	Kernel	Centralized	✗	✗	✓	✓	✓	✗	✓	Process
MyP2P-World [RAAN+08]	User	Centralized	✗	✗	✓	✓	✓	✗	✗	Process
P2PLab [NR08]	Kernel	Centralized	✗	✗	✓	✓	✓	✗	✗	Process
Netkit [PR08]	Kernel	Centralized	✗	✓	✓	✓	✓	✗	✓	VM
DFS [Tan09]	User	Centralized	✓	✗	✓	✓	✓	✓	✗	Process
Mininet [LHM10]	Kernel	Centralized	✗	✓	✓	✓	✓	✓	✓	Process
SliceTime [WSVL+11]	Kernel	Centralized	✗	✓	✓	✓	✗	✗	✓	VM
Mininet-HiFi [HHJ+12]	Kernel	Centralized	✗	✓	✓	✓	✓	✓	✓	Container
SplayNet [SRF13]	User	Decentralized	✓	✓	✓	✓	✓	✗	✗	Process
MaxiNet [WDS14]	Kernel	Centralized	✗	✓	✓	✓	✓	✓	✓	Process
Dockemu [TCB15]	User	Centralized	✗	✗	✓	✓	✓	✓	✓	Container
EvalBox [SW15]	Kernel	Centralized	✗	✗	✓	✓	✓	✓	✓	Process
ContainerNet [PKVR16]	Kernel	Centralized	✗	✓	✓	✓	✓	✗	✓	Container
NEED	**Kernel**	**Decentralized**	✓	✓	✓	✓	✓	✓	✓	**Container**

Table 3.1: Classification of network emulation tools. The following Link-Level emulation capabilities are considered, B Bandwdidth, D Delay, P Packet loss, J Jitter.

Table 3.1 shows a comprehensive comparison of existing systems. Throughout the rest of this section we present a comparison of some of the most common network emulation systems, and also the ones that are most related to our work.

3.1 Userspace based systems

One of the first userspace tools to be described in literature is **Trickle** [Eri05]. Trickle makes use of the linking and preloading functionality of Unix based systems to insert its code between unmodified binaries and the libraries that provide the sockets Application Programming Interface (API). By doing so it can perform both bandwidth limiting and delay, before calling the actual underlying sockets API. The main advantage of this tool is that it runs exclusively in userspace, and therefore requires only a simple configuration step to apply specific rules for individual processes. The main disadvantage of this approach is the lower accuracy of the emulation, when compared to approaches that operate at lower levels in the network stack. Furthermore, although multiple instances of trickle can cooperate, setting up a multi host system to emulate large networks involves a lot of manual configuration since there is no central deployment control system. To a lesser extent, this approach also has the limitation of not working with statically linked binaries.

Splaynet [SRF13] is an extension of the SPLAY [LRF09] distributed systems development and evaluation framework. Splaynet works entirely in userspace and can perform network emulation of arbitrary network topologies, deployed across several physical hosts in a fully decentralized way. Furthermore, Splaynet supports deploying several experiments simultaneously, as long as the underlying physical infrastructure can provide enough resources. Splaynet does not create processes for emulating the intermediate networking devices like routers or switches. In order to emulate deployment conditions, Splaynet relies instead on graph analysis and distributed congestion emulation algorithms, effectively collapsing the inner topology and delivering packets directly from one emulated host to the destination host. The main advantages of Splaynet are its scalability, due to the fact that the emulation is entirely decentralized, and its ability to run multiple experiments simultaneously while maintaining precision and accuracy in the desired deployment conditions. Splaynet however requires the user to be familiar with the Lua programming language and the SPLAY framework, since the experiments must be described as Lua scripts using SPLAY framework functionality.

3.2 Kernel based systems

Dummynet [Riz97] was one of the first network emulation tools based on kernel modifications. Dummynet allows the user to limit bandwidth and introduce delays and packet loss on a specified network interface. Originally using **dummynet** required using a custom kernel, however nowadays it is available within the default FreeBSD and MacOS kernels[dum02]. The fact that **dummynet** operates at a low level on the sockets stack allows it to achieve better accuracy than userspace implementations of traffic shapers. Although **dummynet** does not implement the functionality to emulate entire network topologies by itself, it is used by many full network emulators like **Modelnet** [VYW+02][VGVY09].

Modelnet is an Internet emulation environment that allows the deployment of unmodified application software on a scalable emulation testbed. Modelnet allows arbitrary software (and operating systems) to run on **edge nodes**. All communications are routed through a set of **core routers**. Core routers are dedicated machines running a modified FreeBSD kernel, that cooperate to emulate the properties of the desired target network before relaying the packets back to the destination edge nodes. More recent developments to Modelnet [VGVY09] have also implemented features such as time dilation to emulate faster links on slower physical links. In order to maintain the ability to run unmodified software this feature was implemented by modifying a hypervisor to manipulate the guest operating system's timers. Modelnet aims at being a complete environment for accurately emulating large network topologies. The ability to scale lies in the cooperation of an arbitrary number of core routers. In order to perform efficiently, these core routers need to run on powerful hardware, and the introduction of more core routers can incur in some overhead, due to the necessity of cooperation between them, whenever a packet has to traverse multiple core routers before being delivered to the destination edge node. The main disadvantage of this approach is the necessity of running on a customized testbed.

Emulab [HRS+08b][BJL+02] is a network emulation testbed that allows to deploy emulated topologies through a globally distributed and heterogeneous testbed. Emulab allows for experiments to use unmodified software and even user-provided operating systems. The method of network emulation is similar to Modelnet, but Emulab can also leverage **tc** (on Linux) or **dummynet** (on BSD) to help shape the traffic at the edge nodes directly, which helps reducing the number of hosts needed for experiments. The main advantage of the Emulab testbed is its ability to deploy large topologies across shared clusters, while maintaining the user requested resource allocation, and the ability to perform this scheduling optimally. The Emulab testbed management is hierarchical, and users need to apply for accounts to run experiments on the shared cluster. Its graph coarsening technique is similar in principle to the NEED approach for collapsing the topology.

Mininet [LHM10] is a widely used tool to emulate network topologies on a single host. Mininet makes use of lightweight virtualization mechanisms provided by the Linux kernel to emulate separated network hosts. Similarly to Docker, it creates virtual Ethernet pairs running in separated namespaces and assigns processes to those restricted namespaces. By leveraging this technique, Mininet can emulate up to hundreds of networked hosts (instances) on a single physical host. The main advantages of Mininet lie in how easy it is to create arbitrarily large network topologies, aided by both a Python API and a Graphical User Interface (GUI). Mininet emulates actual switches and routers as instances of their own (running their own processes), and allows Software Defined Networking (SDN) switches and routers to be deployed and managed in real time. This allows for a smoother transition from an emulated environment to a physical deployment. Mininet has evolved beyond the original cited work, and current versions are able to perform traffic shaping between instances through the **tc** command by leveraging the **htb** and **netem qdiscs**. Mininet is designed to experiment with network topologies, accurately emulating all aspects of a real world topology, and is therefore an invaluable tool to test topology configurations. Although Mininet can scale to hundreds of instances in a single physical host, scalability is reduced if resource intensive applications are deployed. This fact hinders its usability for testing large scale distributed systems.

Maxinet [WDS14] is an extension of Mininet that supports deployment to a cluster of worker hosts. Maxinet supports the same functionality of current versions of Mininet, and also supports deployment of hosts as Docker containers. Maxinet extends Mininet by tunneling links that cross different workers, so that switches placed on different workers can communicate. Maxinet aims at allowing emulation of very large scale SDN networks, however it has the drawback of forcing all emulated hosts that connect to the same switch to be deployed on the same worker as the switch.

Dockemu [TCB15] is a network emulation tool based on Docker containers. The main advantages of this tool lie in the simplicity of setting up experiments, and the fact that it can emulate various types of layers 1 and 2 links. To emulate different link types Dockemu makes use of **NS-3** [NS3], a network simulation tool. Dockemu is therefore appropriate for experiments that seek accuracy in emulation of link types other than Ethernet, namely wireless links. The main disadvantage lies in the fact that the tool is intended to run on a single host, and therefore unsuitable for experiments with large network topologies.

3.3 Summary

Single link emulation systems like **dummynet** and **netem**, although limited in capabilities by themselves, are a solid building block for more complete network topology emulators. The complete topology emulation systems presented above are either limited to executing on a single machine, therefore unable to scale to large topologies with resource intensive applications, or have high entry barriers, such as requiring dedicated testbeds or impose limitations on what kinds of applications that can be tested.

To the best of our knowledge, NEED is the only system that can be used to deploy container-based unmodified applications over emulated topologies without any centralized node, supporting a rich set of emulation features and still providing emulation accuracy on par with existing state-of-the-art systems.

Chapter 4

Architecture

In this section, we describe the architecture of NEED and discuss the design of its components. NEED is a network emulation system that leverages Docker to emulate arbitrary network topologies on a cluster of physical machines. The main novelty of this work lies in proposing a solution that combines the usage of containers, for deploying the applications to a cluster, with the usage of techniques for performing point-to-point emulation in a decentralized way. In order to achieve such a system, the following challenges had to be addressed.

- We needed a way to describe any arbitrary network topology.
- We had to be able to turn that topology description into a Docker deployment.
- We had to develop a way of enforcing the limitations of the network topology at each container in a decentralized manner. Latency, jitter, packet loss rate and available bandwidth must be respected. Also bandwidth congestion at inner nodes of the topology must be accurately emulated.
- We had to guarantee that that the emulation is precise in a distributed environment using a decentralized architecture.

4.1 Overall architecture

To address the above mentioned issues, we developed four main software components. The four components are the deployment generator, the emulation core, the tc abstraction layer, and a supervisor dashboard. Figure 4.1 illustrates the architecture of our solution, and the workflow for deploying an experiment.

The deployment generator takes as input an XML description of the desired network topology and produces a Docker Swarm deployment Compose file in YAML. A more thorough description of this component can be found in Section 5.1.

The emulation core should execute inside each of the deployed containers and run side-by-side with the applications under experimentation. This component should take as input the same network topology description used to generate the deployment. Also, this component has the responsibilities of setting up the initial network conditions, monitoring outgoing communications, cooperating with other instances to maintain a global view of the emulated network infrastructure, and modifying the network conditions in real time to match the limitations of the emulated topology. A more thorough description of this component can be found in Section 4.3.

The tc abstraction layer is a library that hides all the interactions with **tc**. It provides a short, high level API for setting network conditions and monitoring traffic.

The supervisor dashboard is a web application that provides a GUI (Graphical User Interface) for starting and cleanly shutting down the experiments, as well as provide monitoring information about the deployed experiment.

Beyond these components, a template was developed to facilitate extending existing Docker images of applications to undergo testing with the emulation core.

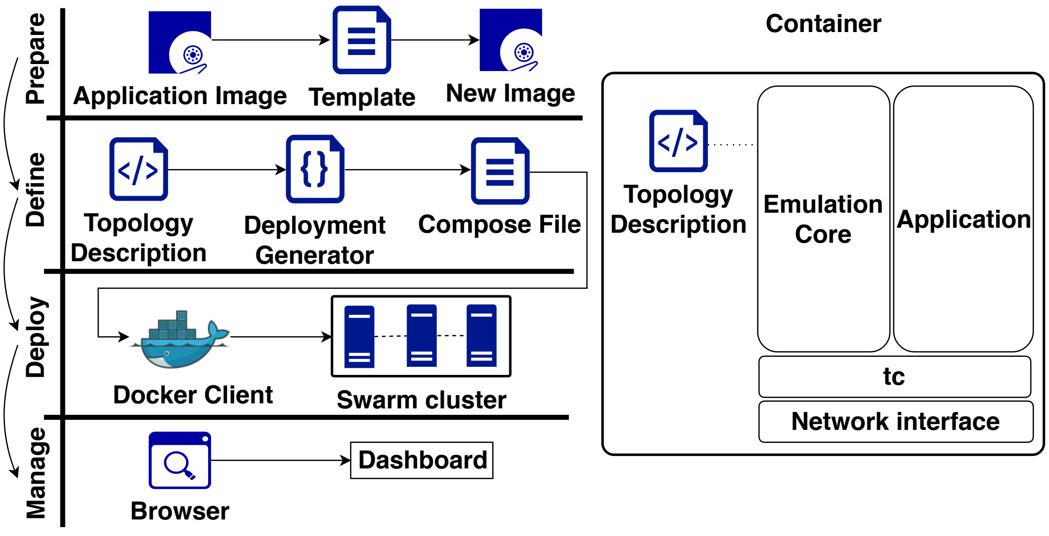


Figure 4.1: NEED architecture and workflow

The workflow to get an experiment running under NEED is the following.

1. Prepare the application images by extending them to include the emulation core.

2. Define the desired topology and use the deployment generator to turn it into a Docker Swarm deployment description. Customize the deployment if necessary.

3. Deploy the experiment with Docker.

4. Manage the experiment through the Dashboard.

We describe in greater detail the architecture of the main components in the next sections.

4.2 Topology specification

To emulate a given network topology we need to have a way of describing it that facilitates both its specification by the user and the parsing by our tools. That description must contain a graph that describes the desired network and its characteristics, as well as a description on how to execute the applications under experimentation. Given a topology description, the deployment generator should be able to produce a valid Docker Swarm deployment in the form of a Docker compose file, that is ready to enable the emulation.

We have decided to create our own specification written in XML that can accommodate the description of the network graph with the parameters we want to emulate, and that also takes into consideration the particularities of deploying applications as Docker containers. The specification is based on the one used in ModelNet.

The specification is composed of **services**, **bridges** and **links**. Services are a group of containers running the same Docker image, and correspond to the same term in Docker terminology. Bridges correspond to generic networking devices that can bridge together multiple links. Incoming connections on a link can be forwarded to any other link attached to the same bridge.

Listing 4.1 shows a short example of a topology specification, and Figure 4.2 shows the resulting network graph.

Listing 4.1: Example of a Topology Description

```xml
<?xml version="1.0" encoding="UTF-8"?>
<experiment>
    <services>
         <service name="c1" image="client:latest"/>
         <service name="sv" image="nginx:latest"
          replicas="2"/>
    </services>
    <bridges>
        <bridge name="s1"/>
        <bridge name="s2"/>
    </bridges>
    <links>
        <link origin="c1" dest="s2" latency="10"
```

```
14          upload="10Mbps" download="10Mbps"
15          jitter="0.25" network="swarm-mv"/>
16         <link origin="s2" dest="s1" latency="20"
17          upload="100Mbps" download="100Mbps"
18          jitter="0.25" drop="0.001"
19          network="swarm-mv"/>
20         <link origin="s2" dest="sv" latency="5"
21          upload="50Mbps" download="50Mbps"
22          jitter="0.25" network="swarm-mv"/>
23     </links>
24 </experiment>
```

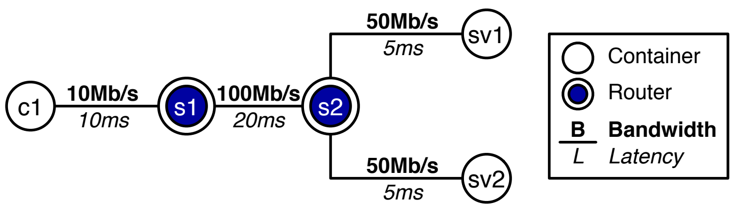


Figure 4.2: Example Topology Graph

Bridges are specified with the **bridge** tag, and must have a unique name attribute. (Example in line 10 of Listing 4.1)

Services should use the **service** tag, and must be specified with attributes defining a unique name, and a valid Docker image. Optionally, the following attributes can also be specified, replicas, command, share, supervisor and port.

The replicas attribute allows the user to specify how many replicas of the same service should be created, when absent this attribute defaults to 1. The command attribute can be used to change the command (in Docker terminology) that is passed to the container. When absent the containers use their default command. The share attribute is a boolean value that only makes sense to use when there are multiple replicas. When set to true, it specifies that the replicas should share the same link attached to them. When set to false the links attached to the replicas are duplicated so that each replica gets its own link. This attribute defaults to true. Finally the supervisor attribute is a boolean value that indicates that this service is a supervisor service. Supervisors are a plugin architecture for NEED that allow to extend experiment logic, a complete overview of this feature is available in Section 4.5. The port attribute indicates the supervisor port that should be forwarded to the outside network, so that users can interact with the supervisor even if the experiment runs on an isolated network.

Edges in the network graph should be specified using the **link** tag. By default links are unidirectional. A link must always have the following attributes, origin, dest, latency, upload and network. The origin and dest attributes must be filled with valid names that correspond to previously declared services or bridges. The upload attribute must be filled with a value for bandwidth capacity. The following units are accepted bps, Kbps, Mbps and Gbps. This bandwidth capacity applies to connections from the origin to the destination. The latency attribute must be filled with an integer latency value specified in milliseconds. The network attribute must be filled with the name of an already existing Docker network with Swarm scope, that must be available at all nodes in the Swarm cluster. Services attached to this link will be attached to the network it specifies. Its important to note that in the current version of NEED, support for multiple networks is not yet implemented. This link attribute is present for supporting future developments. Optionally a link can also be specified with the following additional attributes, drop, jitter and download. The drop attribute should be filled with a float value in the range 0.0 to 1.0 that specifies a packet loss rate for the current link. The jitter attribute should be filled with a float value indicating a standard deviation. This will cause the latency of that link to follow a normal distribution around the specified latency attribute and with the indicated standard deviation. Finally the download attribute indicates that the current link should be bidirectional, and should be filled with a bandwidth capacity in the same way as the upload attribute. The indicated capacity will be enforced to connections from dest to origin. It is important to note that internally to NEED all links are unidirectional, so declaring a link with the download attribute, will cause the creation of two identical links in oposite directions, that share the same attributes except for the bandwidth capacity.

4.3 Emulation core architecture

The emulation core is the main component of our solution. It executes inside every container in parallel with the application undergoing experimentation. The emulation of the network is performed in a fully decentralized way, therefore the emulation core is responsible for parsing the topology description, creating all the local tc infrastructure to enforce the network limitations and cooperate with all the other instances in order to maintain an accurate emulation.

The emulation core executes only on services. This is because bridges do not really exist in the emulation. The specified topology is effectively collapsed, into a different topology where the services are directly connected to all the other services reachable in the original topology. The properties of the original topology however need to be preserved.

In order to do so, an analysis of the original topology graph must be performed. In this analysis we calculate the properties of the paths that connect a service, to all the other reachable services. Given a path $\mathcal{P}$ composed of links $\mathcal{P} = \{l_1, l_2, \ldots, l_n\}$, the latency, jitter and packet loss rate properties of the path can be computed as follows.

$$Latency(\mathcal{P}) = \sum_{i=1}^{n} Latency(l_i) \tag{4.1}$$

$$Jitter(\mathcal{P}) = \sqrt{\sum_{i=1}^{n} Jitter(l_i)^2} \tag{4.2}$$

$$Loss\ rate(\mathcal{P}) = 1.0 - \prod_{i=1}^{n} (1.0 - Loss\ rate(l_i)) \tag{4.3}$$

$$\max Bandwidth(\mathcal{P}) = \min_{\forall l_i \in \mathcal{P}} Bandwidth(l_i) \tag{4.4}$$

An example of the previously described topology and its collapsed counterpart is shown in Figure 4.3

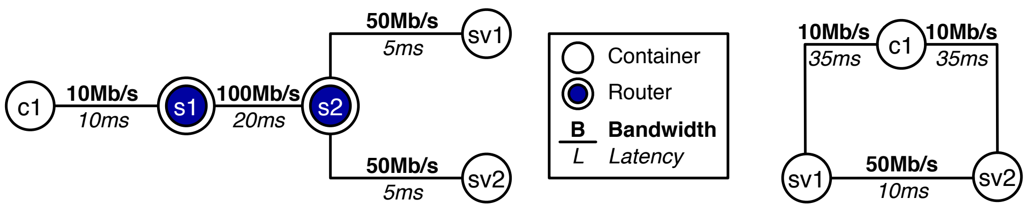


Figure 4.3: Example of a topology and its collapsed counterpart

The latency of a path can be calculated by simply adding the latencies of all its links. Calculating jitter and packet loss rate is more complicated. Jitter for a given link is specified as a standard deviation following a normal distribution centered on the specified latency. In order to calculate the standard deviation of the accumulation of several links, we must add together the variance, which can be obtained by squaring the standard deviation. Packet loss rate on a given link is specified as a probability, hence to calculate the packet loss rate of a path we must take the product of the packet loss probabilities of all its links. Care must be taken however when calculating the product of probabilities, if a given link on a path has 0% packet loss probability then the product for the entire path would have 0% loss probability even if other links have greater probability. To solve this we use negated probabilities. The previously discussed properties remain constant throughout the experiment and the collapsed topology is equivalent to the original with respect to these properties. The amount of available bandwidth, can however change throughout the duration of the experiment, as flows from different services compete for bandwidth on shared links. These shared links where congestion can occur are not represented in the collapsed topology, therefore it is not equivalent to the original topology with respect to the bandwidth properties. The maximum bandwidth available on a (collapsed) path will be the same as the link in the path that has the minimum available bandwidth. This allows us to set an initial upper bound on the bandwidth of each path. The available bandwidth in a real world scenario is dictated by bottlenecks where data arrives faster at a networking device then it can be sent. This situation eventually leads to packets being delayed or lost at inner network elements such as switches or routers. Protocols such as TCP know how to react to this situation, and adjust their throughput in a way that all flows competing for bandwidth get a fair share. In our solution however, we want packet loss and delay to be fully controlled by the user. If a user specifies that they want 0% packet loss on all the links in the topology, then no packet loss should occur even in the case of congestion. This decision can affect the accuracy of the emulation, since we are not dropping packets that would be dropped in the real world due to the operation of the inner network elements, however we gain in reproducibility of the experiments, since the real world packet loss behavior is non deterministic. Instead of relying on this behavior to indicate to the applications that a throughput adjustment is necessary, we rely on a model that allows us to calculate a fair share of the available bandwidth for each competing flow. We then use the results of the model to enforce a new maximum available bandwidth for each path. The model used is the Round-Trip Time (RTT) Aware Min-Max model described in [Kel, MR02]. This model maximizes bandwidth utilization on a link, while giving a share to each flow that is proportional to its round trip time. To calculate the fair share of a flow the following formula is used, where $f \in \{f_1, f_2, \ldots, f_n\}$ are flows using bandwidth on a link.

$$Share(f) = \frac{\frac{1}{RTT(f)}}{\sum_{i=1}^{n} \frac{1}{RTT(f_i)}} \tag{4.5}$$

The formula above gives us a percentage of the maximum bandwidth that any given flow is allowed to use, however it does not guarantee that the available bandwidth on a link will be fully utilized. For example a flow might be using less bandwidth on the current link than the share it is given, because it is further limited by another link in the topology. Therefore a maximization step is required, where we increase the share of the other unrestricted flows. This increase has to be proportional to the original shares given by the formula above. The emulation core must periodically employ the model, taking into consideration the original topology, and change the enforced maximum bandwidth on the collapsed paths throughout the entire duration of the experiment.

4.4 tc abstraction layer architecture

The tc abstraction layer is a library that provides an interface to setup initial networking conditions, retrieve network usage, and modify maximum bandwidth on paths.

The exposed API is the following:

Listing 4.2: tc abstraction layer API

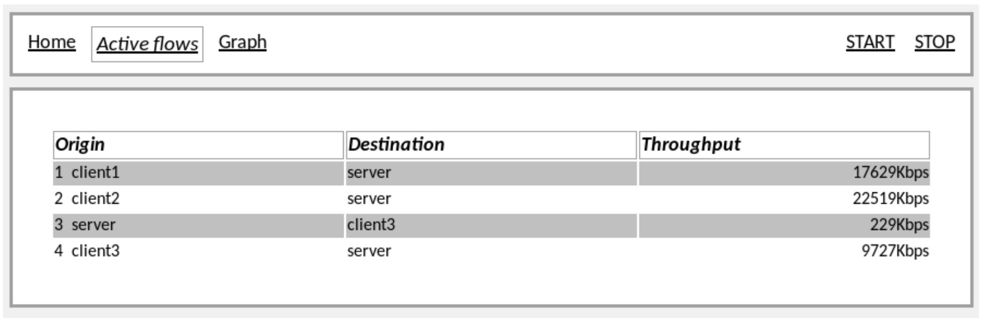
```
void init(short controlPort);
void initDestination(std::string ip, int bandwidth, int latency,
                     float jitter, float packetLoss);
void changeBandwidth(std::string ip, int bandwidth);
void updateUsage();
unsigned long queryUsage(std::string ip);
void tearDown();
```

The **init** function initializes the **tc** infrastructure. It receives as an argument a port address for the control port, used to exchange metadata and commands between emulation core instances. Traffic on this port has to be able to bypass any restrictions imposed on the other traffic. The **initDestination** function sets up the **tc** infrastructure for enforcing delay, packet loss and bandwidth throttling on all traffic directed to the specified IP address. The **changeBandwidth** function modifies the **tc** infrastructure changing the maximum allowed bandwidth to the specified IP address. The **updateUsage** function takes a snapshot of the number of bytes sent to each previously setup destination. The **queryUsage** function retrieves this information for a specific destination. Finally the **tearDown** function destroys all the created **tc** infrastructure.

4.5 Supervisor dashboard

The supervisor dashboard is a web interface that allows the user to control and monitor the experiment. The dashboard allows the user to start and stop the experiment. It also displays the current status of all service instances in the experiment and the current active flows in the topology in real time. It also provides the user with a graphical representation of the topology. The dashboard is made available through HTTP, and is intended to be used through a browser, although starting and stopping the experiments can also be performed through simple HTTP GET requests directed at specific URLs.

The supervisor dashboard makes uses of the supervisor plugin architecture built into NEED. Supervisors are special services that can be registered in the XML topology description (see


Home Active flows Graph START STOP

	Origin	Destination	Throughput
1	client1	server	17629Kbps
2	client2	server	22519Kbps
3	server	client3	229Kbps
4	client3	server	9727Kbps

Figure 4.4: Supervisor dashboard screenshot showing the active flows.

Section 4.2). Supervisors can issue commands to the emulation core instances to begin executing and also to cleanly shutdown an instance. Supervisors will also receive metadata from every emulation core instance indicating active flows, and their bandwidth usage. Upon shutdown of an experiment, the dashboard reports a percentage of metadata lost during the experiments. This value is intended to give the user an indication of how accurate the emulation was.

The need to create the dashboard arose from a bug discovered in the Linux kernel that leads to memory corruption of the host when experiments were not cleanly shutdown. The bug is triggered when tc qdiscs are deleted while packets are being placed on them. Since this situation would occur often if the user removed the experiment with Docker Swarm commands, we had to provide a way of cleanly shutting down the experiment, before the removal of the deployment could be performed. Figures 4.4 and 4.5 show examples of the interface that is presented to the user by the dashboard application.

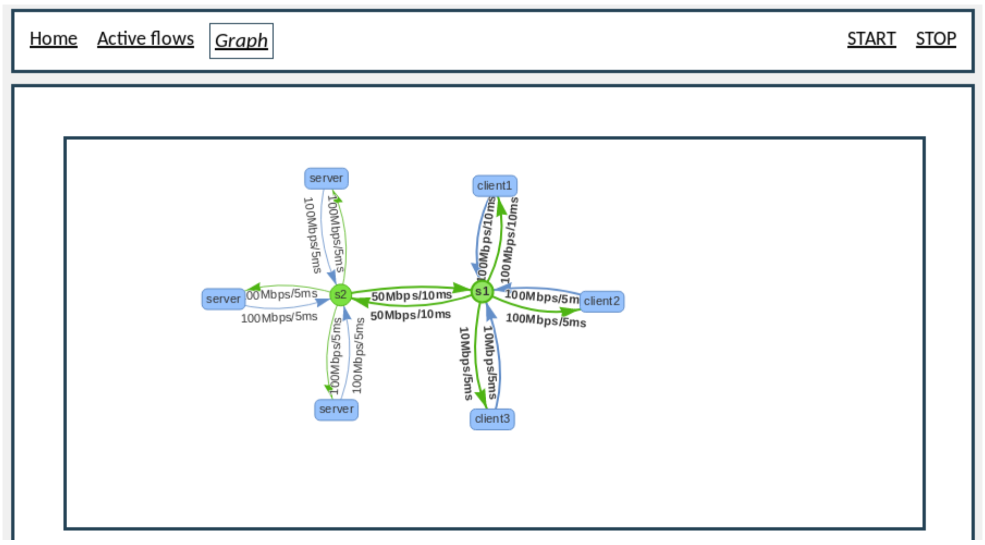


Figure 4.5: Supervisor dashboard screenshot showing the currently deployed topology.

4.6 Summary

Overall NEED is split into the four main components described previously. The topology specification and the deployment generator were designed to integrate seamlessly with Docker workflows, and to be easily adapted to work with other container orchestration systems. The tc abstraction layer was designed to be a simple interface for enforcing point-to-point network characteristics. The emulation core was designed to operate in a fully decentralized fashion, and implement the logic necessary to perform the emulation with a collapsed topology, that is equivalent to the original. Finally, the supervisor plugin architecture was designed to allow for extensions to experiment logic, giving the user control and real-time information on the deployed experiments.

Chapter 5

Implementation

In this section, we discuss the implementation of the various components of NEED. We go into detail on how the various components were implemented and discuss the reasons behind the decisions that were taken.

5.1 Deployment Generator implementation

The deployment generator is a Python script that accepts as input an XML file containing a topology description that must follow the specification detailed in Section 4.2. It then writes to the standard output a Docker Swarm compose file specification written in YAML.

An alternative would be to use the Docker API to immediately deploy the experiment on the cluster, however in many real applications further customization of the compose file is necessary, like for example setting application configurations, or mounting external volumes. The deployment generator works by parsing the XML file and building a graph structure. Afterwards a compose file generator Python class uses that graph to generate the YAML compose file. An alternative would be to translate the XML directly to YAML, however the current approach has two advantages. First it reuses the same code used by the emulation core (described in Section 4.3) to generate the internal graph data structure. Secondly it allows for more easily writing code to generate deployment descriptions for other Docker orchestration systems.

5.2 tc abstraction layer implementation

The tc abstraction layer is implemented as a library, written in C++ for performance reasons. The tc infrastructure set up by the tc abstraction layer has the structure described in Figure 5.1. First a **prio qdisc** is set up with two **classes**, that correspond to queues with differen priority. The highest priority queue will be used for traffic on the control port. The lowest priority queue will be used for the remaining traffic, and contains a hierarchy of **qdiscs** for enforcing the limitations for each destination. For each destination an **htb qdisc** is created and attached to the lowest priority **prio class**. The **htb qdisc** is responsible for enforcing the bandwidth throttling. The **htb qdisc** contains a **class** to which other **qdiscs** can be attached. To the **htb class** we attach a **netem qdisc** that will be responsible for enforcing the delay and packet loss rate. Using the **u32** filter, we setup two filters. One will match the traffic on the control port, and direct it to the high priority **prio** class. This allows traffic on the control port to bypass the shaping performed by the **htb** and **netem qdiscs**. The second filter is a two level hashtable that will match against the destination IP address of packets, and will direct them to their corresponding **netem qdisc**. Traffic directed to the **netem qdisc** will first be subjected to the **netem** rules, when it is dequeued from **netem** it will be queued on the parent **htb qdisc**, before being queued on the parent low priority **prio** class.

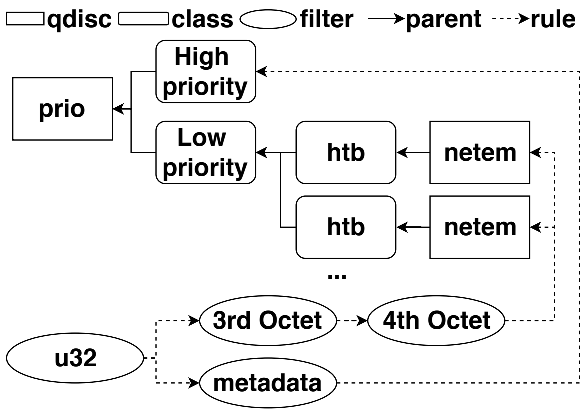


Figure 5.1: Qdisc and filter hierarchy created by the tc abstraction layer

The initialization steps done by **init** and **initDestination** (see Listing 4.2) are performed by executing the **tc** binary. This is acceptable since these operations are only executed once at system initialization. The **changeBandwidth** and **queryUsage** (see Listing 4.2) functions however are called very frequently, and their performance is critical to the correct functioning of the emulation core. The process of forking, executing the **tc** binary and then parsing its output was deemed too expensive. Instead these functions make use of netlink [net17] sockets to communicate with the kernel directly, and perform their function with minimal overhead.

5.3 Emulation core implementation

The implementation of the emulation core, is split into two parts. The program itself is written in Python, and implements the algorithms required to manage the emulation. The emulation core makes use of the tc abstraction layer to interact with **tc** and enforce the calculated network limitations. (See Figure 4.1.) The emulation core is split into two main execution steps, an initialization step and the emulation loop. When the emulation core is launched, it will start by parsing the XML topology description file and build an internal graph data structure. The code to perform this is shared with the deployment generator. After the graph is built, the emulation core then resolves the names of all services to obtain their IP addresses. This is done using the Docker Swarm built-in name resolution system. Once the addresses of all services are known, the emulation core must find the service instance for which it is responsible. With this information, the emulation core executes an implementation of Dijkstra shortest paths algorithm to find the shortest paths between the instance it is responsible for and all the other reachable service instances. In the case of ties one of the alternative paths is picked deterministically, based on the order of links on the topology description. With the shortest paths calculated, the emulation core proceeds to calculate the properties of the collapsed topology using the methods described in Section 4.3. The properties of the paths are then applied using the tc abstraction layer. The emulation core keeps the information about the links in the original topology, that correspond to each path, as well as the original graph in memory, as this information will be crucial for calculating bandwidth congestion restrictions. As soon as the path properties are applied in **tc**, emulation is ready to begin, and the emulation core moves over to the emulation loop.

A pseudo code description of the emulation loop can be seen in Listing 5.1.

Listing 5.1: Emulation core main loop pseudo-code

```
pool_period = 50 #50ms is the default
iteration_count = 2 #2 iterations are the default
while True:
    for i in iteration_count:
        clear_local_state()
        check_active_flows()
        broadcast_flows()
        sleep(pool_period)
    recalculate_bandwidths() # See Section 4.3
    clear_global_state()
```

The loop starts by clearing all state regarding active flows produced by the service instance it is responsible for. Then the tc abstraction layer is queried for the amount of data sent to other service instances. That information allows us to calculate how much bandwidth is currently being used on each path, and therefore also on each link of the original topology. We will refer to this information as the emulation metadata. The metadata is saved locally and then broadcasted to all the other emulation core instances. The broadcasting of the metadata is offloaded to a process pool so that it can be performed in parallel. There is a separate thread running that is responsible for collecting metadata from other emulation core instances. The program then sleeps for a default of 50 milliseconds. This period, referred to as pool period, is configurable and should allow enough time to pass for metadata from other instances to arrive and be processed. The steps described above get executed twice, by default, before the emulation core uses all the collected metadata to decide on new bandwidth limits for each path. The number of times these steps are executed is referred to as iteration count, and it is a configurable variable. After this is done all information on active flows is deleted. The calculations involved in deciding new bandwidth limits are also offloaded to another parallel process. This is done to minimize the impact of lengthy calculations on large topologies on the intervals between sending metadata. To decide on new bandwidth limits for the paths, the RTT Aware Min-Max model is employed, which has been described in Section 4.3.

The algorithm presented above can suffer from two problems that can cause the calculation of bandwidth shares to produce a wrong value. The first problem is metadata packet loss. Each instance is broadcasting the flows it is producing to other instances using UDP packets. We assume that the network supporting the experiment has very low packet loss rates, however the loss of a single packet could be enough for the system to misbehave for a short amount of time. This is because the flows contained in the lost packet would not be taken into consideration when employing the RTT Aware Min-Max model. The second problem is more subtle but leads to the same incorrect behavior. If packets are delayed, they might arrive too late at a given instance to be considered for the RTT Aware Min-Max model. This can happen since the windows of time between cycles of the emulation loop are not guaranteed to be constant across all instances. Also, under heavy networking load it is common to observe such packet delays. Using TCP would prevent packets from being completely lost, as they would eventually be retransmitted. However this would cause delay in the arrival of packets which would bring us to the same situation as when packets are delayed, and therefore arriving too late. To deal with the above mentioned problems we have come up with two techniques that attempt to mitigate the problem and its consequences. When available bandwidth on a given path is decreased, the smaller value is applied immediately. However when it is increased, we can not be sure if the increase is due to a flow effectively terminating, or because a metadata packet was lost or delayed. Therefore, when increasing available bandwidth on a path, we employ an exponential weighted moving average to smooth out temporary spikes. The second method we employ is repeating the broadcast of metadata a number of times controlled by the iteration count variable (2 by default), before taking the available data into consideration for calculating bandwidth restrictions. Under the situation where no data is lost, and delay is not significant we should effectively receive two measurements from each instance before calculating the bandwidth restrictions. In this situation we only consider the last measurement to have arrived. However when packets are lost or delayed, it is much more likely that we have at least one recent measurement to take into consideration. This is a simple fault tolerance scheme that can tolerate $iteration\ count - 1$ omission faults. This method was chosen since more complicated fault tolerance schemes would involve more communication steps which were deemed too expensive. The default value for iteration count of 2 was chosen to minimize the impact on system responsiveness, while allowing for accurate emulation of moderate scale topologies. The default value of the pool period (50 milliseconds) can also be configured by the user, and similarly to iteration count it controls system responsiveness. A smaller value will improve the reaction time of the system, however resource utilization will also increase proportionally.

In Section 4.5 we stated that supervisor services are able to send commands to emulation core instances. Every emulation core instance listens on a TCP port for supervisor commands. The interface provided to supervisors is the following, check if an instance is ready to start emulation, begin emulation and launch the application, stop emulation and terminate the application. The procedure to stop the emulation is the following:

1. Send the application under experimentation a terminate signal.
2. Bring the network interface down.
3. Destroy the previously setup **tc** infrastructure.
4. Bring the network interface back up.

This procedure prevents packets from being inserted into **qdiscs** while the **tc** infrastructure is being deleted, avoiding kernel memory corruption errors, that could otherwise occur. In Section 4.5 we also stated that upon shutdown of an experiment, the supervisor dashboard presented the user with a metadata loss percentage. Whenever the emulation core sends metadata to other emulation core instances a counter is incremented that keeps track of how many packets were sent. Upon receiving a metadata packet from another instance, another counter is also incremented that keeps track of the total number of metadata packets received. When an emulation core instance receives a shutdown command it responds with the value of both these counters. When a supervisor shuts down all instances it is therefore able to sum the values of all counters from all instances, and calculate exactly how many metadata packets were lost. This information serves as an indication for the user, of whether or not the experiment results can be trusted to be accurate.

5.4 Privileged bootstrapping

In order for an application running inside a Docker container to be able to use **tc**, it must be executed with the **CAP_NET_ADMIN** Linux capability [cap17]. Although Docker allows for executing applications in individual containers with user specified capabilities, this functionality is currently not present when deploying to a Swarm cluster [swaa]. Since the emulation core must execute with elevated privileges, the following workaround had to be developed. A special bootstrapping container has to be deployed at every node in the Swarm cluster. This container has access to the local Docker daemon, and uses it to monitor the creation of new containers on that node. When a new container is created that belongs to a NEED experiment, the bootstrapping container uses its access to the docker daemon to launch the emulation core inside the container with elevated privileges. The ability to deploy containers to a Docker Swarm with elevated privileges is expected to be available in a future Docker release, at which point this workaround could be completely removed.

5.5 Docker image template

NEED has to be able to execute in parallel with the application inside every container. In order to do so, the emulation core has to be part of the application container image. There are two possible approaches to achieve that. The first one is providing a NEED image that users can extend with their application. The second one is providing a template for extending an existing application image with the emulation core. We argue that the second approach is more flexible and requires less effort, since we do not force the users to adapt their existing application to work on a potentially different environment.

There are two steps required to extend an image with the emulation core. First the user has to adapt the provided template Dockerfile. The purpose of this Dockerfile is to extend the existing application image by installing the emulation core and its dependencies. An example of this Dockerfile can be seen in Listing 5.2. Then the user must modify a provided script named need-entrypoint.sh with instructions on how to launch their application, so that the emulation core can launch and terminate the application when the experiment starts and finishes. An example of this can be seen in Listing 5.3.

Listing 5.2: Example of extending a third-party Docker image of the Cassandra NoSQL datastore to be used with NEED.

```
FROM cassandra:3.11
ADD need-entrypoint.sh /
RUN apt-get update && apt-get install (NEED dependencies)
RUN mkdir /opt/ && cd /opt && git clone NEED.git && cd NEED && make;
ENV NETWORK_INTERFACE="eth0"
ENTRYPOINT ["/bin/sh", "/need-entrypoint.sh"]
```

Listing 5.3: Example of extending the entrypoint with instructions for running the Cassandra NoSQL datastore.

```
#! /bin/bash
readypipe=/tmp/readypipe
donepipe=/tmp/donepipe

function execute_program{
        # Instructions for starting the application go in here
        /usr/local/bin/docker-entrypoint.sh cassandra -f
}

export -f execute_program

echo "Waiting for the bootstrap code to finish..."
read line <$readypipe;
echo "System has been bootstrapped!"

setsid bash -c execute_program $(printf " %q" "$@") &
pid=$!

read line <$donepipe
echo "Terminating application"
kill -- -$pid

rm $readypipe
rm $donepipe
```

5.6 Summary

The implementation of NEED is split between the deployment generator, the emulation core and the tc abstraction layer. The deployment generator transforms topology descriptions written in XML into Docker Compose files written in YAML, that are ready to deploy the experiment on Docker Swarm. The emulation core executes inside all containers and enforces the characteristics of the emulated topology. To do so, it makes use of the tc abstraction layer, which is a library that interacts with the kernel to allow for modification of point-to-point network properties. Beyond these components, it was necessary to develop a method for executing the emulation core inside the containers with elevated privileges, and templates were developed for easily extending existing application images to integrate with NEED. An example with instructions on how to execute an experiment with NEED is provided in Appendix A.

Chapter 6

Evaluation

We evaluated NEED through a series of micro- and macro-benchmark experiments in a cluster. Further, to validate the soundness of our approach against more realistic scenarios, we compare the behavior of applications running on Amazon EC2 [amaa] and under NEED. Overall, our results show that:

- NEED scales linearly with the number of flows and containers, and has constant cost regarding bandwidth usage.
- NEED emulation accuracy is comparable with other common network emulation systems such as Mininet.
- Running an application with NEED in a cluster or in Amazon EC2 yields similar results.

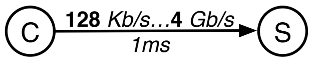
We start by comparing bandwidth emulation accuracy between NEED and other systems (Section 6.1). Next we verify that NEED emulates latency and bandwidth with precision and accuracy in simple topologies (Section 6.2). Next we assess the amount of bandwidth that emulation metadata produced by NEED uses on a given experiment (Section 6.3). Then we assess the scalability of NEED. First we assess the scalability regarding bandwidth congestion scenarios (Section 6.4), and also the ability to execute several experiments in simultaneously. Then we assess the scalability regarding only latency emulation by showing results from executing large scale-free topologies (Section 6.5). Finally, we show that NEED is able to reproduce, in a cluster environment, results obtained on geo-replicated deployments performed on Amazon EC2 (Section 6.6). To this end we reproduce the results of BFT-Smart [BSA14]. And finally, we compare the results of a geo-replicated Apache Cassandra [cas] deployment over 2 continents with a deployment in our cluster.

Evaluation settings. Unless otherwise noted tests were executed on a cluster composed of 4 Dell PowerEdge R630 server machines, with 64-cores Intel Xeon E5-2683v4 clocked at

2.10 GHz CPU, 128 GB of RAM and connected by a Dell S6010-ON 40 GbE switch. The nodes run Ubuntu Linux 16.04 LTS, kernel v4.4.0-127-generic. The tests conducted on Amazon EC2 use `r4.16xlarge` instances, the closest type in terms of hardware-specs to the machines in our cluster. Unless otherwise stated the Docker network driver used was overlay. We use the latest stable versions of Mininet (v2.2.2) and Maxinet (v1.2).

6.1 Link-level emulation accuracy

First we evaluate the accuracy of our bandwidth shaping mechanism under a simple scenario. The topology used consists only of two services, connected by one link. One of the services executes an **iPerf** [ipe] server, the other executes an **iPerf** client. We access accuracy across a range of different target bandwidths, and compare the results with the same experiment executed with Mininet and Trickle. On all cases the **iPerf** client is configured to execute for 60 seconds before terminating, the average throughput reported at the server can be seen in Figure 6.1. The values obtained with NEED and Mininet are very similar. This is because both systems



Link BW	NEED	MiniNet	Trickle (def.)	Trickle (tuned)
Low:				
128 Kb/s	122 Kb/s	123 Kb/s	262 Kb/s	131 Kb/s
256 Kb/s	245 Kb/s	286 Kb/s	472 Kb/s	262 Kb/s
512 Kb/s	490 Kb/s	490 Kb/s	717 Kb/s	525 Kb/s
Mid:				
128 Mb/s	122 Mb/s	122 Mb/s	250 Mb/s	131 Mb/s
256 Mb/s	245 Mb/s	245 Mb/s	493 Mb/s	261 Mb/s
512 Mb/s	487 Mb/s	486 Mb/s	952 Mb/s	518 Mb/s
High:				
1 Gb/s	954 Mb/s	933 Mb/s	1.67 Gb/s	1.00 Gb/s
2 Gb/s	1.91 Gb/s	N/A	1.93 Gb/s	1.97 Gb/s
4 Gb/s	3.79 Gb/s	N/A	4.12 Gb/s	3.61 Gb/s

Figure 6.1: Study of the bandwidth shaping accuracy for different emulated link capacities and tools on a point-to-point topology (above the table).

rely on the **htb qdisc** to perform the bandwidth shaping. It should be noted however that Mininet does not allow imposing bandwidth limits greater than 1Gb/s. NEED does not impose that restriction, and in fact accuracy is always maintained within 5% of error across all ranges of target bandwidths. Although NEED and Mininet use the same **htb qdiscs** for performing bandwidth shaping, NEED configures it differently from Mininet. Throughout the emulation adjustments are made to cope with varying target bandwidths. Mininet on the other hand only configures **htb** at initialization, and is therefore optimized only for bandwidth targets below 1Gb/s.

Finally we also compare results against executing the same experiment with Trickle, a userspace bandwidth shaper. Results using the default settings deviate significantly from the specified throughput rates. In order to achieve accuracy comparable with the other systems, we were forced to tune **iPerf** to use smaller TCP sending buffers. This shows that even though it is possible to achieve acceptable accuracy with userspace tools, the fact that they operate so high on the network stack leads to variable accuracy that depends on application behavior. This does not happen in kernel shapers since those systems operate directly at a network packet level.

6.2 Emulation accuracy in simple topologies

Next we present a series of experiments to access the precision and accuracy of NEED under a more complex topology. The topology we chose to use is represented in Figure 6.2 and consists of 3 servers, 3 clients and 2 bridges.

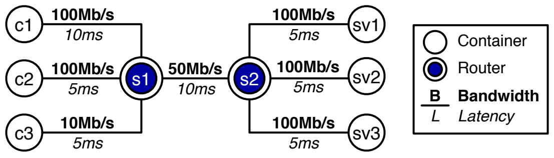


Figure 6.2: Dumbbell topology with 3 clients and 3 servers where the links connecting the clients to switch s1 all have different properties.

The presented topology has several characteristics that make it interesting for testing the behavior of NEED under bandwidth contention. The link between the switches is limited to 50Mbps, which is the main bottleneck of the topology, however client c3 connects to the switch with a 10Mbps link. Furthermore even though clients c1 and c2 have no such bottleneck, the latencies of their links to the switch are different (10ms and 5ms respectively). These characteristics allow us to test all the corner cases of our RTT Aware Min-Max model implementation.

For the first experiment each client pings a server. Ping measures round trip time, as such the expected results are 50ms for client c1, and 40ms for both clients c2 and c3. Measurements were taken from the output of ping, and the results can be seen in Figure 6.3. The results match

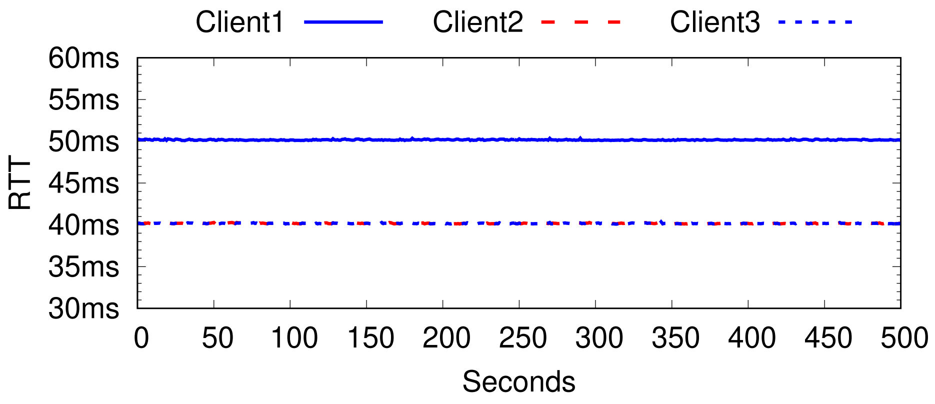


Figure 6.3: 3 Clients **ping** 3 Servers on a dumbbell topology

the expectations. There are small deviations below 1ms of error, that are likely to be caused by the underlying network.

For the second test case each server executes an **iPerf** server, and each client establishes a connection to a different server. **iPerf** is configured to use TCP and will attempt to transmit data at the maximum possible throughput rate in a continuous flow. If we analyze the topology we can see that client c3 connects to bridge s1 through a 10Mbps link, as such client c1 will only be able to achieve a maximum throughput of 10Mbps, since this is its bottleneck link. Clients c1 and c2 connect to bridge s1 through links with 100Mbps of bandwidth capacity, however they share a common bottleneck of 50Mbps at the link that connects bridges s1 and s2. Furthermore the flow produced by client c3 will also use 10Mbps of the capacity of this link. According to the RTT Aware Min-Max model employed, and taking into consideration the RTT of the flows, the flow from c1 should get 28.6% of the available 50Mbps, and the flows from c2 and c3 should receive 35.7%. This would result in a throughput of 14.3Mbps for client c1 and a throughput of 17.85Mbps for both clients c2 and c3. Client c3 however is limited to using just 10Mbps, which leaves 7.85Mbps of spare bandwidth capacity on the link between both switches. The algorithm should therefore scale the allocation for flows from c1 and c2 proportionally to their RTT, which should result in a final allocation of 17.77Mbps for client c1, 22.2Mbps for client c2 and 10Mbps for client c3. The results of running this test case can be seen in Figure 6.4. Measurements were taken by analyzing network traces captured at the servers. The experiment ran for a total of 10 minutes, we discard the first 60 seconds and report the following 500 seconds.

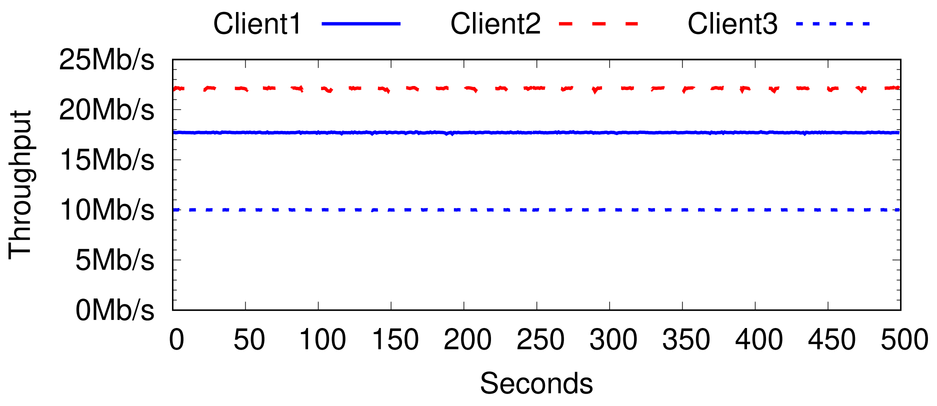


Figure 6.4: 3 clients saturate the bandwidth capacity of a dumbbell topology in NEED

As expected when the 3 clients reach their steady state, they assume the throughput values previously presented. Small deviations can be observed below 1.5% of error.

Next we compare the results of the previous experiment with the results obtained from deploying the same topology in Mininet. As has been previously discussed in Section 4.3, Mininet emulates the switches in the topology, and these switches have limited buffers. Since the output link of switch s1 is capped at 50Mbps and the input links can altogether provide 210Mbps, the buffers at the switch will periodically fill, and packets will be dropped. This packet loss can be observed in the output of **iPerf**. TCP will also react to that loss, which creates fluctuations in throughput. Once again this is the behavior that would be observed on a real network, but it is non-deterministic as multiple executions of the same experiment on Mininet yield very different outcomes every time.

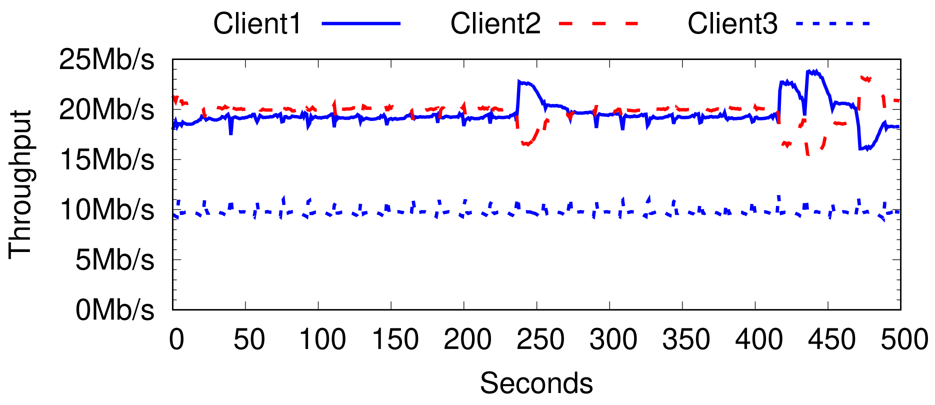


Figure 6.5: 3 clients saturate the bandwidth capacity of a dumbbell topology in Mininet

Figure 6.5 shows one execution of the experiment on Mininet. Although steady state throughput assumes values close to the ones assumed in NEED, there are spikes whenever large quantities of packets are lost, causing TCP to adjust the throughput. NEED does not emulate buffers at the switches, hence preventing by design implicit packet loss from occurring at links with expected 0% packet loss probability.

Implicit packet loss in Mininet only occurs when bottleneck links are saturated, therefore it is expected that both NEED and Mininet behave the same when the bottleneck links are not saturated. To show this we have limited **iPerf** on clients c1 and c2 to produce flows with 20Mbps and 15Mbps respectively. This means that the total bandwidth used by all three clients is now just 45Mbps, 90% of the capacity of the link between s1 and s2. Figure 6.6 shows that in this scenario NEED and Mininet behave exactly the same.

Finally for the third test case with the same topology, we desynchronized the flows so that we can observe how the system reacts when new flows are created and when they cease. The experiment proceeds as follows. Initially, only c1 has an active flow, and hence it should use all the available bandwidth. After 3.5 seconds, c2 starts and thus it will compete for bandwidth over the shared link. At this point, since c2 has a smaller RTT than c1, it should get a proportionally higher share of bandwidth. At second 8, c3 starts and should reach its bandwidth limit of $10Mbit/s$. The bandwidth of the other two clients should get proportionally adjusted to cope with this new competing flow. Finally, c2 and c3 stop allowing c1 to once again use the maximum

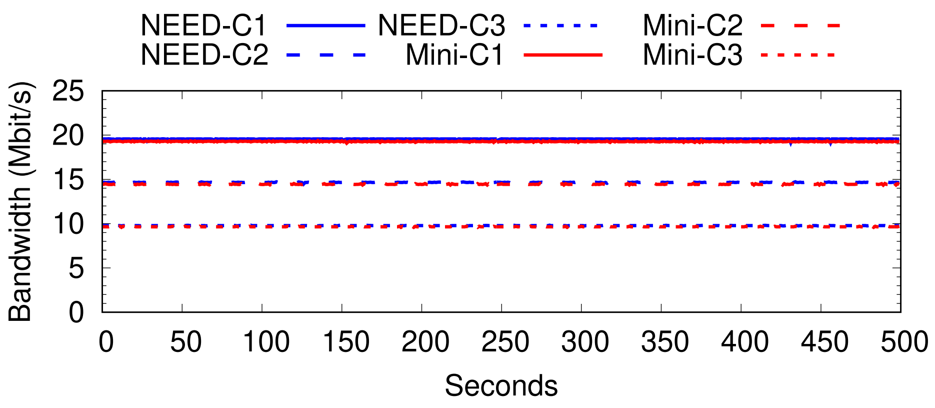


Figure 6.6: 3 clients use 90% of the bandwidth capacity of a dumbbell topology in NEED and Mininet

bandwidth allowed by the topology. The results can bee seen in Figure 6.7. As can be seen,

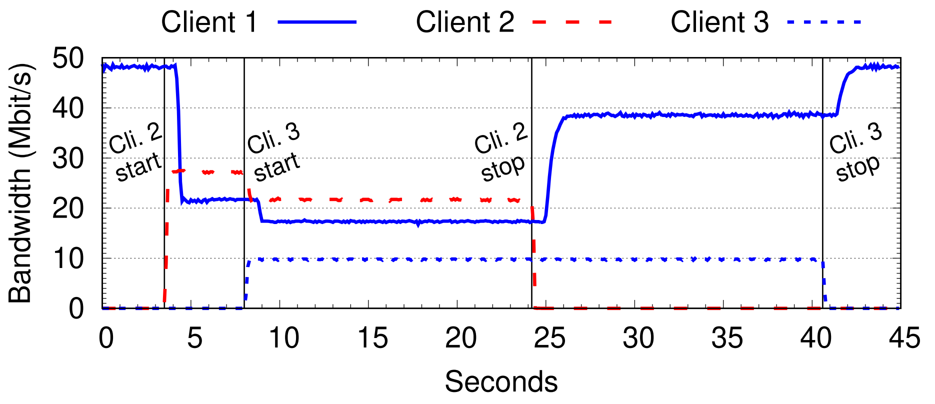


Figure 6.7: Reaction time to throttle 3 flows that vary in time

there is a short interval of time that the system takes to react to new flows. This interval of time has a lower limit of 100ms (the time it takes to complete one cycle of the emulation loop, as described in Section 5.3). We can however observe that it can take as long as 1 second to reach a new steady state. This is in part caused by TCP slow start which causes incremental increases in throughput. NEED takes at least 100ms to react to the increments, leading to further delay in the adjustment. Another observable behavior is the effect of the exponential weighted moving average when the bandwidth limit is increased (whenever a competing flow terminates).

6.3 Metadata bandwidth usage

Bandwidth congestion emulation in NEED relies on a distributed algorithm to calculate new maximum bandwidth rates. This algorithm (described in Section 5.3) takes as input the amount of bandwidth that every currently active flow is using on the topology. We refer to this information as emulation metadata. Every container that is producing flows must disseminate this metadata to every other container running the emulation core. In this section we will explore the costs of metadata dissemination in NEED. Understanding how the flow of metadata works will be crucial for later understanding the scalability limitations of the current design.

In Figure 6.8 we present the total bandwidth that is consumed by metadata under 6 different scenarios.

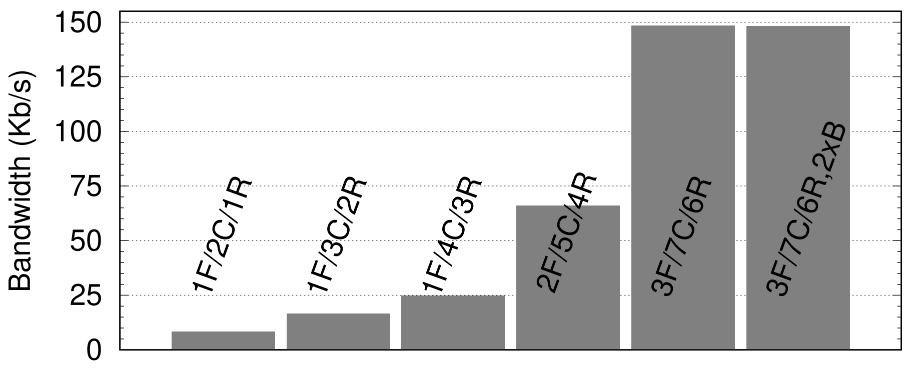


Figure 6.8: NEED metadata bandwidth usage with an increasing number of flows and containers. F:Flow, C:Container, R:Metadata receiver

The scenarios are described by the number of flows that are active, the total number of containers in the topology and the number of containers that are receivers of metadata. This later distinction is necessary since a container does not send metadata to itself. On all scenarios we have containers acting as **iPerf** clients and others are **iPerf** servers. The number of flows is dictated by the number of client containers. In the first scenario we have two containers, a client and a server,that are producing a single flow. Since there are only two containers, there is only one receiver of the metadata produced by the client container. The total amount of bandwidth used by metadata in this scenario is 8.26Kbps. On the second scenario we add an extra server that is idle. This container will however also be a receiver of metadata, hence the client container will have to send metadata to both other containers, resulting in a bandwidth usage of 16.5Kbps, effectively twice as much as when there was only one receiver. The third scenario reinforces this linear growth, as we add a third idle server, metadata bandwidth rises to 24.7Kbps. In the fourth scenario we add a second client producing a second flow, while keeping 3 servers (one of them is idle). We have therefore 2 flows, 5 containers and a total of 4 receivers for each flow. This results in a total usage of 65.9Kbps. For both the fifth and sixth scenarios we increase the number of clients to 3 and servers to 4. The difference between both is that in the sixth scenario, the flows produced by the clients use double the bandwidth than in the fifth scenario. We can observe that metadata bandwidth is the same for both scenarios, and is equal to 148Kbps, which means that metadata bandwidth usage is not affected by the amount of bandwidth being used by the experiment. With these results in mind, we can deduce that the formula for calculating metadata bandwidth usage is:

$$Bandwidth = \alpha * F * R \tag{6.1}$$

Where α is the amount of bandwidth used by disseminating a single flow to a single other container, in this case 8.26Kbps. And where F is the total number of active flows in the topology and R is the number of containers receiving metadata. Since every container is a receiver of metadata except for the container that produces it, we can conclude that $R = number\ of\ containers - 1$. A formula for accurately calculating the value of α for any given experiment is however complicated to derive since it can depend on the following variables: total number of links in the topology; number of flows a single container is producing; length of the paths that each flow must traverse. We can however state that the amount of bandwidth used by metadata grows linearly with the total number of containers in the topology and the number of flows that are active at a given moment.

On large scale experiments with hundreds of containers producing flows, the total amount of bandwidth used by metadata dissemination can rise to a Gbps order of magnitude, however it should be noted that a portion of this traffic will never cross the physical network, as it is destined to containers deployed on the same physical machine. Nevertheless it is the ability of the underlying cluster to handle these large amounts of traffic that will dictate the scalability limits of NEED.

6.4 Bandwidth emulation accuracy at scale

So far the method used to measure throughput was through analysis of network traces. However when scaling to larger experiments, this method quickly becomes unfeasible. The captured traces occupy large amounts of storage space (even if only packet headers are captured), and their analysis is also very time consuming. Another problem is that even though one instance of **tcpdump** might use few resources, when scaling to large experiments with tens of instances on a single Docker Swarm node, the CPU usage of the multiple **tcpdump** instances is no longer negligible, and will in fact interfere with the experiment results. Measuring the throughput with the output of **iPerf** would also not be acceptable since it measures application throughput and therefore shows spikes that are not visible in a network trace. This can be due to a socket blocking (when measuring at the client) or because of packet retransmissions (when measuring at the server). Measuring throughput with software like **dstat** is also not accurate, since they lack the ability to differentiate traffic, and would therefore include metadata traffic in the measurements. We needed a way to measure throughput that differentiated application traffic from metadata traffic, and that was as the least intrusive. Since NEED was already collecting throughput information from **tc**, we decided to log it and use it for the larger scale experiments.

In the following experiment we deploy a dumbbell topology similar to the previous one, but with 30 clients and 30 servers. In this topology the dumbbell is symmetrical, all the links between service instances and the switches are the same. The link connecting the switches is once again the bottleneck, and has a capacity of 600Mbps. This should give each client a share of 20Mbps. We start by deploying a single experiment with this topology, and then deploy 2 and 4 simultaneous deployments of the same experiment. The goal is to verify that NEED can execute multiple experiments simultaneously on the same cluster accurately, and also that the results of the same experiment are reproducible within a small error margin. Figure 6.9

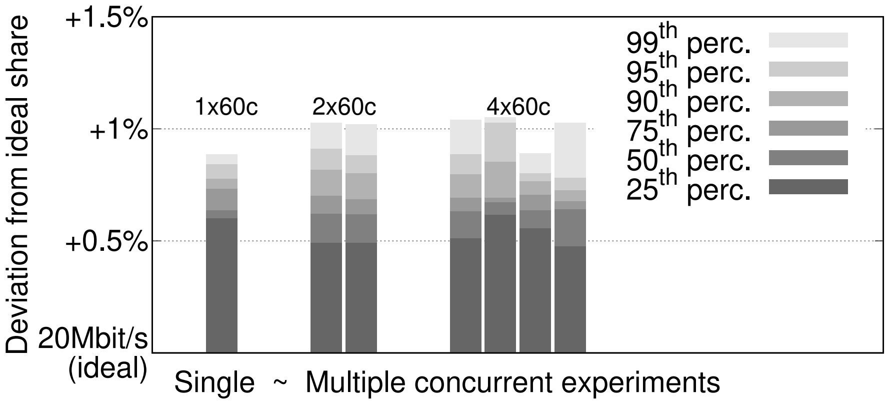


Figure 6.9: Deviation from ideal bandwidth share on the execution of up to 4 concurrent experiments in NEED

shows a bar plot where we represent percentiles 25th to 99th of the throughput values assumed during the execution. Once again the experiments ran for a total of 10 minutes, the first 60 seconds were discarded and the following 500 seconds are reported. With a single deployment all clients achieve the steady state throughput of 20Mbps. There is an error below 1%, this error is consistent with the precision provided by the **htb qdisc**. When deploying 2 and 4 experiments concurrently, we observe that the maximum error is still around 1%.

In Section 5.3, we presented the pseudocode for the emulation loop. There we explained that the emulation core broadcasts the local bandwidth usage twice, with an interval of 50ms between broadcasts, before taking the available information into consideration to calculate new bandwidth limits. Both the interval of time between broadcasts (default of 50ms), and the number of broadcast iterations (default of 2) are configurable by the user, and can affect the precision of the emulation. We have named these variables as pool period, and iteration count respectively, as explained in Section 5.3. These default values were chosen as they yield a good compromise between the response time of the system (demonstrated in Figure 6.7), and scalability to moderate sized topologies. In the following experiments we show how changes to these variables can affect system accuracy. The topology used for these experiments consists of a dumbell topology with initially 100 containers (50 **iPerf** clients and 50 servers) and then with 200 containers (100 **iPerf** clients and 100 servers). In both scenarios we set the available bandwidth in the middle link such that each container should get a share of 20Mbps for their network flow. Figure 6.10 shows one bar for each different setup of the experiment. The first

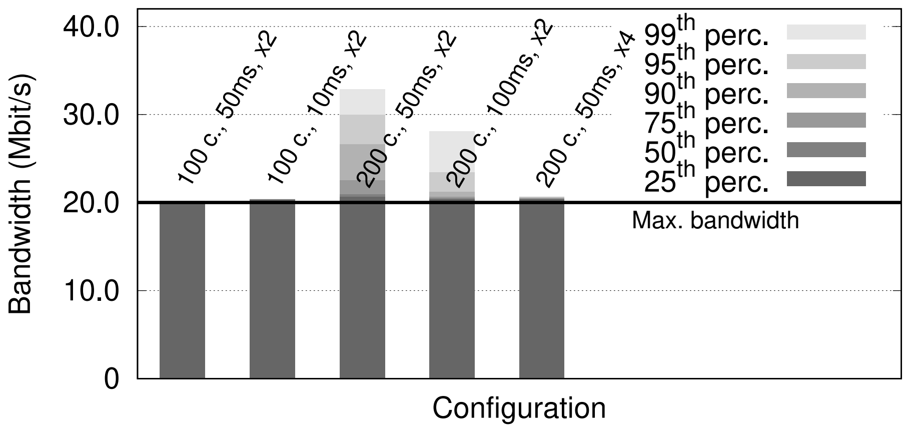


Figure 6.10: Accuracy in large scale experiments and the effects of tuning system parameters on scalability.

bar shows the first scenario with 100 containers. In this scenario we use the default pool period of 50ms and the default iteration count of 2. Results show a 99th percentile that is below 1% of error. The second bar shows the same topology with 100 containers, but this time we decrease the pool period from 50ms to just 10ms. Decreasing the pool period has 2 direct effects on the emulation. Reaction time to new and terminating flows will improve, however this comes at the cost of increased metadata bandwidth usage. In this case metadata bandwidth usage will be 5 times larger. The results show a 99th percentile that has 2% of error. Decreasing the pool period is only safe with small topologies, but it might be desirable to improve reaction time in experiments where the applications produce intermittent network flows. The increased error observed in this experiment, when compared to the previous one that used the default pool period of 50ms, is due to the fact that it takes slightly longer than 10ms for an instance to broadcast all metadata packets. This means that, in the case of packet loss or delay in the network, other instances might not get at least one metadata packet from every other active instance, leading to calculation errors.

The third bar shows the results for deploying 200 containers with the default values for pool period and iteration count. As can be observed, the results are no longer accurate, with the

99th percentile reaching 32Mbps (expected is 20Mbps). The default values no longer allow for accurate results at these high scales. With 200 containers, the amount of metadata being broadcasted is enough to cause delays in the network. These delays are significant enough to cause instances not to receive metadata from other instances in time to take them into consideration when calculating new bandwidth limits. In the fourth bar we attempt to mitigate this issue by increasing the pool period from 50ms to 100ms. As can be observed, there is an improvement in the accuracy, with the 99th percentile dropping to 28Mbps (previous was 32Mbps). By increasing the pool period to 100ms we are reducing the stress on the network, effectively halving the amount of metadata flowing between instances, which helps reduce network delays and packet loss. Note that by increasing the pool period we are effectively doubling the amount of time the system takes to react to new and terminating flows. This is however not enough to achieve accurate results. Finally in the fifth and last bar, we attempt the same deployment but this time we increase the iteration count from the default of 2 to 4, while keeping the pool period at the default 50ms. The default iteration count of 2, allows us to tolerate at most one omission fault (see Section 5.3). An iteration count of 4 should allow us to tolerate up to 3 sequential omission faults. Once again this comes at the cost of doubling the reaction time to new and terminating flows. This configuration allows us to reach more accurate results for the 200 container scenario, with the 99th percentile exhibiting only 3% of error. These experiments were executed using the ipvlan Docker network driver instead of the overlay driver used so far. Using the overlay driver is still possible at this scale, however we observed slightly degraded results when doing so. We believe the cause for this is related to the high bandwidth being used, which appears to be on the limits of what the overlay driver can handle.

6.5 Latency emulation accuracy at scale

On Section 6.4 we evaluated how NEED scales to large topologies while maintaining accuracy regarding the bandwidth sharing. In the current Section we will explore the scalability of NEED with regard to the accuracy of latency emulation. In Section 4.3 we stated that bandwidth emulation requires constant computations throughout the duration of the experiment to maintain accurate behavior. Latency emulation however, requires only computations during initialization, since latency values remain the same throughout the experiment. It is therefore expected that we can scale to larger topologies in experiments where there is no competition for bandwidth shares. So far our experiments have focused on simple dumbbell topologies, in this section, experiments will use more complex scale-free topologies. To generate topologies we use the preferential attachment method described in [BA99]. This method yields scale-free networks, which are representative of the characteristics of Internet topologies. The experiments described in this section consist of end-nodes sending ICMP echo requests to other random end-nodes, and comparing the obtained round trip times, with the theoretical ones. The latency of individual links is randomly generated between 1ms and 20ms. We have generated two topologies, one with 1000 nodes (666 services, 334 bridges), and another with 2000 nodes (1344 services, 656 bridges). On Table 6.1 we show the results of our experiments as a mean squared error between

Node count	NEED (overlay)	NEED (ipvlan)	MiniNet	MaxiNet
1000	0.0595	0.0261	0.0079	28.0779
2000	0.0799	0.0384	NA	347.5303

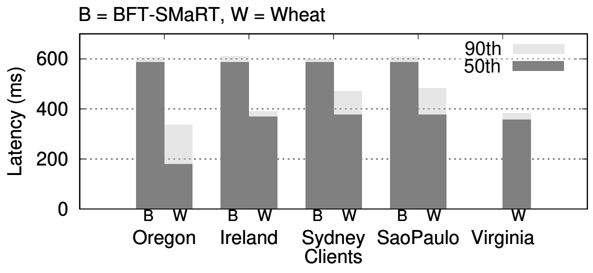
Table 6.1: Mean square error exhibited on latency tests with large scale-free topologies in NEED, Mininet and Maxinet.

theoretical expected values, that were computed during the generation of the topology, and experimentally obtained values. We have tested the same topologies on both NEED, Mininet and Maxinet, and within NEED we tested using both the overlay and ipvlan network drivers. It should be noted that the Mininet experiment was executed on a single machine, while the NEED and Maxinet experiments were executed on a cluster of 4 machines. We can observe that Mininet has a smaller error than NEED, this is likely caused by the overhead of container networking in Docker that introduces small but measurable delays. This overhead error can also be observed when comparing NEED with overlay and with ipvlan. Due to the different way both drivers are implemented, it was expected that overlay would exhibit higher overhead. The largest deviation from the theoretical round trip time value in NEED with overlay was 0.4ms, and with ipvlan was 0.2ms. For reference, the smallest theoretical RTT between two end-nodes in the 1000 and 2000 topologies is 10ms and 22ms, respectively, as such we consider the errors exhibited in NEED acceptable. It should be noted that precision below 0.1ms would be impossible for NEED, since it is executing on several physical hosts, and the communication between services on different hosts must always traverse physical networking links, that have been measured on our cluster to introduce a round trip time of 0.1ms. It should also be noted that **ping** was executed for a total of 10 minutes, and that the first 60 seconds were removed from the results. This was necessary because on Mininet the first 60 seconds of results were very inaccurate, leading to a very high mean square error. This is due to the network in Mininet requiring some packets to be exchanged before all the network elements are fully initialized and ready to operate at full speed. With NEED there is no such issue, results were accurate from the first measurement. Regarding the experiments with 2000 nodes, the results show a slight increase in error. Still the largest deviation from the expected value was the same as before, at 0.4ms with overlay, however errors occur more often, due to the higher resource utilization, which leads to the increased mean square error. Mininet freezes while launching this experiment, which prevents a comparison. Regarding Maxinet, the errors observed on both the 1000 and 2000 nodes experiments are significantly higher than on NEED or Mininet. We can observe a maximum deviation of 11ms and 40ms on the 1000 and 2000 nodes experiments respectively. It should be noted however that we found these errors to be directly related to both the OpenFlow controllers used and the scale of the experiments, and that it is perhaps possible to achieve better results with more advanced controller configurations. The controller configuration used for these experiments consisted of 4 POX [pox18] controllers executing the forwarding.l2_nx module. We experimented with several POX sample modules and also with Floodlight [flo18] and Opendaylight [ope18] controller configurations, and found this to be the configuration that produced the best results.

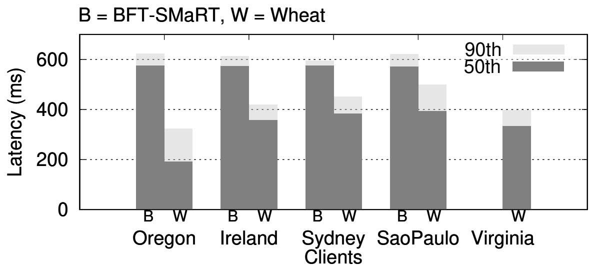
6.6 Geo-replicated systems

So far we have only executed experiments with synthetic traffic generation tools like **ping** and **iPerf**. In this section we evaluate NEED using typical distributed applications, and topologies that are representative of real-world deployments. We compare real geo-replicated deployments with emulated deployments in NEED.

Reproduction of results Our first experiment that replicates a real deployment within NEED makes use of BFT-SMaRt [BSA14], and its optimized version Wheat [SB16]. BFT-SMaRt is a library for development of distributed systems that implements Byzantine fault tolerant state machine replication. The authors of the previously mentioned systems evaluate and compare them in [SB16] by deploying both systems on geo-distributed Amazon EC2 instances spanning 5 regions. In [SB16], the authors provide together with the results of their experiments, a table with the average latency measured between the used regions, and also the measured jitter. With this data we were able to model a topology in NEED that should behave close to the real, global scale topology that connects EC2 regions. The modeled topology consists of a bridge at each region, that connects together the hosts on that region, and a link between every two regions. The latency within a region was set to 0ms, and the latency and jitter between regions was set according to the values provided in the paper. Since BFT-SMaRt is constricted by the high latencies between geo-distributed replicas, the experiment uses very little bandwidth (below 250kbps). We do not know the actual bandwidth limits between Amazon EC2 instances or regions, but it is safe to assume they are much higher than the bandwidth effectively used by BFT-SMaRt under this scenario. We therefore set the bandwidth limit on the links to safe values of 100Mbps between regions and 1Gbps inside a region. The experiment consists of placing one server and one client at each region. The servers run a simple replicated counter application, and the clients place requests to increment the counter. The 90th and 50th percentiles of the latency of requests is measured at each client. Figure 6.11a shows the results



(a) BFT-SMart deployment on Amazon EC2. Data from [SB16].



(b) BFT-SMaRT deployed on a local cluster with NEED.

Figure 6.11: Reproduction of an experiment with a geo-replicated deployment of BFT-SMart and Wheat - two Byzantine fault tolerant state machine replication libraries. The experiment measures the latencies of clients located in different Amazon EC2 regions. On the top, we have the results from the original paper, on the bottom the same experiment done with NEED.

from the original experiment on EC2. Figure 6.11b shows the results of executing the same experiments on NEED. As can be observed in the plots, the results of executing the experiment in NEED are close to the results achieved by the authors on EC2, with a maximum error of 6.8%. This deviation is thought to be caused by jitter on Amazon EC2 instances being highly variable, especially on the t1.micro instances used by the authors in the original experiments. In NEED jitter is assumed to follow a normal distribution. Since BFT-SMaRt and Wheat use very little CPU resources under this high latency scenario, these experiments were not executed on the cluster. Instead they were executed on a single commodity laptop, which shows that NEED is not limited to running experiments on powerful clusters.

NoSQL database evaluation The second experiment we performed to compare a NEED deployment with a real deployment makes use of Apache Cassandra [cas, LM10]. Cassandra is a NoSQL database system, widely adopted in production geo-replicated deployments. In our experiment we make use of YCSB [CST$^+$10] to benchmark a Cassandra geo-replicated deployment, first on Amazon EC2, and then on our cluster with NEED. The deployment consists of 4 Cassandra replicas in Frankfurt, and another 4 replicas in Sydney. Also 4 YCSB clients are deployed in Frankfurt. We setup Cassandra so that there is a replication factor of 2 at each region, meaning that each region will hold 2 copies of the same data. We also setup the regions to actively replicate one another, meaning that there should always be 4 copies of the same data overall. We configure YCSB to use a workload consisting of reads and updates in equal proportion. Furthermore we configure the consistency of YCSB operations to require a quorum on writes, and only one response on reads. Also we inform the YCSB clients that the closest Cassandra replicas are the ones located in Frankfurt. Due to the default load-balancer used in the YCSB client, this causes most requests to be directed at the Frankfurt replicas, leaving the Sydney replicas to only be directly contacted when the Frankfurt ones are under high load. However, due to the requirement of a quorum on updates, and the replication factor used, a response from at least 1 Sydney replica must always be present on every update to satisfy the quorum. In order to model the network topology in NEED, we collected the average latency and overall jitter between all the Amazon EC2 instances used prior to executing the experiment on Amazon. We found that the latency between two instances in the same region was negligible (below 0.1ms), and that the average round trip times between instances in Frankfurt and Sydney was 290ms with a standard deviation of 0.343. With that information we modeled a topology consisting of a bridge representing the Frankfurt region and another bridge representing the Sydney region. We created links connecting the two that forced a latency of 145ms with a standard deviation (jitter) of 0.243 in each direction. We then attached 4 Cassandra services to each of the previous bridges and 4 YCSB services to the Frankfurt bridge with no delay. Once again, we do not know the bandwidth limits between Amazon EC2 instances or regions, so we set them to 1Gbps in our modeled topology. We did however observe, that during the execution of the experiment, the bandwidth used between any two instances was always far bellow this value. Figure 6.12 shows the throughput-latency curve obtained from the benchmark on both the real deployment on Amazon, and on NEED. The curves for both reads and updates are a close match, showing only more significant differences after the turning point where response latencies climb fast, as Cassandra replicas are under high stress. It should be noted that it was unexpected to find that the latencies on the update curve decrease slightly as throughput increases before the turning point. However this behaviour occurs both on the real deployment and on NEED, and is likely caused by our Cassandra configuration. The fact that such issues are visible in emulation with NEED, aids our argument that they can be debugged and eliminated in emulation, which should be simpler and more cost-efficient than debugging them on real

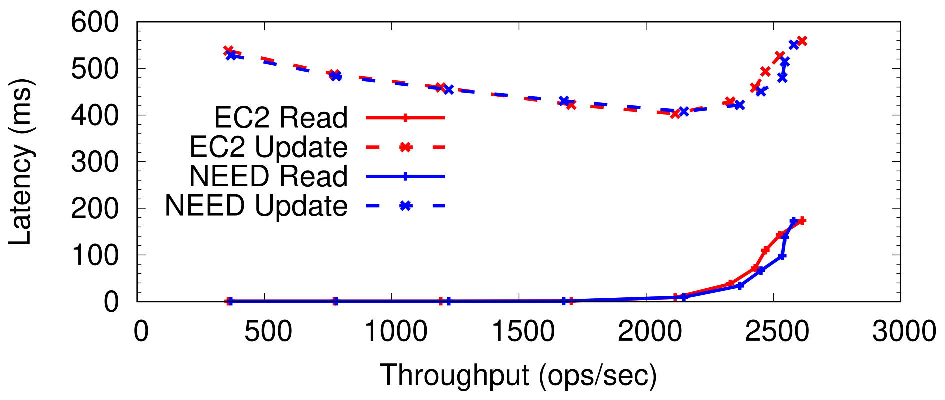


Figure 6.12: Comparison between benchmarking a Cassandra deployment on Amazon EC2 and on NEED. Each point corresponds to an increase of 100 threads distributed across 4 clients.

deployments.

Finally we present the results of executing the same Cassandra experiment but on a different hypothetical topology, where the latency between regions was reduced to half. Results are presented in Figure 6.13, alongside with the previous results to facilitate the comparison. As a

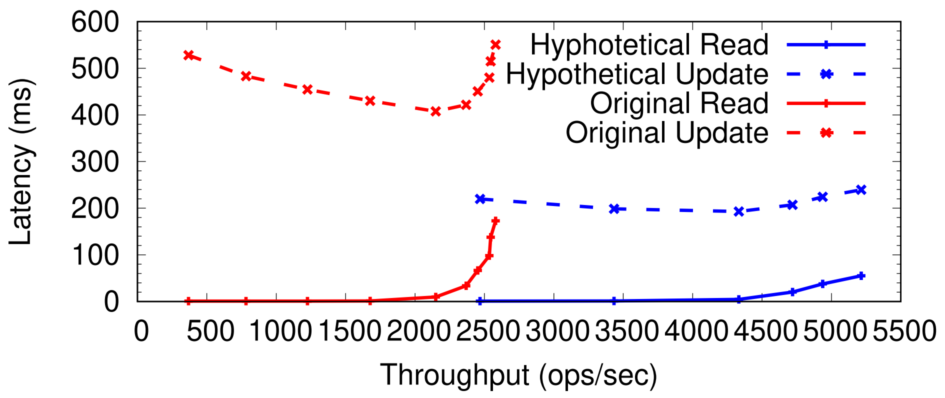


Figure 6.13: Benchmarking a Cassandra deployment on a hypothetical topology with NEED. Original results are included for comparison.

result of halving the latency between regions, the same Cassandra cluster was able to achieve a much higher throughput, and write latencies dropped to about half. Halving the latencies between Sydney and Frankfurt could correspond to testing what would happen if the Sydney replicas where moved somewhere else instead. This shows that NEED can be used for experimenting with hypothetical what-if scenarios, in order to assess application behavior under a wide range of deployment scenarios.

Chapter 7

Conclusions

The evaluation of distributed systems is a crucial step to assess their correctness and good performance. Network emulation systems help developers and researchers reduce the costs and effort of performing this task. In this work we have presented the architecture and an implementation of a network emulation system targeted at distributed systems developers. The presented system combines the usage of containers and a decentralized design that allow to introduce new ideas on the design of network emulation systems. By collapsing the topologies into end-to-end links that retain the high level properties of the original topology, we free the users from dealing with issues and unexpected behaviors, related to the components that form the network, and allow them to focus on the applications and on the macro properties of the network that affect them. By basing our system on containers we come closer to achieving reproducibility of experiments, and allow for a broad range of applications to be tested.

The presented system can scale to hundreds of nodes while maintaining accuracy on all emulated properties, however in scenarios where bandwidth contention does not occur, the decentralized point-to-point architecture of the system presents no limitations, and the system can scale as far as the underlying Docker Swarm technology allows. We have shown that despite the simplifications made about network state, the presented system is able to accurately reproduce real-world deployments of off-the-shelf popular systems such as Apache Cassandra. We have also shown that the presented system is capable of reproducing results presented in literature. We showed this by reproducing results from a geo-replicated state machine replication system presented in the literature [SB16]. The presented system can also be used to conduct what-if scenarios, allowing engineers to evaluate application performance and correctness under hypothetical, but fully controlled, network conditions.

7.1 Future Work

The presented system has shown potential as a useful tool for distributed systems developers, however it is still only a prototype. Next we present some limitations of the system and discuss possible solutions.

Reproducibility NEED is capable of achieving replicability of experiments, however to achieve full reproducibility, awareness of the resource limits of the underlying cluster (such as maximum bandwidth between nodes) must be taken into consideration. With this information NEED could perform scheduling of its containers to avoid attempts to use more resources than those that are available.

Dynamic topologies The current system has no support for dynamic topologies, which prevents common evaluation scenarios like hosts leaving and new hosts joining the network. This functionality could be added to NEED, simply by making all emulation core instances coordinate to switch to a new topology at the same time, however coordinating thousands of instances to do so is challenging.

Scalability Finally, although the presented system has been shown to scale to as much as 2000 nodes, scaling the number of nodes while maintaining accurate bandwidth contention emulation, requires sacrificing response times. The main bottleneck of the system lies in the large amount of metadata traffic that is generated with large topologies. We present two alternative solutions to mitigate this problem. The first alternative would be to centralize the broadcasting of metadata within each node to a separate container. This container would be responsible for disseminating metadata to all containers within the same host, and to the other broadcast containers on the other hosts. This approach would significantly reduce the amount of metadata that must cross the physical network, by removing duplicated data that is originally targeted at multiple destinations. The second alternative would be to change the current behavior of always broadcasting the current bandwidth usage at fixed intervals, and only broadcast when a significant change occurs to the active flows. To do so the emulation core would have to be modified to remember previous data, and metadata would have to be exchanged on a reliable channel.

Bibliography

[Ale01] Alexey N. Kuznetsov, J Hadi Salim, Bert Hubert. *PRIO(8) Linux User's Manual*, December 2001.

[amaa] Amazon Elastic Compute Cloud. `https://aws.amazon.com/ec2/`. Accessed: 2018-9-25.

[amab] Amazon Elastic Container Service. `https://aws.amazon.com/ecs/`. Accessed: 2018-9-25.

[AV06] M. Avvenuti and A. Vecchio. Application-level network emulation: the emusocket toolkit. *Journal of network and computer applications*, 29(4):343–360, 2006.

[BA99] Albert László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[Ber02] Bert Hubert, Martin Devera. *HTB(8) Linux User's Manual*, January 2002.

[BJL$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *Osdi02*, page 255–270, 2002.

[BRNR15] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. A survey of general-purpose experiment management tools for distributed systems HAL Id : hal-01087519. *Future Generation Computer Systems, Elsevier*, 45,:1 – 12, 2015.

[BSA14] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with BFT-SMART. *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, pages 355–362, 2014.

[cap17] *CAPABILITIES(7) Linux Programmer's Manual*, September 2017.

[cas] Apache Cassandra. `https://cassandra.apache.org/`. Accessed: 2018-9-25.

[CR09] Marta Carbone and Luigi Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12, 2009.

[CS03] M. Carson and D. Santay. NIST Net-a linux-based network emulation tool. *ACM SIGCOMM Com. Comm. Rev.*, 33(3):111–126, 2003.

[CST$^+$10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[Doc] Docker home page. `https://www.docker.com/`. Accessed: 2017-10-16.

[dum02] *DUMMYNET(4) FreeBSD Kernel Interfaces Manual*, October 2002.

[Eri05] Marius a Eriksen. Trickle: A Userland Bandwidth Shaper for Unix-like Systems. *Usenix*, pages 61–70, 2005.

[flo18] Floodlight. `http://www.projectfloodlight.org/floodlight/`, 2018. Accessed: 2018-9-25.

[Hau16] Michael Hausenblas. *Docker-Networking-and-Service-Delivery.* O'Reilly Media, first edition, 2016.

[Hem05] Stephen Hemminger. Network emulation with NetEm. In *Proceedings of the Linux Conference*, 2005.

[Hem12] Stephen Hemminger. *BRIDGE(8) Linux User's Manual*, August 2012.

[HHJ$^+$12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[HRS$^+$08a] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference*, 2008.

[HRS+08b] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 113–128, 2008.

[Hub01] Bert Hubert. *TC(8) Linux User's Manual*, December 2001.

[Hub15] Bert Hubert. *Universal 32bit classifier in tc(8) Linux User's Manual*, September 2015.

[IEE06] IEEE Computer Society. *IEEE Standard for Local and metropolitan area networks - Virtual Bridged Local Area Networks*, volume 2005. 2006.

[IP94] David B Ingham and Graham D Parrington. Delayline: a wide-area network emulation tool. *Computing Systems*, 7(3):313–332, 1994.

[ipe] iPerf. `https://github.com/esnet/iperf`. Accessed: 2018-9-25.

[Kel] Frank Kelly. Charging and rate control for elastic traffic. *European Trans. on Telecommunications 8*, pages 33–37.

[Kub] Kubernetes home page. `https://kubernetes.io/`. Accessed: 2018-9-25.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop. *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, pages 1–6, 2010.

[Lit12] Michail Litvak. *IP-LINK(8) Linux User's Manual*, December 2012.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LRF09] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 185–198, Berkeley, CA, USA, 2009. USENIX Association.

[lxc] Linux LXC. `https://linuxcontainers.org/`. Accessed: 2018-9-25.

[mac] Support swarm-mode services with node-local networks. `https://github.com/moby/moby/pull/32981`. Accessed: 2018-9-25.

[MDD$^+$14] M Mahalingam, D Dutt, K Duda, P Agarwal, L Kreeger, T Sridhar, M Bursell, and C Wright. RFC 7348. 2014.

[Mer14] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment, 2014.

[Mes] Apache Mesos home page. `https://mesos.apache.org/`. Accessed: 2018-9-25.

[MR02] Laurent Massoulié and James Roberts. Bandwidth sharing: Objectives and algorithms. *IEEE/ACM Transactions on Networking*, 10(3):320–328, 2002.

[nam17] *NAMESPACES(7) Linux Programmer's Manual*, September 2017.

[net17] *NETLINK(7) Linux Programmer's Manual*, September 2017.

[Nom] Hashicorp Nomad home page. `https://www.nomadproject.io/`. Accessed: 2018-9-25.

[NR08] L. Nussbaum and O. Richard. Lightweight emulation to study peer-to-peer systems. *Concurrency and Computation: Practice and Experience*, 20(6):735–749, 2008.

[NS3] Network simulator 3 website. `http://www.nsnam.org`. Accessed: 2017-10-21.

[ope18] Opendaylight. `https://www.opendaylight.org/`, 2018. Accessed: 2018-9-25.

[PKVR16] Manuel Peuster, Holger Karl, and Steven Van Rossem. Medicine: Rapid prototyping of production-ready network services in multi-pop environments. In *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*, pages 148–153. IEEE, 2016.

[pox18] POX. `https://github.com/noxrepo/pox`, 2018. Accessed: 2018-9-25.

[PPM08] Z. Puljiz, R. Penco, and M. Mikuc. Performance analysis of a decentralized network simulator based on IMUNES. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, SPECTS, 2008.

[PR08] M. Pizzonia and M. Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *4th Intl. ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities*, TridentCom, 2008.

[RAAN$^+$08] Roberto Roverso, Mohammed Al-Aggan, Amgad Naiem, Andreas Dahlstrom, Sameh El-Ansary, Mohammed El-Beltagy, and Seif Haridi. MyP2PWorld: Highly reproducible application-level emulation of P2P systems. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW, 2008.

[RDS+07] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau. The Flexlab Approach to Realistic Evaluation of Networked Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2007.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[SB16] Joao Sousa and Alysson Bessani. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2016-Janua:146–155, 2016.

[SRF13] Valerio Schiavoni, Etienne Riviere, and Pascal Felber. SplayNet: Distributed User-Space Topology Emulation. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 62–81. Springer, 2013.

[Ste11] Stephen Hemminger, Fabio Ludovici, Hagen Paul Pfeifer. *NETEM(8) Linux User's Manual*, November 2011.

[SW15] Vineet Sinha and Mea Wang. evalbox: A cross-platform evaluation framework for network systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 15–18. IEEE, 2015.

[swaa] Missing from swarmmode –cap-add. `https://github.com/moby/moby/issues/25885`. Accessed: 2018-9-25.

[swab] Run docker engine in swarm mode. `https://docs.docker.com/engine/swarm/swarm-mode/`. Accessed: 2018-9-25.

[Tan09] C. Tang. DSF: a common platform for distributed systems research and development. In *10th ACM/IFIP/USENIX International Conference on Middleware*, 2009.

[TCB15] Marco Antonio To, Marcos Cano, and Preng Biba. DOCKEMU - A Network Emulation Tool. *Proceedings - IEEE 29th International Conference on Advanced*

Information Networking and Applications Workshops, WAINA 2015, pages 593–598, 2015.

[VGVY09] Kashi Venkatesh Vishwanath, Diwaker Gupta, Amin Vahdat, and Ken Yocum. ModelNet: Towards a datacenter emulation environment. In *IEEE P2P'09 - 9th International Conference on Peer-to-Peer Computing*, number May, pages 81–82, 2009.

[VYW+02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002.

[WCL07] SY Wang, CL Chou, and CC Lin. The design and implementation of the NCTUns network simulation engine. *Simulation Modelling Practice and Theory*, 15(1):57–81, 2007.

[WDS14] Philip Wette, Martin Dräxler, and Arne Schwabe. MaxiNet: Distributed emulation of software-defined networks. *2014 IFIP Networking Conference, IFIP Networking 2014*, 2014.

[WSVL+11] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. SliceTime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 253–266, 2011.

Appendix A

NEED experiment example

Listing A.1: Instructions for executing an experiment with NEED

```
This experiment executes 3 iperf3 clients (and respective servers).
To execute this example a machine running Docker is required.
A single laptop is enough.

The docker images provided in ./Docker_Images folder need to be built.
To do so execute the following commands:
  $docker build -t need/client:1.0 ./Docker_Images/client
  $docker build -t need/server:1.0 ./Docker_Images/server
  $docker build -t need/dashboard:1.0 ./Docker_Images/dashboard
  $docker build -t privilegedbootstrapper:1.3 ./Docker_Images/bootstrapper_image

If the current machine is not part of any Swarm cluster create one with:
  $docker swarm init

A docker swarm network named test_overlay must also exist.
To create it run:
  $docker network create --driver overlay test_overlay

This folder contains a topology description file "./topology5.xml".
To turn this description into a Docker Swarm deployment execute:
  $../NEED/deploymentGenerator.py ./topology5.xml > topology5.yaml

Now the experiment can be deployed with:
  $docker stack deploy -c topology5.yaml topology5

Next open a browser and navigate to:
  http://127.0.0.1:8088
This will open the dashboard that allows for monitoring the experiment.
Click "START" at the top right of the screen to begin the experiment.
The active flows can be monitored in the "Active flows" tab.
A graphical representation of the topology can be seen in the "Graph" tab.

To stop the experiment click "STOP" at the top right corner.
The deployment can then be removed with:
  $docker stack rm topology5
```