A Pragmatic Testbed for Distributed Systems

by

Parth Patel

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Applied Science

in

Electrical and Computer Engineering

University of Ontario Institute of Technology

November 2017

Abstract

Distributed systems have become ubiquitous and they continue their growth through a range of services. With advances in resource virtualization technology such as Virtual Machines (VM) and software containers, developers no longer require high-end servers to test and develop distributed software. Even in commercial production, virtualization has streamlined the process of rapid deployment and service management. This thesis introduces a distributed systems testbed that utilizes virtualization to enable distributed systems development on commodity computers. The testbed can be used to develop new services, implement theoretical distributed systems concepts for understanding, and experiment with virtual network topologies. We show its versatility through three case studies that utilize the testbed for designing a service based solution, implementing a theoretical algorithm, and developing our own methodology to find high-risk edges. The results of using the testbed for these use cases have proven the effectiveness and versatility of this testbed across a range of scenarios.

Acknowledgements

I would like to express my sincere gratitude to my co-supervisors, Dr. Mark Green and Dr. Ying Zhu for their mentorship throughout my time as a graduate student. Without their persistent guidance and patience, this work would not be possible. Under their leadership, I have gained invaluable research and development skills that will serve me well as a Software Engineer.

Table of Contents

List of Figures

List of Acronyms

API – Application Programming Interface

BSP – Bulk Synchronous Parallel

IOT – Internet of Things

MST – Minimum Spanning Tree

OS – Operating System

P2P – peer to peer

RPC – Remote Procedure Call

SOA – Service Oriented Architecture

VM – Virtual Machine

1 Introduction

Distributed computing has been an active topic of research ever since the first form of process to process communication. With the current state of advancements made in the Internet network infrastructure, software has migrated from the user's local computer to the cloud where it provides service through the Internet. Not just software, but also physical devices equipped with communication hardware can now provide integrated services from the cloud, also known as Internet of Things (IOT). This is due to the reliability of the network infrastructure, and a gradual increase in the average bandwidth of an internet connection. Research in distributed computing requires a network environment where new and existing concepts can be experimented with. In universities, test networks are relatively small in terms of the number of physical nodes in the network. Many times, there are Virtual Machines (VMs) within the physical nodes that emulate multiple virtual nodes. There is a limit to the number of VMs a physical host can host and therefore the number of nodes in a testbed is usually limited. With new advances in hardware and software virtualization, we present a distributed systems testbed that can handle hundreds of lightweight virtual nodes within a physical node.

For many distributed algorithms, experiments take place in the form of concurrently running threads that communicate via inter-process communication. Although this may model many scenarios to a satisfactory level, it abstracts many of the aspects that factor into a real functioning network. Communication, for example, requires a message to be encoded in a pre-defined format and transmitted using an agreed upon protocol such as TCP or UDP. In contrast, inter-thread communication is based on inserting and removing

messages from shared memory. In distributed algorithms where nodes execute the algorithm asynchronously, unordered events in the network are much more realistic when we see messages flowing through network communication links. Communication through local shared memory becomes deterministic and the algorithm executes without being tested for realistic failures such as byzantine failures that disrupt the execution. Implementing an algorithm to test also becomes a two-fold effort where first it's implemented locally using threads, and then re-programmed to deploy it in a real network. If the system testbed preserves the same code implementation as one would deploy in a real network, then verification becomes efficient. Taking this into consideration we present a testbed consisting of a virtualized node that is as light-weight as a process, however it provides a runtime platform that emulates a real node in terms of its networking Application Programming Interface (API).

1.1 Motivation

The motivation for this thesis is to introduce a pragmatic distributed systems testbed that can be used to model systems that reflect real-world networks with a reasonably large number of nodes within a physical host. In an academic setting, this is highly desirable as it may not always be feasible to acquire the resources to build an actual network. Our focus is primarily on distributed programs at the application level, and therefore low-level networking is out of the scope of this paper.

We intend to utilize the advancements in resource virtualization technology to simulate many aspects of a distributed system. The test bed will allow users to simulate various network topologies by using overlays on the virtual networks. Using software

containers such as Docker we can simulate individual nodes in a network with very minimal hardware requirements. We have also carefully considered the language of choice for implementing the logic, so that rapid prototyping, ease of learning, and performance are not compromised.

1.2 Contributions

This thesis contributes a distributed systems testbed that can simulate a large number of nodes and network topologies that can be used to test application level distributed programs. This is highly desirable in peer-to-peer networks where topology changes occur frequently and so does peer churn. The objective is to allow users with little hardware resources to simulate networks that are not meant to be computationally intensive. Some use-cases include rapid prototyping for a service based system, theoretical work verification, or simply to help understand the mechanisms of a distributed system. Through three case studies, we show the versatility of our testbed in different setups.

In the first case study we utilize microservices, a form of Service Oriented Architecture, for rapid prototyping. Microservices are fine-grained modules of logic that provide a specific functionality to the overall system. This concept contrasts with the traditional monolithic application architectures that tightly couple all functionalities of the system into a single tiered application. Using software containers as separate services we show the efficiency in developing a distributed application with a modular approach to integrating the system. This allows researchers and developers to focus on the logic rather than the integration and deployment. Light-weight software containers allow application functionalities to be rapidly prototyped and packaged for deployment on the local machine.

This case study is based on a commercial solution, named ResourceManager, proposed by a local software firm to UOIT as a joint development project. ResourceManager is a distributed application that would provide a federated collaboration platform where organizations can share resources from their own local repositories. Business requirements of the platform are organized into features that can be implemented as independent services. We discuss how using our testbed can streamline the development of this multi-organization project with a lean environment for rapid prototyping.

Our second case study implements a distributed Minimum Spanning Tree (MST) algorithm to explore a theoretical concept in practice. The algorithm was developed by Gallager, Humblet, and Spira [31]. Compared to a standard MST algorithm, this algorithm solves the MST in a completely distributed manner where the global topology of the network is not known. Nodes are only aware of their adjacent neighbours, and they compute the MST via message passing. In practice, this algorithm can be used in peer-to-peer networks where the global topology is unknown by peers, however economical broadcast routes need to be computed. Using our proposed testbed, we implement the GHS algorithm where each node is encapsulated within a software container, and the network topology is simulated through a virtual network using overlays. We show that our testbed can run a network of hundreds of nodes which gives us the opportunity to observe the execution of such an algorithm in practice.

In the final case study, we design and develop a completely distributed method to detect high-risk communication links that can be used to increase the reliability and resilience of a distributed system. Using the topologies of scale-free networks where some nodes have a high degree of links, and many nodes have a small degree, we define important nodes as ones with a high number of links. High-risk communication links are defined as a

connection between two important nodes in the network. Knowing the global topology and determining such links is a trivial task, however without any information about the global topology, identifying such links becomes a challenge. We assume a peer-to-peer network where each node is only aware of its adjacent neighbours, and can only infer information about the topology via message passing. To simulate traffic, messages are passed around using the random walks on networks model described in chapter 5. Redefining the important nodes as the ones with significantly higher traffic than the rest of the network, the high-risk links connect two high-traffic nodes. The next challenge is to determine if a node is high-traffic without having traffic information of the rest of the nodes in the network. Utilizing gossip-based network aggregation to compute the system's traffic average, a baseline was formed to determine if the current node is a high-traffic node. The network aggregation algorithm used was Push-Sum [41]. With the proposed testbed, we were able to simulate various scale-free network topologies with hundreds of nodes within a single physical host to verify our methodology. Chapter 5 describes the methodology in detail and the results are discussed.

The testbed itself consists of 3 major components:

- Docker – A software containerization technology that uses operating system virtualization to abstract and share the hardware resources on the host. This is the major component as it also simulates an overlay network within a host that can be configured extensively.
- Go – A systems language used to implement the node logic.

- Etcd – A decentralized log used as a shared-data mechanism in a distributed system.

1.3 Thesis Organization

This thesis is structured so that the three case studies are separated into chapters as they are not directly relevant to each other, but rather use the proposed testbed in different situations. Chapter 2 describes the testbed in extensive detail outlining the internal mechanisms of the components and justification for why they were chosen. Chapter 3 presents the first case study where rapid prototyping for a software project is streamlined using the proposed testbed. Chapter 4 describes the GHS algorithm and implementation results. Chapter 5 introduces a new methodology for detecting high risk edges in distributed peer-to-peer manner, developed using the proposed testbed. Chapter 6 discusses conclusions drawn from utilizing the testbed and potential future work on this project.

2 The Testbed

In distributed systems research, new and existing concepts need to be verified and validated. To achieve this, the concepts must be implemented and a model must be produced. A distributed system model is comprised of two nodes and a link between them at a minimum. Each node should have its own memory and processor, since a node is considered a complete separate unit. In a message-passing peer-to-peer model, the algorithm to be tested is implemented and all nodes execute the same algorithm. The nodes respond based on the messages they receive and execute the program accordingly. Ideally this algorithm would be deployed across nodes in a physical network to verify its correctness in a real-world setting. However, it's not always feasible to obtain such a physical network for experimentation. With resource virtualization, distributed system environments can be simulated within a physical node. There are a plethora of such environments and selecting an environment narrows down to the specific requirements of the algorithm and model in question. Some environments for example may focus on providing ease in simulating overlay networks whereas others are more suitable for lower level network functionalities. Since the focus of the thesis was not low-level details, but the architectural level concepts of distributed systems, a versatile virtual environment was composed that can simulate various high level models. We utilize Docker, a containerized virtual environment, and GoLang which is designed for server-side concurrent programming. The following sections elaborate on these two components.

2.1 Docker

Docker [1] is a platform that utilizes Operating System (OS) level virtualization to provide resource isolation. This type of virtualization is known as software containerization, where applications with their dependencies are deployed and executed in separate containers. By contrast, Virtual Machines (VMs) use hardware level virtualization of resources. The abstraction of discrete hardware components is achieved with a hypervisor software [2] which allocates and manages the execution of multiple guest operating systems that are hosted on one physical host. Figure 1 shows a high level overview of the differences between containers and VMs. The following sections explore the major components of Docker containers such as the operating system and networking functions. They will be compared to a VM's setup and show why Docker is more lightweight in those areas and why its more suitable for rapid prototyping of distributed systems research.

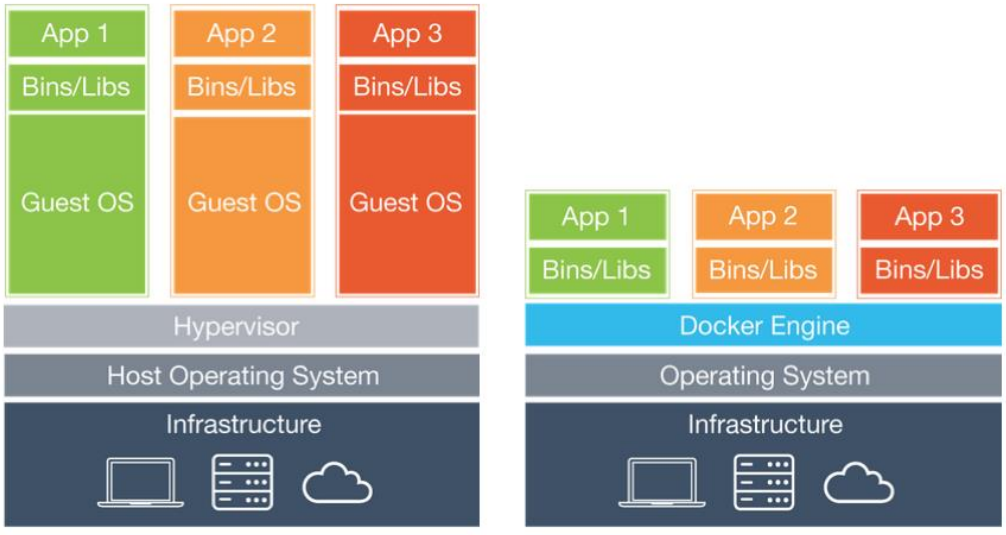

Figure 1: Architectural overview of VMs compared to Containers (Source: docker.com).

2.1.1 Process Space

The key component of OS level virtualization is the kernel [3] which is essentially the core program of an OS. A kernel manages the computer's hardware resources and abstracts the low-level details. It manages the execution of processes, memory, and the input/output peripherals by exposing an interface to the higher-level applications. Applications are free from low level execution details such as scheduling, allocating free memory, and obtaining device drivers to access the peripherals. Multiple applications can execute concurrently on an OS with these facilities provided by the kernel. Docker utilizes the kernel's ability to isolate multiple user spaces, which allow processes within the containers to run independently.

Namespaces is the major resource isolation feature of a kernel which provides an identity to the resource space. When an OS boots up, a single instance of each type of namespace is started for OS resources. In the Linux OS kernel, the *pid* (process ID) namespace type is of interest for understanding containers. A process ID is used to identify running processes in an environment, which can then be accessed via system calls. *pid* namespaces work as a nested hierarchy where each process can spawn nested child processes under its own namespace. Processes can only access and modify the child processes, not vice-versa. Therefore, processes spawned by process *A* will not be able to address processes under *B*. Processes may also have multiple *pid*s since a process will be under a parent's namespace and have its own namespace for its own child processes.

Docker containers are essentially isolated processes which can spawn child processes (includes user space applications) within the container's process. Each container is a child process of the Docker daemon, which is a background process running on the

host operating system to manage containers. Figure 2 shows the list of processes from the host's perspective. Here the Docker daemon (*dockerd*) is the root process, and each container is a child process. Note the containers' *pid* for the processes are within the namespace of the daemon process.

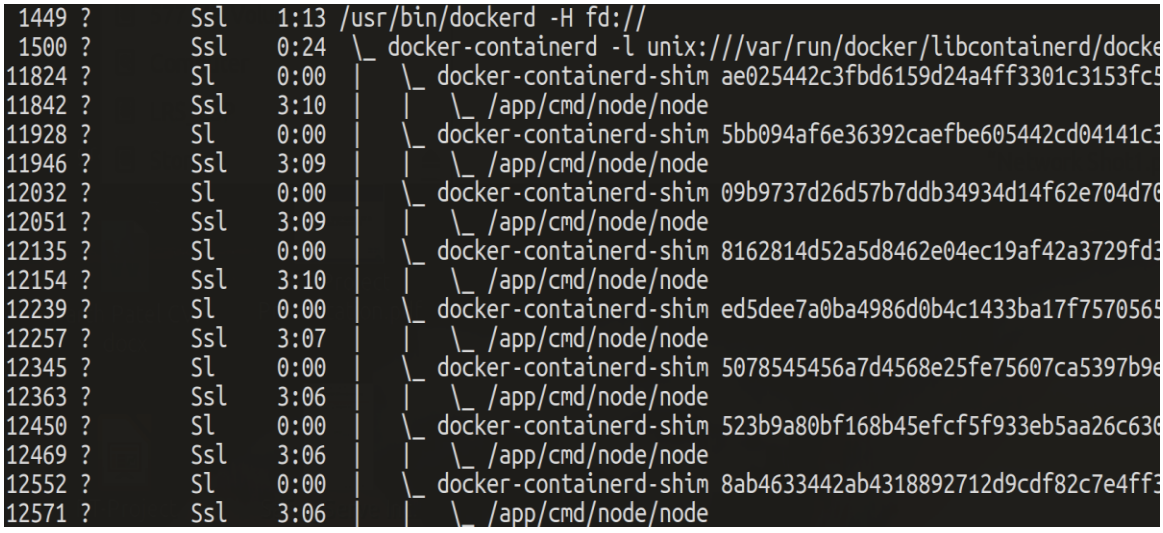

Figure 2: Process list from host using command 'ps -afx'.

Processes running within the containers, such as an example P2P application "node" is a child process of the container it is executing in. There are multiple containers executing the same application in isolation as it would in a typical P2P network. Figure 3 shows the list of processes from a container's perspective and here the P2P application "node" is a root process with a different *pid* namespace. To the "node" application, other containers are a logically separated entity as it would if they were on physically different machines. This logical separation is sufficient for most experiments testing distributed systems algorithms at the conceptual level.

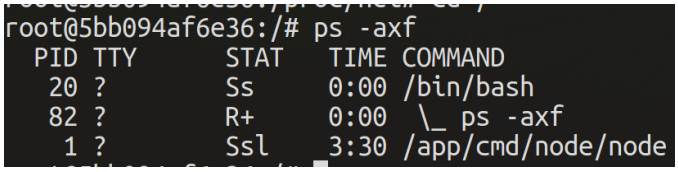

Figure 3: Process list from container using command ‘ps -afx’

Compared to a VM, this isolation has very little to almost no overhead on performance since the process partitions can share the host OS’s system calls. Moreover, hardware dependencies that have to be resolved by hypervisors for VMs to function are also eliminated.

Memory in containers is treated like any other process’ memory in the OS and is associated with the running container. It can be configured through the Docker daemon to be limited if the container is running on a resource limited host.

2.1.2 Images and Volumes

A Docker image can be described as an immutable template that can be executed as a container. A container is an instance of the image. Multiple layers make up an image, and are combined when the image is built using *docker build* command. When a container is instantiated with a built image using the *docker run* command, a thin writeable layer is added. This layer is known as the container layer and is required to hold runtime system data during container execution. See Figure 4. It is important to note that since images are idiomatically designed to be stateless, any changes made during container runtime are not reflected on the images themselves, but rather on the temporary container layer. Multiple containers can instantiate a built image, and the overhead on memory is only the container layer per container. In contrast, VMs require their own set of resources for memory and

storage for each instance. As an example, if a VM image is 1GB, 10 VMs would require 10GB whereas a container image of the same size would require under 2GB for the same 10 instances.

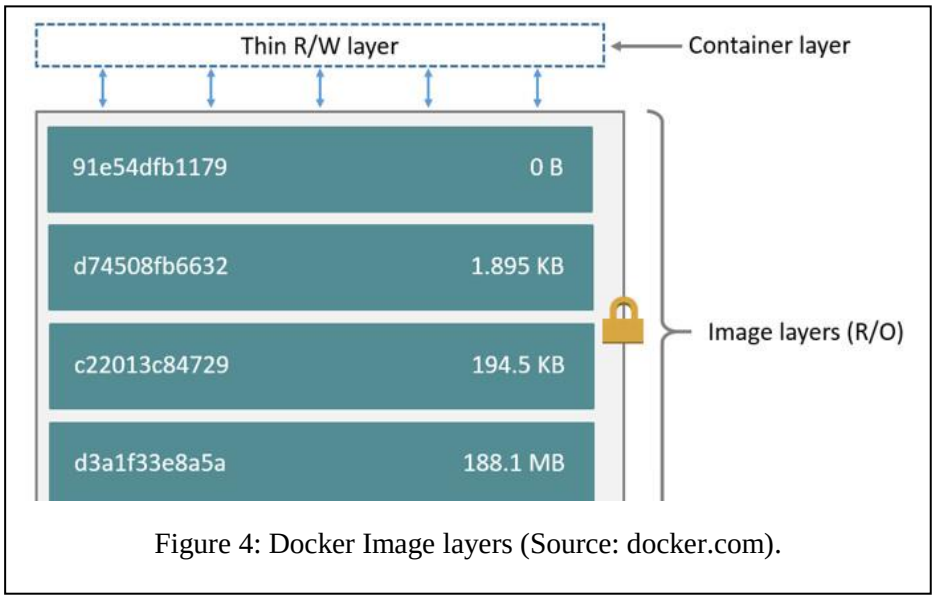

Figure 4: Docker Image layers (Source: docker.com).

For persistent storage, Docker utilizes data volumes which persist data even after the end of the container's lifecycle. The volume resides on the host's filesystem and is initialized when the container is first created. Decoupling persistent storage from the lifecycle of the container significantly reduces the physical memory resource requirements on the host. Depending on the size of the application itself, persisting runtime data from the application can occupy large amounts of unnecessary disk space on the host. A VM in a stored state contains this temporary runtime data from user applications and guest OS unless it is manually removed before saving its state. This isn't ideal when running distributed system experiments that only need the output data of the experiment for offline analysis. Therefore, storing only the required data for analysis on the data volumes is the

most efficient method of disk space utilization. Runtime data is removed at the end of the container's lifecycle. Multiple containers can also share a data volume on the host, which provides flexibility in modelling a distributed system experiment. For example, in a P2P network, nodes can write their output data on a shared volume that can be used for both online and offline analysis of the global network changes.

2.1.3 Network Space

Docker networking is isolated among containers using the *net* namespace. Similar to the processes, virtual ethernet (veth) interfaces used for networking are assigned to the *net* namespaces. Containers have emulated logical network interfaces originally found on the host machine, such as the *eth0* and *lo* loopback interfaces. Since the interfaces are in different namespaces, they can share the same names while listening on different virtual addresses. Each *eth0* interface binds to a virtual ethernet interface on the host which is part

```
docker0   Link encap:Ethernet  HWaddr 02:42:e6:95:b7:08
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:e6ff:fe95:b708/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1

enp9s0    Link encap:Ethernet  HWaddr 50:7b:9d:69:b8:fe
          UP BROADCAST MULTICAST  MTU:1500  Metric:1

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1

vethdf5a870 Link encap:Ethernet  HWaddr 9e:b7:bf:7e:b1:4f
          inet6 addr: fe80::9cb7:bfff:fe7e:b14f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1

wlp8s0    Link encap:Ethernet  HWaddr a4:34:d9:46:a9:e2
          inet addr:192.168.0.20  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fd00:f0f2:494a:9802:a2a:74dc:4a0c:1d58/64 Scope:Global
          inet6 addr: fe80::753d:d3a0:d3a1:f7e8/64 Scope:Link
          inet6 addr: 2607:fea8:33df:fc00:408:8f86:ea04:be1f/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

Figure 5: Running "ifconfig" command.

of the original *net* namespace. It will have a name with a unique suffix to differentiate multiple virtual interfaces such as “veth12abc3”. This binding allows communication from the container to the host, see Figure 5 for network interfaces in a Docker enabled host. Figure 5 is the result of running the linux command "ifconfig", which lists the network interfaces on the host. The "vethxxx" interface is bound to a running container's "eth0" interface.

With Docker installation, a default networking bridge named *docker0* is added to the host. Bridge devices combine multiple networks together, but they do not handle the routing. The *docker0* bridge resides on the host’s net namespace, and aggregates the virtual ethernet interfaces from the containers. All container traffic passes through this bridge, including communication from the host to the containers and vice-versa. Figure 6a shows the result of inspecting the docker bridge from the host, and Figure 6b depicts the high-level architecture.

```
parth ~ $ brctl show
bridge name   bridge id           STP enabled   interfaces
docker0       8000.0242e695b708   no            vethafa6174
                                                vethf1155d5
```

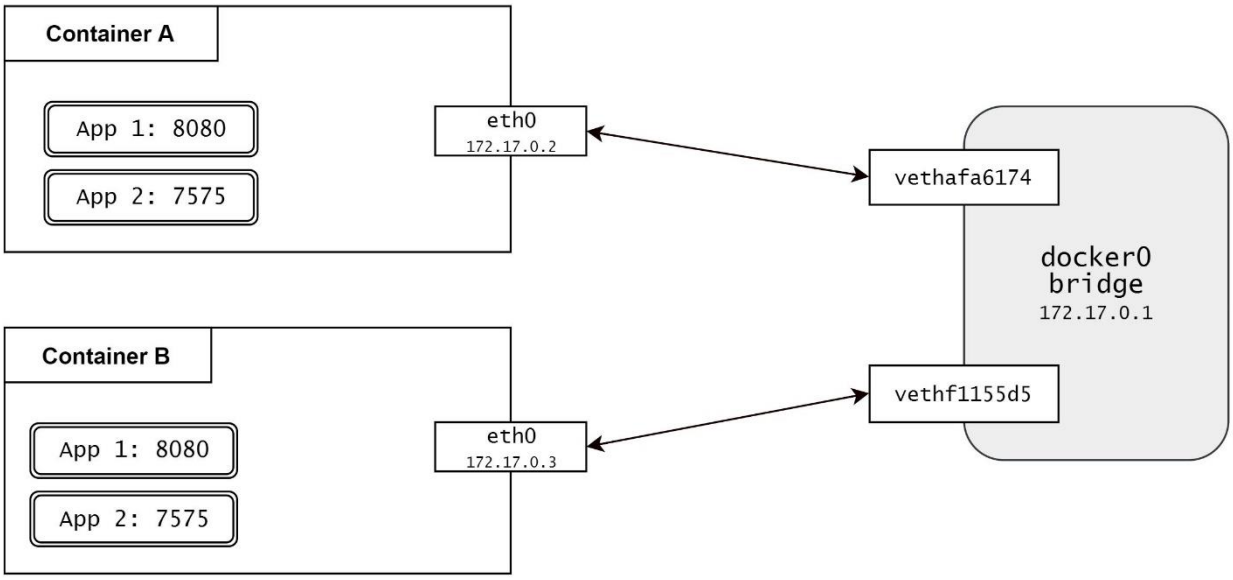

Figure 6: (a) Result of "brtcl" tool for showing bridge information. (b) A conceptual depiction of how the docker bridge interacts with containers. Each container has a virtual interface connected to the docker bridge on the host.

In the default bridge network mode, containers are given IP addresses within a subnetwork created by the Docker bridge. Using the IP address, containers can communicate amongst each other within a host. To communicate to containers outside the host, have specific rules on which containers can communicate with each other, or managing a cluster of hosts with containers would require user-defined networks. User-defined networks are out of the scope of this paper. IP packets are initially sent to the default gateway, which is the bridge in this case. The bridge then forwards the packet through one of the virtual interfaces, or passes the request to the host if no virtual interface is registered with that address. If a process is listening for requests on the host, containers can use the default gateway's IP address and the process' listening port to send requests to that process.

Figure 6b shows two containers running the same applications and listening on the same ports for requests. However, because each container has its own *net* namespace, the ports bind to the *eth0* interface of the container. Therefore, a request sent to 172.17.0.2:8080 will be handled by App 1 in container A. Network isolation allows for multiple instances of the same applications with reserved listening ports to serve requests in the same physical host.

When developing a distributed systems prototype, flexibility in networking is one of the major components. Considering a VM to develop a prototype means addressing overhead on the networking facilities. Since a VM is an isolated system, setting up a network requires changing configurations across all VMs. In contrast, a container can use existing network configurations upon instantiation. Although VMs offer much more sophisticated networking capabilities, most of them are unnecessary for high level distributed systems problems (involving just nodes and edges) and instead become an obstacle to rapid prototyping.

2.2 Shared Data

In distributed computing environments, nodes require consistency on the state of shared data values. This fundamental requirement can be non-trivial to achieve in a realistic model where processes can fail or communication may be unreliable. To build a fault tolerant system, such individual failures must be tolerated and overall system consistency must be maintained.

2.2.1 Replicated State Machines

State machines [4] can be used to model nodes in a system which consist of state variables, and commands to modify the state variables or produce an output. Clients of a state machine can make requests to execute a command. Requests include the name of the command and arguments needed by the command. Replicated state machines coordinate with each other to compute identical outputs and states. Therefore, input commands must be executed in the same sequence across all machines, and consistency in replications must be held regardless of node failures. Replicated state machines are applied as solutions to a variety of fault-tolerance problems that require coordination in a distributed system. Machines are typically synchronized using a replicated log where the series of commands to be executed are stored, shown in Figure 7.

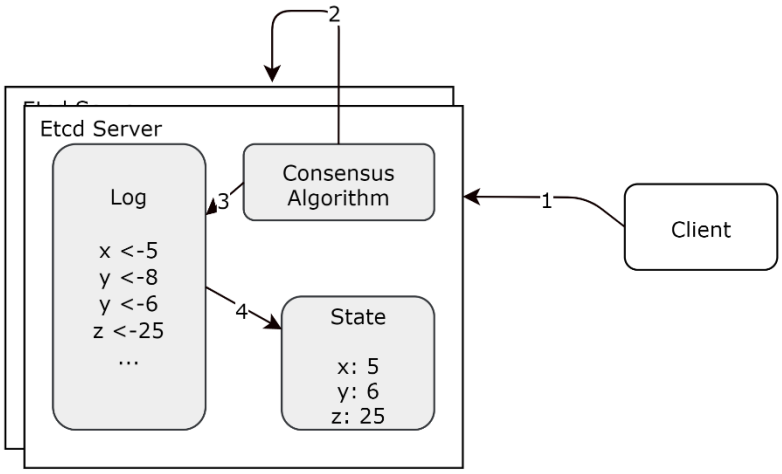

Figure 7: An Etcd Replicated State Machine. 1) Client sends log command. 2) Consensus algorithm executes with other servers to agree on the next entry. 3) Agreed entry inserted to log. 4) Log entry command executed on state.

When a machine receives commands from clients, it appends it to its log and communicates with the other replication machines to ensure all logs contain the same sequence of commands. This can be achieved using consensus [5], where nodes (processes) in a system advertise their local data values and communicate with other nodes to agree on a single data value. In this context, consensus is used to order input commands to a set a of state machines to guarantee a consistent system output. Each process p_i in a system S (i = 1, 2 …N) begins with an *undecided* state and proposes a value v_i. The processes communicate and vote to enter the *decided* state on the value and reach consensus. The following properties must be held for the execution of a consensus algorithm:

- Termination – Eventually every valid process must decide a value.
- Agreement – Every valid process must agree on the same value.

- Integrity – If a value is proposed, then any valid process in the *decided* state has chosen that value.

Although the Paxos algorithm has become synonymous with consensus protocols, it is difficult to comprehend, and thus complex to implement. As an alternative to address these problems, the Raft protocol was introduced by Ongaro et al [6]. Raft implements consensus by electing a leader in the system that manages the replicated log. All log entries flow from the leader to the other servers, simplifying the entry propagation. If the leader process fails, then an election is held for a new leader.

2.2.2 Raft

Nodes participating in a raft protocol can be in one of three states: *leader*, *follower*, or *candidate* for leader. There is always only one leader in a cluster of machines, who handles all the client requests. If a request is sent to a client then it is redirected to the leader for processing. The leader sends periodic heartbeat messages (in the form of Remote Procedure Calls) to its followers to maintain leadership. If a follower does not receive any messages from the leader for a predetermined amount of time, then it assumes the leader has failed and begins an election process. This amount of time is a protocol parameter known as the *election timeout*. The follower transitions into candidate state, votes for itself and requests votes from the other nodes. If a majority of the nodes in the cluster vote for the candidate, then it wins the election. Otherwise it will receive a heartbeat message from another node that has already established itself as leader. If no candidate can gain the

majority then the candidates will timeout and a new election will start. The Raft protocol has been implemented in various languages such as Go, C++, Java, and Scala.

2.2.3 Etcd

Etcd is a distributed key value store that implements the Raft protocol to maintain consistency in the state of a distributed system cluster [7]. Using Etcd, a cluster can store global configuration values required for a coordinated execution. Since Etcd runs in a cluster of instances with replicated logs, the shared configuration data is always available. With this architecture, nodes in a distributed environment can share data at a logically centralized but physically distributed location. See Figure 8. In a Linux/Unix OS, the directory "/etc" contains configuration files for the system, and the suffix "d" is for distributed. The core design goal of Etcd is to provide an underlying layer of coordination for large scale systems that can't function without a distributed synchronization. Applications using Etcd are commonly implemented on containers, since containers are running in large numbers in modern cloud applications. Furthermore, with the adoption of the microservices architecture (Section 3), cloud services have been modularized into multiple containers that were once tightly coupled into a monolithic application running in a VM. Some notable implementations in production include distributed data storage, container task schedulers, container monitoring, and service discovery. Kubernetes [11], for example, provides production container orchestration which is essentially a suite of the Etcd implementations discussed above.

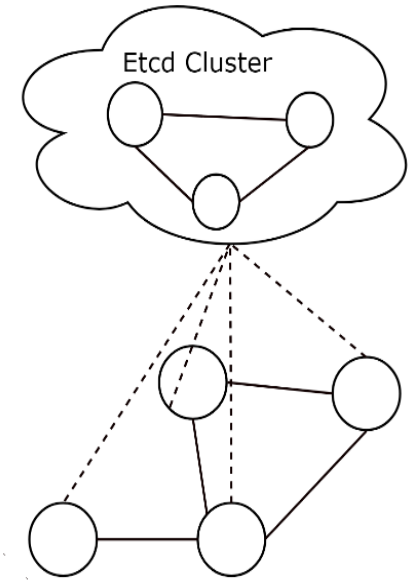

Figure 8: Nodes have a logically centralized location for the log, however it is physically decentralized.

Etcd operates in multiple instances to maintain availability, these instances must first be initialized with the address of other instances to begin operation. If the nodes, their addresses, and the size of the cluster is known, then a configuration file can be created for a static bootstrap. Otherwise, instance discovery mechanisms may be used to locate the other instances to join an Etcd cluster. Once a cluster is online, clients can begin sending HTTP requests to the advertised client URL. The communication interface with the cluster is an open-source Remote Procedure Call (RPC) framework called gRPC [8]. RPC works by allowing clients to invoke methods on an object that resides on another machine as if it were local. On the server side, methods on the objects are implemented and an interface is exposed to handle client calls. The client uses a stub, which acts as a local proxy to method

calls on the server, and has the same method signatures exposed on the server interface. See Figure 9. To serialize the structured objects, gRPC uses protocol buffers [9].

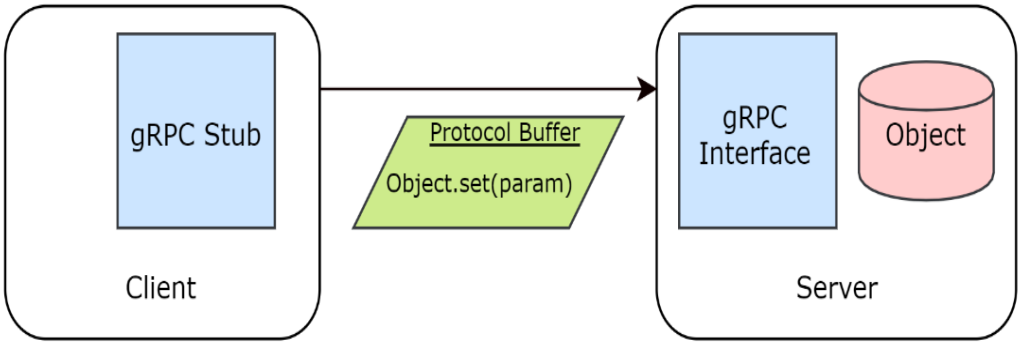

Figure 9: The client is setting parameters on a non-local object by using the gRPC call. The gRPC stub contains the methods exposed on the server interface.

Using the gRPC framework, Etcd can provide high level primitives that can be used at the application layer such as distributed locks, barriers, and leader election. These primitives are implemented in the case studies of this thesis to develop a fault-tolerant system. The core interfaces of interest are [10]:

- Read
    - *Range* – gets the Keys from <K,V> store.
    - *Watch* – Events happening or that have happened can be watched on multiple key ranges at once. Watch can be used as an event based trigger.
- Write
    - *Put* – puts the given <K, V> into the store, an event for the key is generated.

- *Delete* – deletes a given range from the <K, V> store, and also generates an event.

With these core operations, the case studies in this paper will demonstrate how Etcd can be used to provide resilience in a distributed system. With multiple nodes in an Etcd cluster there's no need to address the problem of single point failures, and still have the advantage of a logically centralized log.

2.3 Go

Go is a systems programming language initiated at Google, and has become an open source project since 2009 [12]. The motivation for its conception was to address problems introduced by multicore processors, network dependent systems such as data centers, web programming models, and large-scale software development processes commonly used today. Key design principles of Go include:

- Ease of learning for new developers through clear and concise syntax and semantics
- Build time reduction by efficient package dependency management
- Meeting modern computing demands (built-in concurrency, networking, and ease of web application development)

Go's syntax bears resemblance to C, with advantageous adoptions from dynamic typed languages for variable declaration and initialization. Go, however, is statically typed and is compiled to native code for execution which provides the performance and safety of languages such as C++ and Java. Pointers are part of the language, but pointer arithmetic is not allowed. The language runtime automatically manages memory allocation and garbage collection [13].

Although Go can't be classified as an Object-oriented programming language, it has concepts analogous to objects which are defined as type structs and methods that can be invoked on them. There no notion of inheritance and hierarchy, which reduces the overhead required to declare relationships between types. Instead, types satisfy interfaces by implementing a subset of their methods. Unlike object-oriented inheritance which allows only one parent, types can satisfy multiple interfaces at once.

2.3.1 Networking for Distributed Computing

With Go's support for distributed computing, applications can be implemented with just the core language. For example, the "net" package is included in the standard library and it provides access to low-level networking primitives such as TCP/IP, UDP, Unix domain sockets, and domain name resolution [14]. Many clients will only utilize the basic communication interfaces, such as Listener, Dial, and Conn.

Package "net/http" provides client and server interfaces for communication at the application layer, and is designed for simplicity in assigning HTTP routes as exemplified below. Figure 10 is a trivial implementation of a web server that responds with "hello, world!' when a HTTP GET is requested at the specified route ":8585/hello". Configuring REST routes is done by adding handlers that implement an interface that accepts http.ResponseWriter and *http.Request as arguments. For each incoming request, a new Go routine is created so that multiple connections can be served concurrently. This greatly reduces implementation efforts in managing connection threads for incoming requests.

```
package main

import (
    "io"
    "net/http"
)

// hello world, the web server
func HelloServer(w http.ResponseWriter, req *http.Request) {
    io.WriteString(w, "hello, world!\n")
}

func main() {
    http.HandleFunc("/hello", HelloServer)
    http.ListenAndServe(":8585", nil)
}
```

Figure 10: HelloServer implementation.

2.3.2 Concurrency

The key feature of Go that makes it suitable for distributed systems modelling is its core support for concurrency. Concurrency is identified as the composition of multiple independently executing processes. Although parallelism may be a related concept, the two are not the same. Parallelism, in contrast, is a simultaneous execution of processes that distribute a large computation into sub units. In a single processor, a program can be concurrent, however it cannot be parallel. In a distributed system, nodes may be handling multiple communication channels during execution, which makes concurrency a fundamental requirement in the design of such a system.

In many concurrent programming environments implementing correct synchronization to access shared data can be challenging. Go's approach is passing the reference of the shared data around, instead of all threads having the reference and coordinating the read and writes. By passing the reference, only one thread will have access to the data at any given time therefore eliminating data-races between threads that use channels. Go's authors have summarized this design paradigm to a slogan, "do not communicate by sharing memory; instead, share memory by communicating". It should be noted that this paradigm is used alongside mutex locks and not replace it. In a trivial reference counter, a mutex around the counter would be the clear solution. Go's concurrency model originates from Hoare's "Communicating Sequential Processes" [15]. The two fundamental constructs for concurrency are goroutines and channels, which are described in the following sections.

2.3.2.1 Goroutines

Goroutines are described as lightweight threads that run concurrently with other goroutines and asynchronously with the calling code. An independently executing function may return control to the caller before completion, these functions are known as coroutines [16]. Essentially the concept of goroutines is to multiplex coroutines on to a smaller number of OS threads which results in a N:1 mapping of coroutines and threads. When a system blocking call is made by a coroutine, the rest of the coroutines on the blocked OS thread are automatically moved to another, active, OS thread by Go's runtime which manages the scheduling. By migrating, the rest of the routines will not be blocked. Typically, the number of OS threads is set to the number of cores on the system to maximize CPU utilization. Goroutines have a significantly reduced memory footprint compared to OS threads. When a goroutine is created, Go's runtime allocates a few kilobytes to a resizable and bounded stack, which then grows or shrinks as per the execution demands. With this architecture, goroutines are very lightweight and a Go program can run hundreds of thousands of goroutines whereas traditional thread-based concurrency would run out of resources.

To create a goroutine, the syntax begins with the "go" statement followed by a function call. It is common practice to call anonymous functions, known as a function literal, as goroutines as they will not be called anywhere else in the program. Function literals are closures, which allows them to refer to variables in the surrounding function. See Figure 11 for an example of goroutine syntax.

```
go list.Sort()  // runs list.Sort concurrently; doesn't wait for completion.

func Process(message string, delay time.Duration) {
    go func() { //literal does not name func
        time.Sleep(delay)
        fmt.Println(message) //Message is within the variable scope of the
caller; accessible to the goroutine
    }()  //parentheses calls the function, it may include input arguments.
}
```

Figure 11: Goroutine example.

Unlike regular function calls, goroutines are asynchronous in the sense that the main program execution does not wait for the invoked goroutine to reach completion. An invoked goroutine will independently complete the function call and exit. Input parameters are evaluated as regular functions however return values are discarded. Because of this independent execution, the main program may complete without knowing the state of the invoked routine. In order to allow communication and synchronization between goroutines, Go uses channels.

2.3.2.2 Channels

Channels are the mechanism for inter-process communication between concurrently executing goroutines. They are explicitly typed and bidirectional, however they can be used unidirectionally by using a send-only or receive-only directive as shown in Figure 12. Channels are unbuffered by default, which provides synchronization of states between goroutines on opposite ends of the channel. In an unbuffered channel, senders block until a receiver has retrieved the value from the channel before sending additional data. Receivers always block until there is data to be received, regardless of whether the

channel is buffered or unbuffered. A buffered channel, removes the synchrony as senders can now send data asynchronously until the capacity of the channel has been reached.

```
//The channel is receive-only
func consumer(ch <-chan int) int {
    return <-ch //Blocks until value is received
}

//The channel is send-only
func producer(i int, ch chan<- int) {
    ch <- i
}

func main() {
    ch := make(chan int) //Initializes an unbuffered channel of type int
    go producer(99, ch)
    result := consumer(ch)
    fmt.Println(result)
}
```

Figure 12: An example of goroutine communication via channels.

2.3.2.3 Go Race Detector

Programs written with goroutines typically share global variables where reads and writes can be performed concurrently, therefore synchronization is crucial. Data races are notoriously difficult to detect and even with Go's support for idiomatic concurrent programming, race conditions are not eliminated. Many failures occur erratically and can be difficult to reproduce for analysis as they occur depending on the timing of the goroutines. There was an unnecessary amount of time spent debugging such errors in this thesis. Figure 13 shows a simple yet elusive example using a timer.

```
func main() {
     start := time.Now()
     var t *time.Timer
     t = time.AfterFunc(randomDuration(), func() {
         fmt.Println(time.Now().Sub(start))
         t.Reset(randomDuration())
     })
     time.Sleep(5 * time.Second) //Keeps main thread alive for 5 seconds
 }

func randomDuration() time.Duration {
    return time.Duration(rand.Int63n(1e9))
}
```

Figure 13: Timer data race example.

In this example, a timer is started to print a message after a random duration which repeats for five seconds. Printing the message is encapsulated in a function literal which also resets the timer to schedule the next print message. The function literal is called as a goroutine, and every reset reuses the timer. This program runs as expected almost all the time, however in between random runs it may crash with the following errors in Figure 14:

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xb code=0x1 addr=0x8 pc=0x41e38a]

goroutine 4 [running]:
time.stopTimer(0x8, 0x12fe6b35d9472d96)
    src/pkg/runtime/ztime_linux_amd64.c:35 +0x25
time.(*Timer).Reset(0x0, 0x4e5904f, 0x1)
    src/pkg/time/sleep.go:81 +0x42
main.func·001()
    race.go:14 +0xe3
created by time.goFunc
    src/pkg/time/sleep.go:122 +0x48
```

Figure 14: Null pointer error.

A null pointer error crashed the program after a goroutine reset the timer. To debug the error, it must first be reproduced. However, since it's a random sequence of execution,

reproducing the error becomes non-deterministic. To analyze such bugs Go has included a data race detector, which can be enabled by the "-race" flag when running a Go program. Figure 15 shows the output of using this tool on our sample program.

```
==================
WARNING: DATA RACE
Read by goroutine 5:
  main.func·001()
     race.go:14 +0x169

Previous write by goroutine 1:
  main.main()
      race.go:15 +0x174

Goroutine 5 (running) created at:
  time.goFunc()
      src/pkg/time/sleep.go:122 +0x56
  timerproc()
     src/pkg/runtime/ztime_linux_amd64.c:181 +0x189
==================
```

Figure 15: Data race warning.

A data race has been detected on the timer variable *t*, where two goroutines are concurrently reading and writing without any synchronization. The null pointer error is caused during the initialization of *t* for very small random durations because the timer's goroutine function has finished execution and invoked reset before variable *t* is assigned a value. Such errors are very difficult to detect when working with concurrency as they can arise in any context. Go's race detection support is more than just a convenient tool, it plays a crucial role when building robust programs.

3 Microservices

Service Oriented Architecture (SOA) was coined in 1996 by Roy Shulte and Yefim Natis [17]. It's a basis to design multi-tiered computing platforms that enforces loose coupling between tiers by using interfaces. SOA is the replacement of the traditional application architectures which have remained tightly coupled over time. The microservice architecture is a modern interpretation of SOA which defines fine-grained services as processes communicating over the network. Microservices are enabled due to the advancements in software containerization technology such as Docker. They address the drawbacks of a traditional monolithic architecture which will be compared in this section This section will describe microservices including two of its relevant features, rapid prototype deployment, and service orchestration. It will also present a case study, where our framework was used to design the solution in a joint project with UOIT and a local software firm specializing in providing commercial business solutions.

3.1 Service Oriented Architecture

SOA can be described as a loosely-coupled software architecture where business logic is separated into indivisible units of services [18]. Services are accessed through advertised interfaces that define what the service accepts as input and what the user can expect as output. They are deployed throughout the network and communication protocols are pre-defined by the service providers. SOA applies the modular programming paradigm which helps businesses focus on developing the logic. Figure 16 provides a concrete example. Though the principles of SOA are broad, the key characteristics are [19]:

- **Autonomous** – Services can independently maintain their internal logic without disrupting other services.
- **Stateless -** Services do not store state information as it can add dependency to its functionality. If state information is required, it is managed externally.
- **Abstraction via interfaces** – Underlying logic is considered irrelevant to the service users, and thus a service contract is exposed as an interface.
- **Reusability** – Services are designed to be reused.
- **Composable** – Service composition creates abstraction layers and can provide a higher-level service that combines single services.
- **Discoverable** – Interfaces exposed by services should be discoverable to potential service requestors.
- **Loose Coupling** – Key characteristic of SOA where services should execute without any cross-service dependencies.

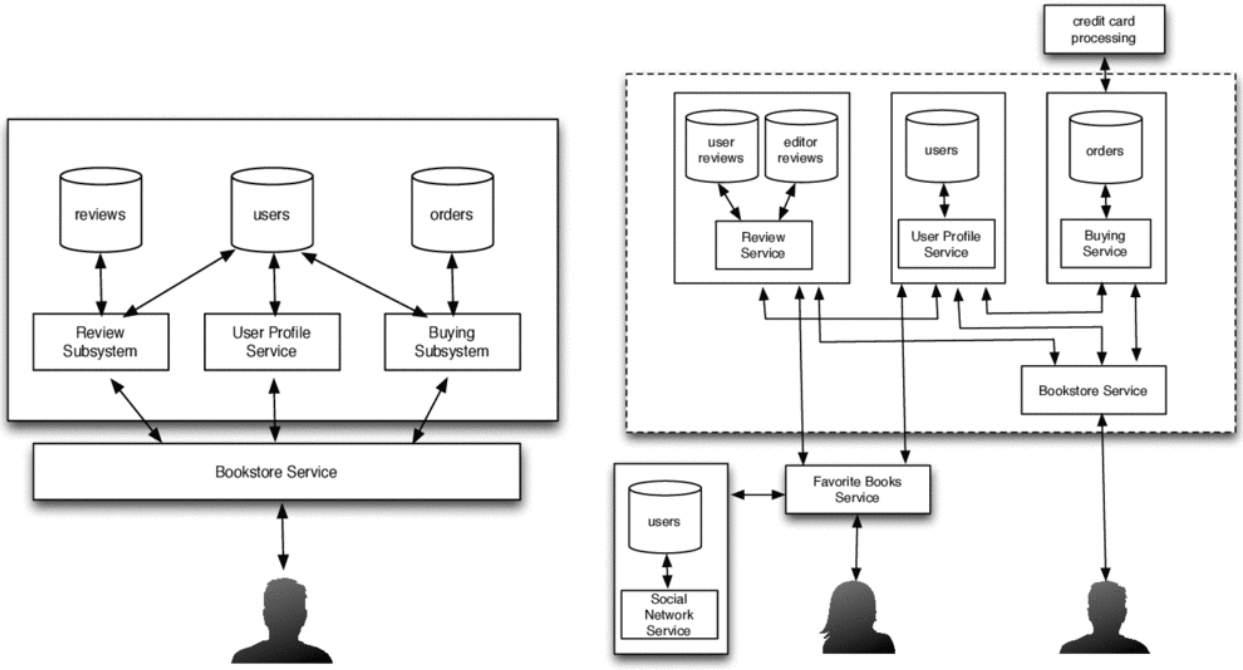

Figure 16: Difference between SOA and a traditional application in the form an implementation of a bookstore application with three main services, review, user profile, and buying.

Figure 16 [20] is an example showing the difference between SOA and a traditional application. It's an architectural design of a bookstore application with three main services, review, user profile, and buying. Figure 16 on the left side shows dependencies between the services, and the all the sub-systems are exposed through one aggregated API. Figure 16 on the right side shows a SOA implementation where all the sub-systems are independent. Each system is highlighted by the solid boundaries to show its separation. Now, for example, if the buying system wants access to the user's information, it can't access the users database directly. It must interact with the users service and invoke the correct exposed API.

3.2 Monolithic Applications

To understand the concept and necessity of microservices, a brief description of a traditional monolithic architecture is essential. When developing an enterprise level application, the three main components are presentation, application (or logic), and data persistence. Monolithic architectures combine all three components into a single-tier application that serves pre-defined requirements. As the components grew, the tiers were separated for development and integration efficiency, however they are still very tightly coupled. This leads to multiple drawbacks [21]:

- Code maintenance – Without strict module boundaries, it becomes challenging to make changes to the code base without affecting numerous dependencies.
- Large scale development – After growing to a certain size, applications are typically divided into teams that manage specific features. With a tightly coupled application, this practice becomes inefficient.
- Continuous deployment - A major challenge in monolithic architectures which hinders rapid updates.

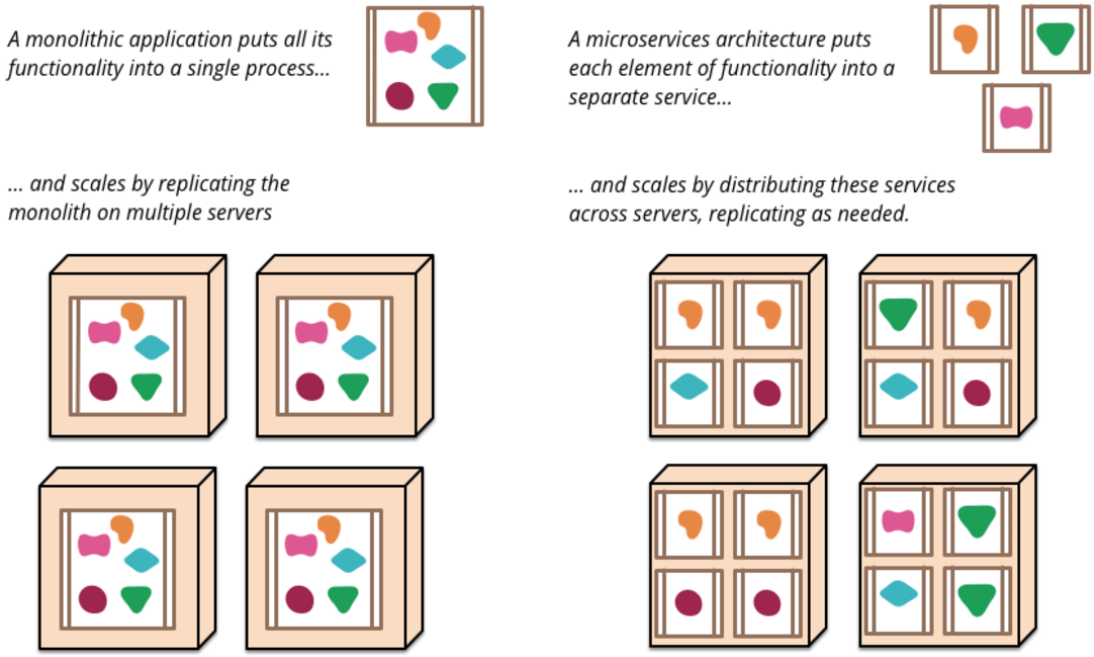

Figure 17: Difference between monolithic and microservice architectures.

The microservice architecture directly tackles the issues of monoliths with a design of loosely coupled, self-contained services. Figure 17 [22] depicts a high-level contrast between monoliths and microservices. As shown, microservices can scale more efficiently and can be deployed individually as there no cross dependencies with other services. Due to its modularity, maintenance and continuous improvements can be streamlined concurrently. Microservices follow SOA, but granulate the services to units of logic. This enables software engineering teams to take complete ownership of its service in the software development life cycle, as Amazon puts it “You build it, you run it” [23].

3.3 Performance and Orchestration

Microservices can be provisioned in various deployment patterns such as multiple instances per host, single instance per host, instance per VM, and instance per software container [24]. The latter pattern using containers has been picking up momentum in the industry due to its efficiency and performance. Containers, as described in Chapter 2, are much more lightweight compared to VMs. This yields rapid provisioning of service instances which simplifies frequent modifications to the service. Since services are a logically packaged unit, software containers provide the optimal platform as they are designed to be self-contained down the application binaries. Engineering teams working with software containers such as Docker gain development productivity by having completely isolated services. In production, each container can have fine-grained rules on allocation of system resources providing more control at the infrastructure level [25].

Scaling an application, using a single instance of service per container, requires a large and growing number of containers and hosts. A host can handle hundreds of containers based on the workload and available resources. Thus, it has become a necessity to automate container management from start to end of the container lifecycle. This is known as container orchestration, some of its key functionalities are [26]:

- Provisioning hosts, instantiating selected containers, and rolling out updates
- Container health checks, and re-instantiating failed containers
- Service discovery, using a Key-Value store such as etcd for registering containers' interfaces information
- Managing intra-host and inter-host networking
- Scaling up containers or down based on the service demands

Some of the most commonly used orchestration tools include Apache Mesos[27], Kubernetes [28], and recently introduced Docker Swarm that has been natively added to Docker container Engine.

3.4 Service Discovery

In this thesis, there was some exploratory work done on service discovery using Docker containers, etcd as the registry, Go for service implementation, and Registrator a service registry bridge. Registrator registers active services running on Docker containers, and deregisters services that have been detected as offline [29]. As a service bridge only, it does not support other features of a complete orchestrator. Figure 18 shows the high-level setup of the implementation.

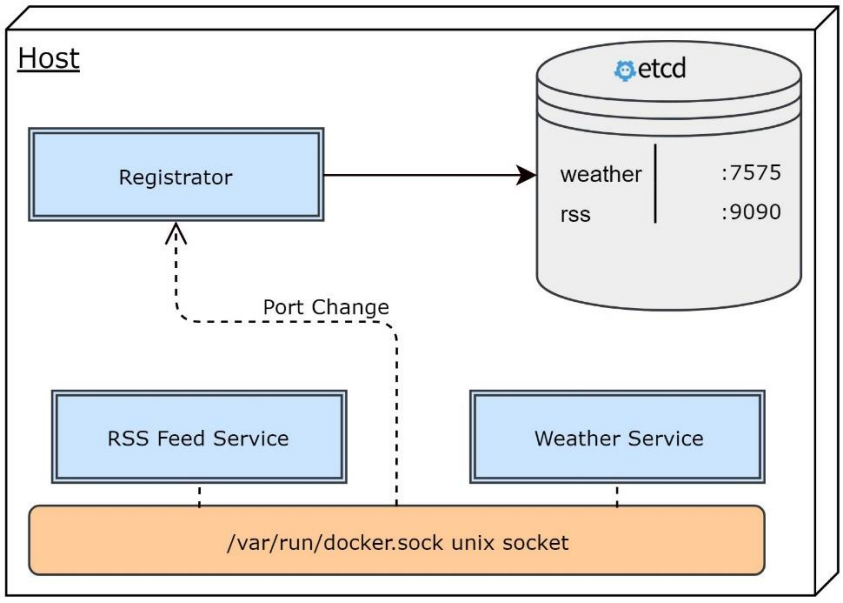

Figure 18: Architecture of implemented service discovery.

In this setup, we start with two simple services implemented in Go. RSS Feed service can fetch from any generic API such a news, social media, or open data. Weather service is implemented to fetch local weather from Environment Canada [40] whenever a service request is invoked. Services are packaged and deployed as Docker containers on a single host. They expose ports that are interfaces to the service, the ports are mapped to a host port during container instantiation using the following syntax:

```
docker run --rm -p 7575:7575 --name weather wserv
```

Here a container named “weather” has been linked to port 7575 from the host to the container, thus the service can be accessed using the host’s IP address and mapped port. Before service instantiation, Registrator is started also as a container with a shared volume with the “docker.sock” file which for inter-process communication between containers. Registrator detects any published ports, or ports taken offline and updated the etcd registry accordingly.

```
parth ~ $ docker run  --rm  --name=registrator     --net=host     --
volume=/var/run/docker.sock:/tmp/docker.sock     gliderlabs/registrator:latest
etcd://172.17.0.1:2379
2017/08/07 03:00:45 Starting registrator v7 ...
2017/08/07 03:00:45 Using etcd adapter: etcd://172.17.0.1:2379
2017/08/07 03:00:45 Connecting to backend (0/0)
2017/08/07 03:00:45 Listening for Docker events ...
2017/08/07 03:00:45 Syncing services on 1 containers
2017/08/07 03:00:45 ignored: 2bbb3b5225bb no published ports
2017/08/07 03:00:45 ignored: 2bbb3b5225bb no published ports
2017/08/07 03:01:09 added: abb8c6e5ee65 parth-Y700:weather:7575
2017/08/07 03:01:44 removed: abb8c6e5ee65 parth-Y700:weather:7575
```

Figure 19: Registrator Output

Figure 19 shows the output of running the Registrator container. Etcd's address is given as a parameter during instantiation, then the weather service is started. Registrator successfully adds the service under the hostname, service name, and interface port. If the weather service fails or is stopped, then it is automatically detected and the port mapping is removed from etcd. Figure 20 shows the output when querying an etcd cluster.

```
parth ~ $ etcdctl ls -r
/wserv
/wserv/parth-Y700:weather:7575
```

Figure 20: Etcd registry query.

With this prototype, the simplicity and effectiveness of service discovery can be observed. Other services on the network can query the etcd registry on this host to know what services are offered at which interfaces. When there are many services running on a host, automatic detection of active and non-active services can reduce the effort of dynamically configuring service interfaces.

3.5 Use in Distributed Systems Research

Light-weight containerized services compose the ideal testbed for modern distributed systems research. With the ubiquity of today's service infrastructure on the internet, microservices can model various distributed computing scenarios. The researcher should only focus on developing the logic. Deploying the service can easily be performed by packaging all dependencies into one image, and starting several Docker container instances within the order of minutes. This enables rapid prototyping iterations needed

during preliminary research. Cooperation within a research project is also streamlined with the complete isolation of services.

3.6 Document Management Case Study

Our proposed testbed in this thesis was applied on a joint industry project between UOIT and a local software firm. The firm will remain unnamed, and the project will be called ResourceManager in this thesis for simplicity. The objective of ResourceManager's functionality was to provide a secure federated collaboration platform that lets organizations share important resources from their local repositories. After developing a high-level design for a solution based on services, this section asserts the benefits of the proposed framework discussed in previous sections.

3.6.1 Requirements

For simplicity, only the most essential requirements are outlined from the project. At the architectural level, there are three main sub-systems at minimum that represent the main functionality. The ResourceManager federated server, and two clients wanting to collaborate on a project while sharing resources from their local repositories. See Figure 21 for the architectural view. Majority of the logic resides on the federated server which requires the following features:

- Federated authentication and secure collaboration: Requests from clients must be authenticated by the system prior to any transactions. Once authenticated, the

consequent transactions with the server must remain secure including resource collaboration sessions.

- Logs: ResourceManager server must log all transaction records for data analysis and reporting for audits.
- Data Persistence: User credentials, logs, and active session data must be stored in a database.
- User Interface: Administrators should be able to view the server details through a web application.

The client must implement logic to interact with the server and then other clients after authentication. Some features include:

- Repository Integration: Organizations will have different document repository software, thus ResourceManager must integrate with the repositories to enable access to the resources.
- Authenticated Communication: The client must obtain authentication from the server before engaging in collaboration with other clients.

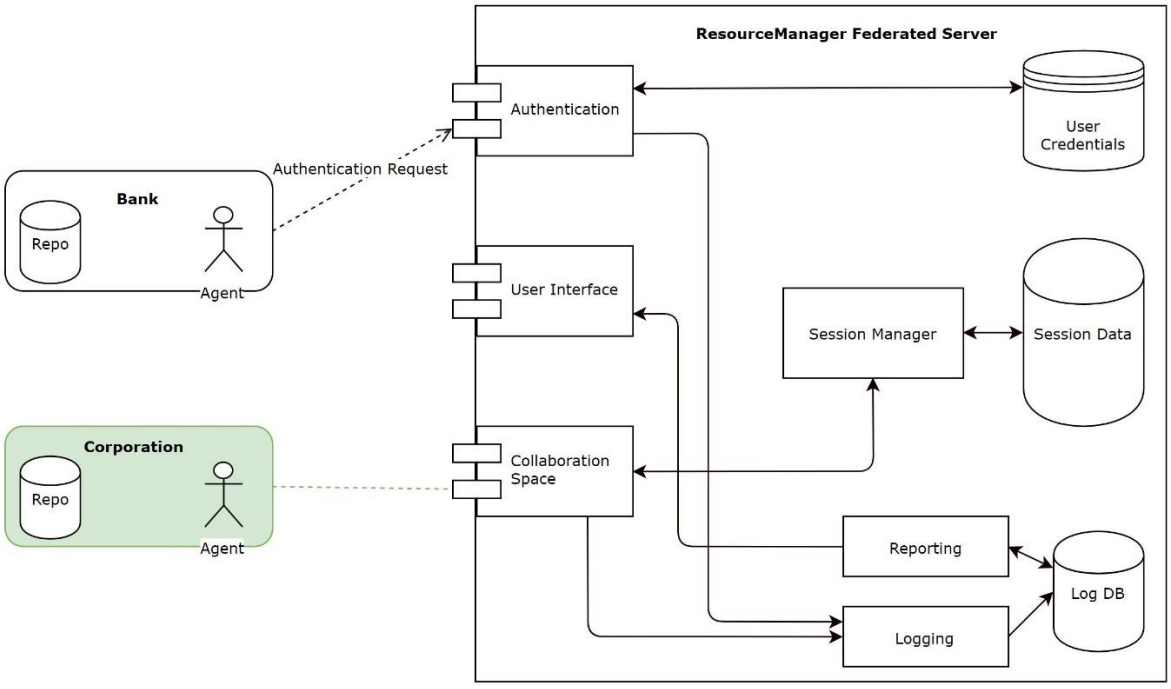

Figure 21: ResourceManager architecture.

UOIT's contribution for this project was to develop a prototype architecture and implementation that can be used to analyze potential issues in a production grade solution. Based on the requirements, Figure 21 was the high-level architecture developed that highlights the main logic modules. Secure communication was the major concern as it is noted that resources being shared had information sensitive content. To address this issue, the authentication module would generate security tokens for authenticated clients. The client would then use this token for subsequent transactions with other clients and collaboration space in an authorized manner. After developing a high-level concept of secure communication, the next step was to determine a technology stack where Docker and Go played a pivotal role.

Designing this system as a microservice architecture will provide development efficiency with coarse requirements, rapid prototyping, and ability for fast deployment. Since the requirements are based on logically separated units, it would be effective to develop isolated services. Each service can be developed as the project demands, starting with the minimum required services for functioning then adding additional features as the project scales. Another important consideration is that the development team for this project is multi-organizational. Therefore, services should be self-contained and portable to allow teams to work with minimal setup time.

With Go's approach to simple and concise syntax, newcomers to the project can pick up the language quickly. Its networking features are ideal in a microservice communications environment. Interfaces can easily be defined for individual functions required for each module. It is important to note that with containerized services, the technology stack can vary from container to container. This characteristic is desirable in such a project where technology stack is still undetermined and experimenting with different stacks is part of the design phase. Only the interface defined at the container level through IP and port is exposed, so the internal logic is completely abstracted. In contrast to a monolithic architecture, the modular approach can let independent co-op students work on services that don't have cross-dependencies. Due to Docker's light-weight design, hardware resources in the form of servers are minimized during the design phase of the project. Unlike VMs, any developer can setup the entire server composed of different service containers on a local machine.

4. GHS Algorithm Implementation

4.1 Description

In a connected graph, $G = (V, E)$ with weighted edges, a minimum spanning tree (MST) is a subset of the edges from G that creates a spanning tree T, minimizing the following weight function:

$$w(T) = \sum_{e \in T} w(e)$$

Graph G is undirected and sometimes assumed to have distinct edge weights so that the MST is a unique solution. A MST can be applied to a variety of disciplines where problems can be modeled as a graph data structure. An MST problem is typically given as [30]:

> A graph is modelled to express nodes as cities and edges as communication links, electric grid lines, or transportation routes. The edges have weights to represent the costs of construction, or length of the edge. Find the set of edges that would connect all the cities and have the minimum total edge weight. They must not cycle and therefore must form a minimum spanning tree as the solution.

In this thesis, the main focus is on communication links in distributed systems. Each link can be weighted using a variety of metrics such as bandwidth, delay, or reliability. Solving such a problem using an MST with a given network topology is no different than the problem described above. However, in a peer-to-peer distributed system where nodes are independently computing the MST path, the global topology is not known. Nodes are

only aware of the adjacent edges and their respective weights. This is drastically different from the traditional algorithms used to solve an MST. Nodes communicate via message passing to infer the state of the global network. The information inferred depends on the reliability of their adjacent neighbours. Distributed MST algorithms can solve for the most economical broadcast routes in networks which have costs associated with communication channels. Their distributed execution is crucial in P2P networks where topology changes due to peer churn and failure is uncontrolled. To compute the MST in a decentralized and asynchronous setting, we utilize Gallager, Humblet and Spira's distributed algorithm (GHS) [31].

The GHS algorithm uses fragments as the core idea. A fragment of an MST can be defined as a connected sub-tree within a forest of nodes and edges. MST algorithms initialize with individual fragments per node and terminate with the MST as the fragment. An important property of MSTs is:

> *Given a fragment of an MST, let e be a minimum-weight outgoing edge of the fragment. Then joining e and its adjacent nonfragment node to the fragment yields another fragment of an MST.* [31]

An edge is outgoing from a fragment if the adjacent node from this node is not part of the same fragment. Using this property, algorithms ensure cycles are not formed in the MST edges. For example, Prim's algorithm [32] begins with a single node and sequentially selects the least weighted edge outgoing from the current node to a node not part of the same fragment. Nodes are iterated until all nodes are in the same fragment. Kruskal's algorithm [33] uses a greedy approach by creating a set $S\{e \in G\}$ and removing the edge

with the minimum weight from S on every iteration until there is a single fragment. Initially each node starts as an individual tree within an empty forest *F{}*, each edge removal from set *S* connects two trees and adds it to the forest *F*, when the forest contains only one tree, the MST is found. Finally, the algorithm that bears the most resemblance to GHS is Boruvka's algorithm [34]. It initializes by assigning each node as its own fragment, and then iterations occur on a per fragment basis. From each fragment, the algorithm looks for the minimum weight outgoing edge, and combines the fragments. By iterating per fragment, each node within the fragment is iterated to check if it has an adjacent edge going out of the fragment. This slightly differs in other algorithms where iteration is per node or edge basis.

The GHS algorithm utilizes the concept of fragments but asynchronously finds the minimum weight outgoing edges through message passing. With a decentralized setting where the computation occurs separately in each node, identifying and combining fragments becomes non-trivial. The union of two fragments, in GHS, is determined by a property known as a *level* which is assigned to fragments based on previous unions. Summarizing the rules:

- A fragment is at level L=0 when it contains only one node
- If fragment F is at $L \geq 0$ and the adjacent F' is at L' then:
    - $L<L'$ will result in F is absorbed into F'. See Figure 22.
    - If $L=L'$ and the two fragments have the same minimum weight outgoing edge then they combine to a fragment at $L + 1$, the edge is now called the

core. See Figure 23. If its not the same outgoing edge, then the fragments must wait for their levels to change.

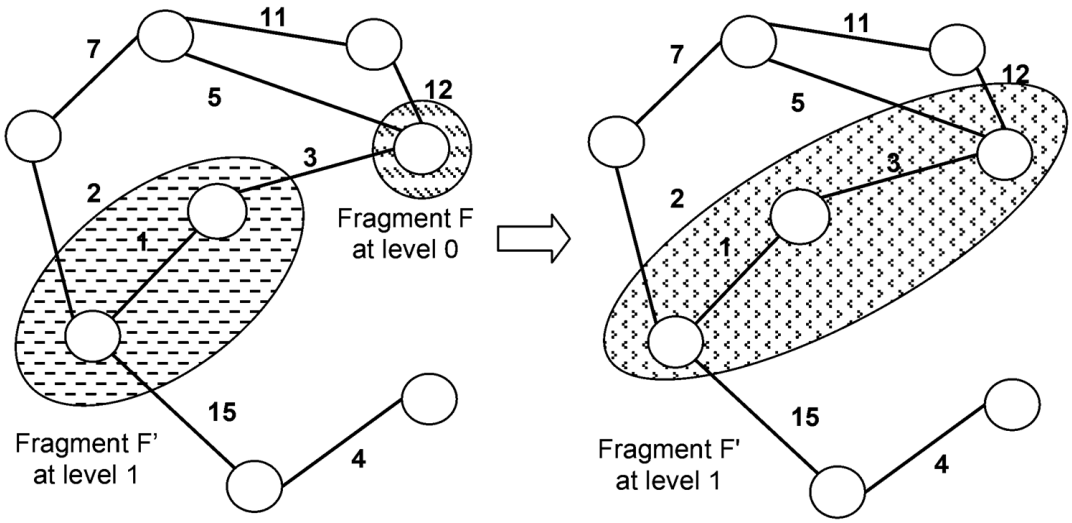

Figure 22: Fragment absorption.

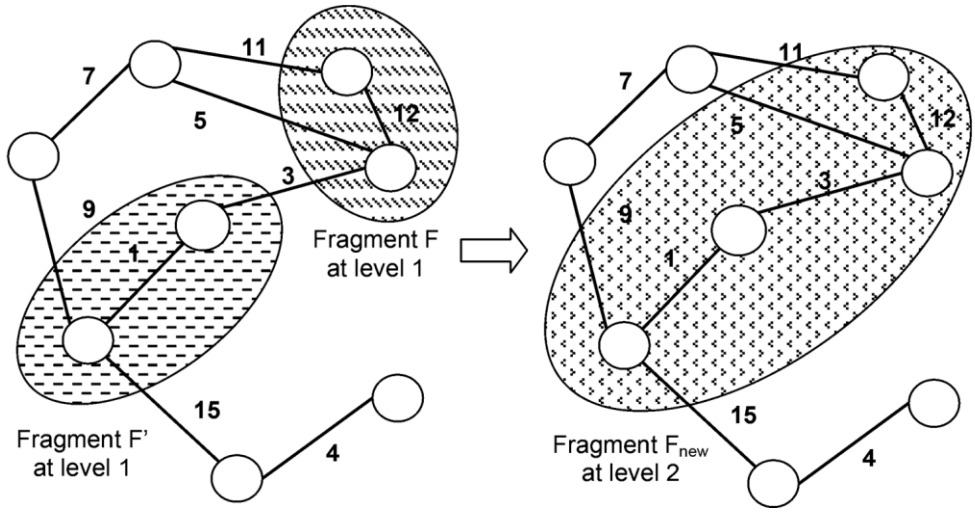

Figure 23: Fragment combination.

During the execution of the algorithm, node can be in one of three states:

1. *Sleeping* is the initial state of all nodes.
2. A node goes into state *Find* when searching for the minimum weight outgoing edge.
3. At all other times, its in state *Found.*

For every edge $e = (u, v)$ both nodes *u* and *v* save the state of the edge based whether it is part of the MST or not. Since both nodes store the edge state, it is possible for the edge states at the two nodes to be temporarily inconsistent. The possible states are:

1. *Basic* when the algorithm has not determined if it's a *Branch* or *Rejected*
2. *Branch* means the edge is part of the MST in the fragment
3. *Rejected* means the edge is not part of the MST

During initialization of the algorithm, all nodes are in the *Sleeping* state and all edges are in the *Basic* state. The algorithm, executed at each node, is shown below in pseudocode. There are variables to store the state of the nodes and edges, level of the fragment, and a fragment identity. Incoming messages are queued to preserve a first in, first out (FIFO) order, and then processed according the algorithm. To begin the algorithm, a random node must be chosen to be spontaneously awakened.

(1) Response to spontaneous awakening (can occur only at a node in the sleeping state)

execute procedure *wakeup*

(2) procedure *wakeup*

```
begin let m be adjacent edge of minimum weight;
SE (m ) ← Branch;
LN ← 0;
SN ← Found;
Find-count ← 0;
send Connect(O) on edge m
end
```

(3) Response to receipt of *Connect(L)* on edge j

```
begin if SN = Sleeping then execute procedure wakeup;
if L < LN
 then begin SE (j) ← Branch;
            send Initiate(LN, FN, SN) on edge j;
            if SN = Find then
               find-count ← find-count + 1
      end
else if SE(j) = Basic
      then place received message on end of queue
      else send Initiate(LN + 1, w(j), Find) on edge j
end
```

(4) Response to receipt of *Initiate* (L, F, S) on edge j

```
begin LN ← L; FN ← F; SN ← S; in-branch ← j;
      best-edge ← nil; best-wt ← ∞ ;
      for all i ≠ j such that SE(i) = Branch
         do begin send Initiate(L, F, S) on edge i;
                  if S = Find then find-count ← find-count + 1
            end;
      if S = Find then execute procedure test
end
```

(5) procedure *test*

if there are adjacent edges in the state *Basic*

then begin *test-edge* ← the minimum-weight adjacent edge in state *Basic;*

send *Test(LN, FN)* on *test-edge*

end

else begin *test-edge* ← *nil;* execute procedure *report* end

(6) Response to receipt of *Test(L, F)* on edge j

begin if *SN* = *Sleeping* then execute procedure *wakeup;*

if L > *LN* then place received message on end of queue

else if *F* ≠ *FN* then send *Accept* on edge j

else begin if *SE (j)* = *Basic* then *SE (j)* ← *Rejected;*

if *test-edge* ≠ *j* then send *Reject* on edge j

else execute procedure *test*

end

end

(7) Response to receipt of *Accept* on edge j

begin *test-edge* ← *nil;*

if w(j) < *best-wt*

then begin *best-edge* ← *j; best-wt* ← *w(j)* end;

execute procedure *report*

end

(8) Response to receipt of *Reject* on edge j

begin if *SE (j)* = *Basic* then *SE (j)* ← *Rejected;*

execute procedure *test*

end

(9) procedure *report*

if *find-count* = *0* and *test-edge* = *nil*

then begin *SN* ← *Found;*

send *Report(best-wt)* on *in-branch*

end

```
(10) Response to receipt of Report(w) on edge j
        if j ≠ in-branch
          then begin find-count ← find-count - 1
                    if w < best-wt then begin best-wt ← w; best-edge ← j end;
                    execute procedure report
                end
          else if SN = Find then place received message on end of queue
                else if w > best-wt
                          then execute procedure change-core
                          else if w = best-wt = ∞ then halt
```

```
(11) procedure change-core
        if SE (best-edge) = Branch
                then send Change-core on best-edge
        else begin send Connect(LN) on best-edge;
                   SE (best-edge) ← Branch
        End

(12) Response to receipt of Change-core
        execute procedure change-core
```

The GHS algorithm's pseudocode was implemented in Go where each node was running in a separate Docker container. Peer-to-peer networks were created within the Docker networking space based on pre-determined test topologies. Theoretically this algorithm can be used a network where the network topology is constantly changing since nodes are only aware of their local neighbours and not the global topology. However, there is a lot of overhead required to manage how peer churn during the algorithm's execution will affect the results. For example, if the algorithm has finished execution, and

the network needs to recalculate the MST after some peers have left and joined the network, then there needs to be a mechanism to initialize the variables of the algorithm within all nodes in the network. We discovered these issues after implementing it as a network node instead of threads.

4.2 Discussion

Implementing the GHS algorithm as an independent network node was challenging due to the lack of global synchronization. Each node is independently managing its state variables, including the state of the edge, which becomes complex when there is inconsistency between the two nodes of the edge. Tracing the execution of the algorithm was also difficult since there is no view of the global network and each node must be analyzed one at a time. We also discovered an error with the GHS algorithm where the algorithm becomes deadlocked. Chou describes the bug in a manuscript [35]. After further probing, we found other proofs that acknowledge this error such as Moses et al in [36]. There is no known solution in the literature addressing the deadlock scenario in this algorithm. To find such a solution is non-trivial, and would require core changes to the GHS algorithm.

5 Detecting High Risk Links in Peer-to-Peer Networks

The objective of this case study was to detect high risk communication links in peer-to-peer networks using the message passing model. A high risk can be identified as a link between two nodes that are important in the network. We simulate network traffic and define important nodes as the ones with higher traffic than most nodes. Our proposed method of detection is desirable in peer to peer networks where live topology changes occur and link detection is decentralized where each node separately performs the computation. Peers can only communicate to their immediate neighbours, without information regarding the global network topology. The metric for evaluation is the message traffic in the network simulated as a random walk. Each node locally tracks the total number of messages received and transmitted per adjacent edge.

5.1 Traffic Simulation Using Random Walks

In a real-world application, network traffic flow depends on various factors such as bandwidth, whether the node is a server or client, or simply services being offered at a node. We however take a much simpler approach to avoid the overhead in simulating a real-world network since our methodology doesn't have a global network view but we need a ground truth to verify our solution. Each node initially generates a message and an adjacent edge is randomly chosen to send the message. Upon receiving a message, the recipient node again forwards the message to a randomly chosen neighbor and the initially generated messages hop across the network indefinitely. Formally, this is a stochastic process in the form of a random walk across a given state space. This process can be mathematically modelled and we can compute the distribution of the random walker's visits in each state.

According to the law of large numbers, the frequency of a node being in the path of a message hop (or walker visit) will become equal to the calculated distribution as the total number of hops increase. In our traffic simulation, the messages hop indefinitely until the node fails, or the network is terminated. Using the random walk model, simulations can be verified with computed ground truths.

Noh and Rieger introduce the random walk centrality C for each node which determines how fast the node can receive and disperse information across the network [37]. In our simulation, this translates into the proportion of messages passing through the particular node. We denote the degree, the number of adjacent edges, of a node i by K_i given by:

$$K_i = \sum_j A_{ij}$$

Where A represents an adjacency matrix for node connectivity, where $A_{ij} = 1$ if there is connectivity, 0 otherwise. In this scenario, the network is undirected and therefore $A_{ij} = A_{ji}$. A node i at time t randomly selects an adjacent edge with equal probability to send the message to at time $t+1$. Thus, the probability of the message being sent to node j next is given by:

$$P_{ij} = \frac{A_{ij}}{K_i}$$

Given this probability, we can now model the distribution of messages expected to pass through a given node *i* by the following equation:

$$V_i = \frac{K_i}{N} \times \frac{1}{2}$$

Here *V* is the distribution of messages arriving at this node, compared to the entire network and *N* is the total number of edges in the network. We half the probability as each edge is bidirectional. Using this model, we can compute the percent of messages that will arrive at each node in the network giving us a predictable traffic simulation. See Figure 24 as an example.

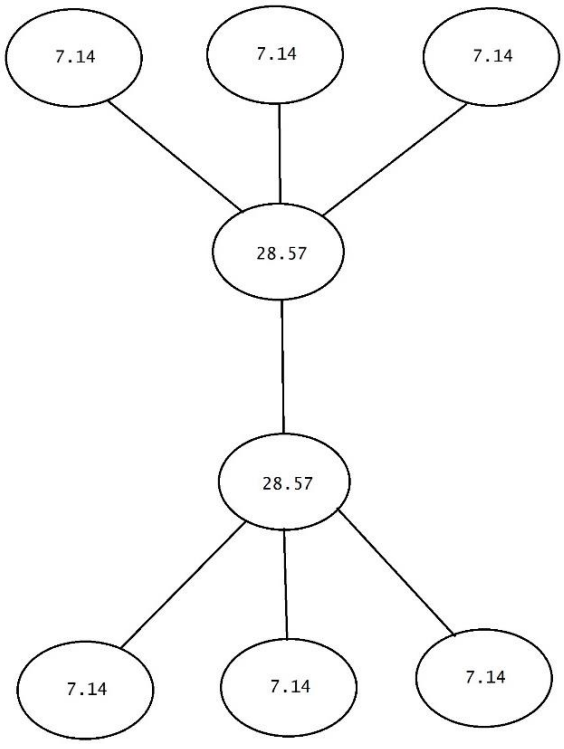

Figure 24: Message visit distribution is proportional to the degree of the node.

5.2 Network Aggregation with Push-Sum:

To discover the high-risk edges, traffic data from each node must be compared with other nodes in the network. Because the global topology is unknown, aggregation of the network traffic statistics must be distributed. This can be modelled as the following: in a network of n nodes, each node i holds a value x_i, compute some aggregate function of these values. In our context, x_i will hold the total number of messages received and transmitted on node i. The objective is for each node to determine whether it's a high traffic node or not based on the traffic of the rest of the nodes, therefore, our aggregate function of interest is the average.

A potential approach may be to assign a leader in the network that collects the data from all nodes and aggregates the statistics. However, this node can be a central point of failure and can leave the network aggregation in an inconsistent state. If the system grows, the leader may become a bottleneck itself. The network also becomes centralized which is not the ideal solution in a peer-to-peer network. Therefore, the aggregation must be decentralized, and this can be achieved using gossip-based (or epidemic) protocols for network average aggregation.

In gossip-based protocols, each node communicates with one or more adjacent nodes in each round, and exchanges information with these nodes. Gossip communication doesn't rely on central coordination, and is highly fault-tolerant in disseminating information across the network. It's probabilistic by nature instead of absolute guarantees provided in other forms of data dissemination protocols. Since we are using random walks in our simulations, we can calculate the probability of a node receiving information in a uniform gossip protocol with the same model.

5.2.1 Push-Sum Description

Push-Sum protocol computes averages of values at the nodes in a decentralized gossip based aggregation [41]. The push-sum protocol can be defined as follows: At all times *t*, each node *i* maintains a *sum* $s_{t,i}$, initialized to $s_{0,i} := x_i$, and a *weight* initialized to $w_{0,i} := 1$. At time 0, it sends the pair $(s_{0,i}, w_{0,i})$ to itself, and in each subsequent time step t, each node *i* follows the Push-Sum protocol given as Algorithm 1. In this case, x_i is the traffic at node *i*.

Algorithm 1 Protocol Push-Sum

1: Let $\{(\hat{s}_r, \hat{w}_r)\}$ be all pairs sent to i in round $t-1$
2: Let $s_{t,i} := \sum_r \hat{s}_r$, $w_{t,i} := \sum_r \hat{w}_r$
3: Choose a target $f_t(i)$ uniformly at random
4: Send the pair $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ to $f_t(i)$ and i (yourself)
5: $\frac{s_{t,i}}{w_{t,i}}$ is the estimate of the average in step t

For our purposes, we define a convergence when a local node's approximation of the average does not differ more than 10^{-9} for three consecutives times. It is important to note that the convergence does not equate to the true average but instead yields a bounded approximation. The deviation from the true average correlates to which node begins the algorithm, as discussed under diffusion speeds. To obtain a more accurate approximation the push-sum protocol is executed for 30 iterations (iteration defined as when there is a convergence), and the average of the iterations is used as the system average. To reset the Push-Sum protocol in a new iteration, all nodes in the network must synchronize by reinitializing $s_{0,i} := x_i$, and $w_{0,i} := 1$. To synchronize the nodes, we use a distributed barrier.

5.2. Distributed Barrier

In multicore computing, threads are assigned asynchronous tasks where each thread has its own local memory, and they communicate through shared memory. Valiant introduced a *Bulk Synchronous Parallel*(BSP) model [38] for an algorithm where:

- Each processor/thread is concurrently running a local computation.
- Threads exchange data through shared memory.
- A computation consists of *supersteps* where in each step individual threads are assigned tasks. There needs to be synchronization between the supersteps to ensure all threads have completed their step before moving to the next superstep.

The Push-Sum algorithm is distributed concurrent algorithm, where communication is through the network instead of shared memory on a single host. BSP uses a barrier for synchronization where the threads enter the barrier at the start of a computation step, and then wait for the barrier to release upon completion. See Figure 25.

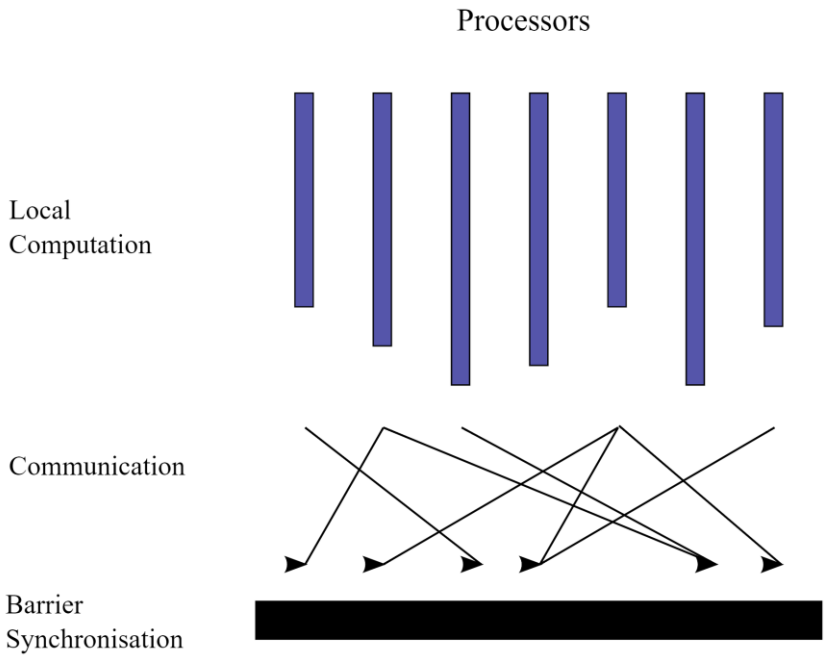

Figure 25: BSP based model.

In a distributed environment, where processors are on separate hosts, implementing a synchronization barrier becomes challenging. It has to be global in the network so that all nodes can access it. A single node can act as a centralized barrier however it will not be resilient to failures, and will also not scale with the network. Our solution to synchronization composes of multiple nodes, making it resilient and scalable. We utilize ETCD's registry to create a key to act as a barrier, since ETCD is logically centralized by physically deployed on multiple nodes. ETCD allows a watch feature to subscribe for key changes. The first node to enter the Push-Sum protocol creates the barrier key, and when the rest of the nodes join in they watch for changes to the barrier key. When an algorithm convergence occurs at a node, the node deletes the key from ETCD and the rest of the nodes are released from the current iteration of the algorithm. When a node starts the iteration of

a Push-Sum protocol, it takes a system snapshot to initialize the traffic measurement at the current node. The snapshot ensures that although the traffic count keeps changing, the value used for the algorithm is consistent.

5.3 Network Diffusion

How fast a value originating from a node diffuses the network depends highly on the topology of the network. The number adjacent edges a node holds also determines the node's participation in a protocol such as Push-Sum. Participation in Push-Sum would be for a node to receive and send the sum and weight in an iteration at least once. Depending on the topology some nodes may not participate in the protocol at all. For example, in Figure 24 the nodes with the lowest probabilities may not be aware that an iteration took place since they may not receive the initial Push-Sum message to begin the protocol. For our methodology this is acceptable since the search is for nodes with high traffic. A node receiving a high number of Push-Sum messages is likely to converge, therefore we can disregard low traffic nodes.

Another important observation is the impact that the seed node, selected to begin the Push-Sum protocol, has on the average approximation. If the seed node has its value lower or higher than the true average, then the estimation will reflect that deviation by being lower or higher than the true average. This deviation is due to the fact that the weight is halved every time the ratio is sent to another node, which means the first nodes where the protocol begins will have the most influence on the estimate. For this reason, we run 30 iterations of Push-Sum and randomize the seed node every iteration.

5.3 Determining the High-Risk edge

To determine if an adjacent edge is a high-risk edge, a node must first determine if it's a high traffic node itself. The detection is performed in a decentralized fashion where nodes are not aware of the traffic statistics of other nodes in the network. It must infer from another form of information if its own traffic is higher than most nodes. Using the Push-Sum estimations, we can establish a threshold for high traffic. Since we are using iterations for Push-Sum, we can use each estimation as a sample in a distribution that models the actual distribution of the network. We define our threshold based on the sample distribution as the following:

$$T = 2\sigma + \mu$$

Where μ is the mean and σ is the standard deviation of the samples. If a node's value is higher than *T* then it labels itself as a high traffic node. Then, it queries its adjacent neighbours to see if any of them are also labeled as high traffic. If they are high traffic, then the shared edge is a high-risk edge. Determining the value of the threshold is relative to the distribution, because setting the threshold too high will result in no matches, and setting it too low will make more than half of the nodes in the network be identified as high-traffic. Since the search is for a node that has higher traffic than most nodes, we know that it's value should be higher than μ. Using the standard deviation as the measurement of the variation in the samples, we can define a high traffic node by how much it varies from the mean. Therefore, the threshold *T* can be adjusted by changing the factor of σ.

5.4 Results

For test topologies, the Barabasi-Albert (BA) model was used to create scale free networks [39]. Scale free networks contain hub nodes which have a very high number of links attached to them, where as most of the other nodes in the network have fewer connections. More specifically, the degree distribution in a scale free network follows a power law. Random networks, in contrast, have no notion of hubs as they have a probabilistic degree distribution. Scale free networks have been revealed to be the underlying structure of a wide range of systems. In economics, for example, where countries have trade agreements with other countries, strong economic countries will be network hubs. On the internet, some webpages are hubs such as Google. The BA model is an algorithm that generates scale free networks using preferential attachment, which links a new node to another based on the number of links the potential attachment node has. Thus, nodes with higher degrees get more new nodes attached to them over the iterations. This model is ideal for our tests as it fits our random walk model which is also based on the degree distribution of the network. See Figure 26 as an example

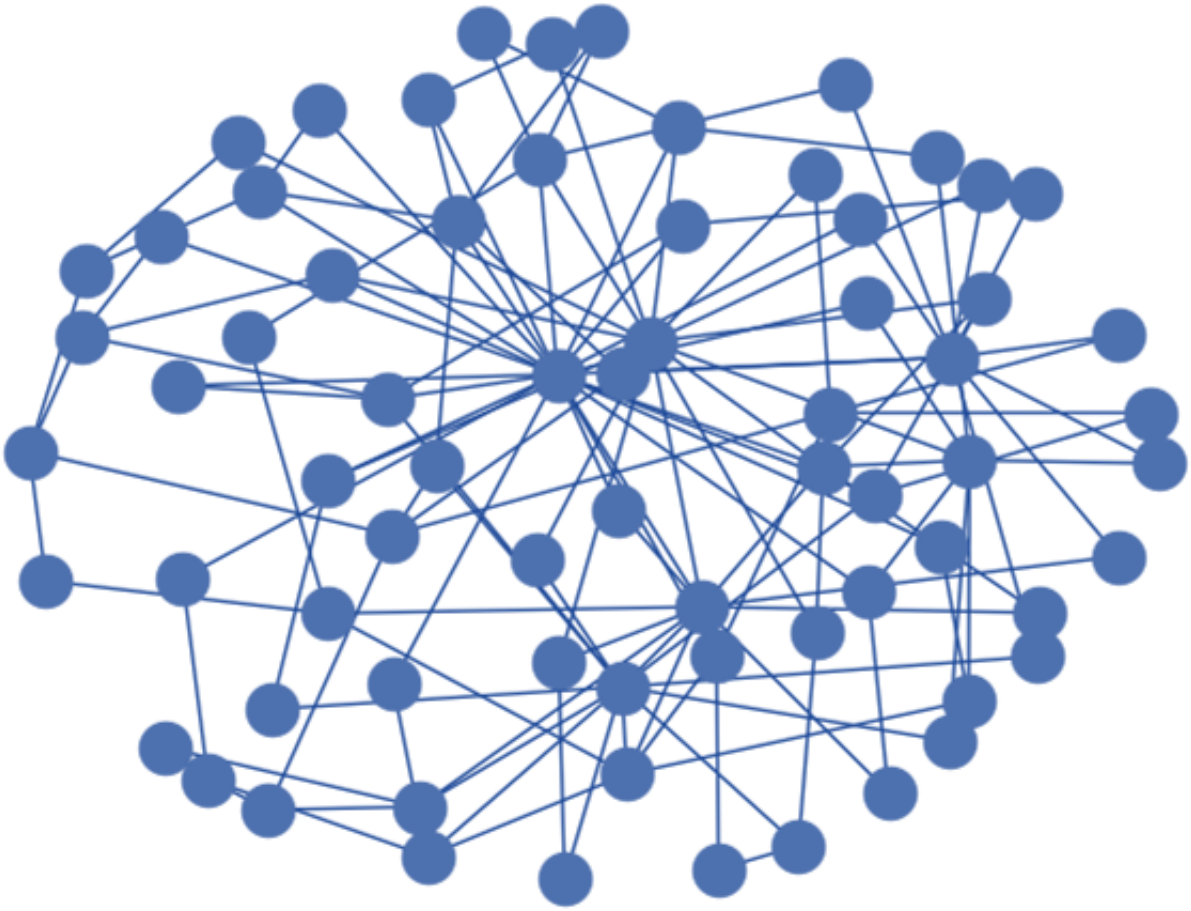

Figure 26: An example of a scale free network.

Tests are executed on a single physical machine with a quad core processor and 12GB of ram. This is not a server-spec computer, but instead an above average performance laptop so we can show the light resource footprint of the testbed. The logic is implemented in Go, and ETCD is used as a distributed barrier. Each node is a Docker container and the network topology is an overlay on the native Docker network. Apart from the efficiency of the methodology we will also see the efficiency of the proposed testbed on a single host.

Due to the simplicity of our methodology, the detection's classification is deterministic and the only variable is the threshold which determines which nodes should

be classified. To optimize the threshold value, there needs to be a model to associate the value with the size of the network. We used the same threshold across all our tests. Detection yields multiple edges that are potentially high risk. Figure 27 shows the average number of Push-Sum messages per node in one iteration. Gradual decline in the number of messages while the network size grows indicates that the Push-Sum iteration converges faster. This is because when a scale free network grows, naturally, there will be some hubs that acquire many links and many nodes with very few links. Therefore, it is much faster to determine the high traffic node since the traffic distribution will be very skewed towards those nodes.

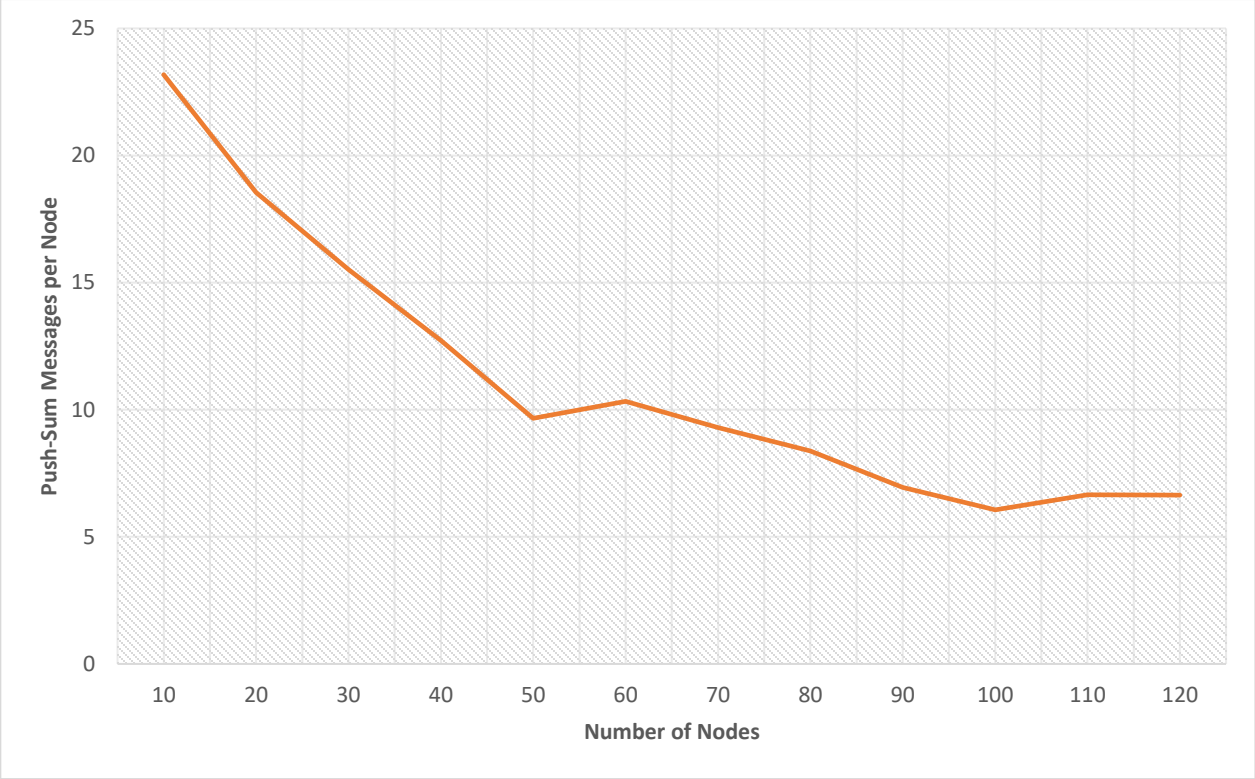

Figure 27: PS Messages as network size increases.

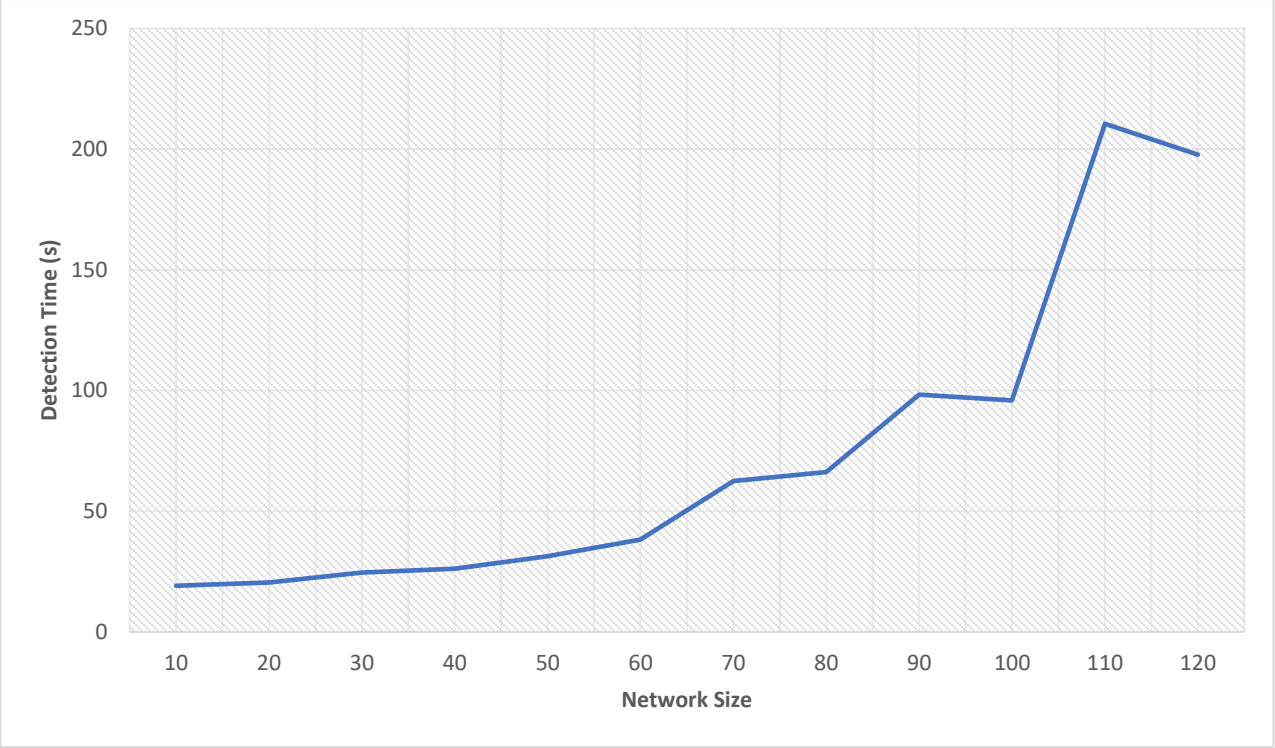

Figure 28: Detection time and network size.

Even though there is message efficiency, the detection time per network size shows the contrary in Figure 28. Running a network with over a hundred Docker nodes begins to show signs that the resource consumption is nearing capacity. Message flow within the Docker virtual network starts to slow down as hardware resources are constrained. In a real network with sufficient resources, the detection time would be lower. In terms of the testbed, we show that it is capable of handling a high number of nodes on a non-server machine. Other virtualization technologies such as VMs would not match the efficiency of simulation achieved by using Docker containers. Figure 29 shows the time it takes to start the containers. Starting 120 nodes is under 2 minutes, which is sufficient to start only one VM.

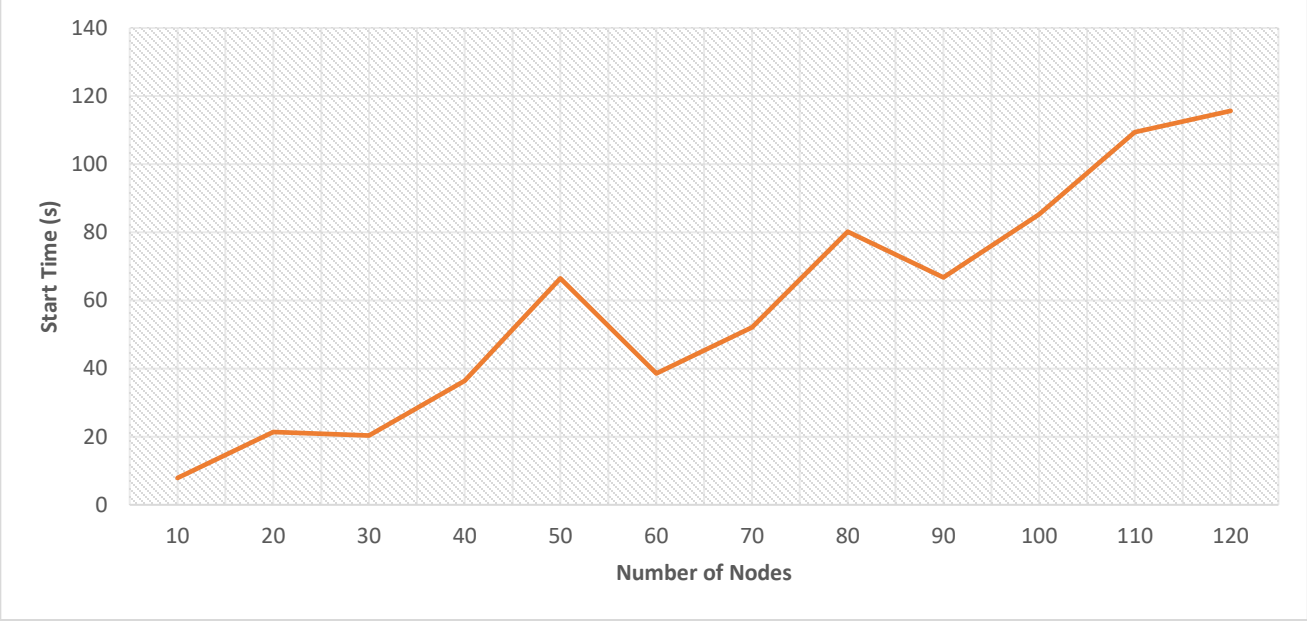

Figure 29: Node start time.

6 Conclusion and Future Work

In this thesis, we introduced a lightweight application layer distributed systems testbed. The components composing the testbed were described in detail and their usage was exemplified through three case studies. Using this testbed, developers and researchers can experiment with distributed systems through the use of resource virtualization, minimizing the hardware requirements of such experimental systems. The testbed uses production ready components such as Docker containers, Etcd, and the Go language which are actively supported with good coverage of documentation to get started.

Three separate case studies show the versatility of the testbed where it can be used for practical software development, and working with theoretical ideas. In the first case study, we use the components as they are typically used in a commercial software environment to prototype a design for a software solution. Requirements were provided by the industry partner with whom the project was co-developed with. We used a concept of SOA, microservices, to translate the requirements into individual modular services that can be independently developed and managed. Using this architecture, software development becomes streamlined in terms of dividing service ownership among developers, rapid deployment for testing, and growing the system from prototype to production.

In the second case study, we implemented the GHS algorithm for finding the MST in a decentralized setting. Through commonly implemented using threads and shared-memory, we implement it using networking constructs in Docker and Go. We describe issues that arise when developing a theoretical algorithm in a real network, which is another benefit of using the testbed.

Finally, in the third case study, we develop a methodology to find a high-risk edge in a decentralized peer-to-peer configuration. We use the concept of random walks on networks to simulate and model the message traffic so that we have a deterministic simulation. Then the Push-Sum protocol is used to aggregate network statistics using gossip based communication. An edge is high-risk if its between two high-traffic nodes. Nodes infer the existence of a high-risk edge using aggregated network statistics. In this case study, we also present the efficiency of the system testbed by experimenting with a large number of nodes. Compared to VMs, Docker containers prove to be significantly more lightweight and efficient with hardware resources. Running hundreds of nodes in a single commodity machine demonstrates this efficiency.

Future work on this testbed can be done to formalize the framework and develop interfaces for specialized distributed systems. In distributed systems courses, this testbed can be used by students on single machines to test theoretical work from the classroom, mainly due to the simplicity of Docker containers and the Go language. Another direction is to extend the third case study of network aggregation to be robust to node failures. Especially failure of hubs in scale free network, which can be disruptive to the network's operation.

References

[1] Docker, "Docker Docs," [Online]. Available: http://www.docker.com. [Accessed January 2017].

[2] B. Golden, Virtualization For Dummies, Indianapolis: Wiley Publishing Inc, 2008.

[3] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Hoboken: Wiley Publishing Inc, 2013, pp. 55-60, 105-110.

[4] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299-319, 1990.

[5] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, Distributed Systems: Concepts and Design, Boston: Addison-Wesley, 2012.

[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX* , Philadelphia, 2014.

[7] "Etcd," [Online]. Available: https://coreos.com/etcd/. [Accessed May 2017].

[8] "gRPC," [Online]. Available: http://www.grpc.io/. [Accessed 2017].

[9] "Protocol Buffers," [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed May 2017].

[10] Etcd, "etcd API Reference," [Online]. Available: https://coreos.com/etcd/docs/latest/dev-guide/api_reference_v3.html. [Accessed May 2017].

[11] "Kubernetes," Google, [Online]. Available: https://kubernetes.io/. [Accessed May 2017].

[12] "The Go Programming Language: Frequently Asked Questions," [Online]. Available: https://golang.org/doc/faq. [Accessed 05 05 2017].

[13] "The Go Programming Language Specification," 18 Nov 2016. [Online]. Available: https://golang.org/ref/spec. [Accessed 06 January 2017].

[14] "The Go Programming Language: Packages," [Online]. Available: https://golang.org/pkg/. [Accessed 05 May 2017].

[15] C. Hoare, Communicating Sequential Processes, Prentice Hall, 2015.

[16] D. Knuth, "Coroutines," in *The Art of Computer Programming: Volume 1: Fundamental Algorithms*, Boston, Addison-Wesley, 1997, p. 193.

[17] W. Schulte and Y. V. Natis, ""Service Oriented" Architectures, Part 1," *Garter Report*, 12 April 1996.

[18] K. B. Laskey and K. Laskey, "Service Oriented Architecture," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, pp. 101-105, August 2009.

[19] T. Erl, Service-Oriented Architecture, Prentice Hall, 2005.

[20] A. Fox and D. Patterson, Engineering Software as a Service: An Agile Approach Using Cloud Computing, Strawberry Canyon LLC, 2014.

[21] C. Richardson, "Monolithic Architecture," Microservices.io, [Online]. Available: http://microservices.io/patterns/monolithic.html. [Accessed 30 July 2017].

[22] M. Fowler and J. Lewis, "Microservices," 25 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 30 July 2017].

[23] J. Gray and W. Vogels, "A Conversation with Werner Vogels," *ACM Queue*, vol. 4, no. 4, pp. 14-22, 2006.

[24] C. Richardson, "Choosing a Microservices Deployment Strategy," 10 February 2016. [Online]. Available: https://www.nginx.com/blog/deploying-microservices/. [Accessed 29 July 2017].

[25] Docker, "Limit a container's resources," [Online]. Available: https://docs.docker.com/engine/admin/resource_constraints/. [Accessed 20 July 2017].

[26] M. DB, "Containers and Orchestration Explained," [Online]. Available: https://www.mongodb.com/containers-and-orchestration-explained. [Accessed 10 July 2017].

[27] "What is Mesos? A distributed systems kernel," Apache, [Online]. Available: http://mesos.apache.org/. [Accessed 20 July 2017].

[28] Kubernetes, "Production-Grade Container Orchestration," [Online]. Available: https://kubernetes.io/. [Accessed 10 July 2017].

[29] G. Labs, "Registrator," [Online]. Available: https://github.com/gliderlabs/registrator. [Accessed 2 December 2016].

[30] R. L. Graham and P. Hell, "On the History of the Minimum Spanning Tree Problem," *IEEE History of Computing,* vol. 7, no. 1, pp. 43-57, 1985.

[31] P. A. Humblet, R. G. Gallager and P. M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Transactions on Programming Languages and Systems,* vol. 5, no. 1, pp. 66-77, 1983.

[32] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal,* vol. 36, no. 6, pp. 1389-1401, 1957.

[33] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society,* vol. 7, no. 1, pp. 48-50, 1956.

[34] J. Nesetril, E. Milkova and H. Nesetrilova, "Otakar Borůvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history," *Elsevier: Discrete Mathematics,* vol. 233, no. 1-3, pp. 3-36, 2001.

[35] C.-T. Chou, "A Bug in GHS," 1988.

[36] Y. Moses and B. Shimony, "A New Proof of the GHS Minimum Spanning Tree Algorithm," *International Symposium on Distributed Computing,* pp. 120-135, 2006.

[37] J. D. Noh and H. Reiger, "Random Walks on Complex Networks," *Physical Review Letters,* vol. 92, no. 11, 2004.

[38] L. G. Valiant, "A Bridging Nodel for Parallel Computation," *Communications of the ACM,* vol. 33, no. 8, p. 103, 1990.

[39] R. Albert and A.-L. Barabasi, "Statistical Mechanics of Complex Networks," *Reviews of Modern Physics,* vol. 74, no. 1, 2002.

[40] G. o. Canada, "Environment Canada," [Online]. Available: https://weather.gc.ca/. [Accessed 17 10 2017].

[41] D. Kempe, A. Dobra and J. Gehrke, "Gossip-Based Computation of Aggregate Information," *44th Annu. IEEE Symp. Foundations of Computer Science (FOCS '03),* vol. 8, no. 3, pp. 482-491, 2003.