NAIST-IS-DD1561034

Doctoral Dissertation

Increasing Data Center Efficiency with Improved Task Scheduling and Communication

Pongsakorn U-chupala

March 5, 2018

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Pongsakorn U-chupala

Increasing Data Center Efficiency with Improved Task Scheduling and Communication*

Pongsakorn U-chupala

Abstract

A data center has become a crucial element of the modern information technology society. Operating a data center is costly so the efficiency is vital. The primary objective of this research is to maximize the efficiency, specifically the "computation efficiency", of a data center. There are two factors that have the major impact on the computation efficiency of a data center: Task Scheduling efficiency and communication efficiency. We develop and evaluate several techniques to address these two factors. These works are organized into three contributions. Firstly, we propose Container Rebalancing, a novel task scheduling mechanism which increased task scheduling efficiency for Linux Container (LXC) based data center. Container Rebalancing takes advantage of LXC's rapid container migration to increase the optimal overcommit ratio of a Linux container cluster and in turn, increases overall resources utilization of the data center. Secondly, we propose the application-aware routing, a software-defined network (SDN) assisted routing mechanism. The application-aware routing route each network flow individually according to the characteristics of the corresponding application, resulting in a better performance to each flow and an increased communication efficiency to the overall network. A network implementing application-aware routing is called an application-aware network. Thirdly, we propose a self-optimizing application-aware network, an improved version of an application-aware network. The self-optimizing application-aware network use an automatic network traffic classification model instead of manual user configuration to identify and categorize network flow, making it easier to deploy. The automatic network traffic

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, NAIST-IS-DD1561034, March 5, 2018.

classification model is a major component of the self-optimizing application-aware network. The model is learned using a stacked denoising autoencoder, a deep-learning-based classification technique.

Keywords:

Datacenter, Networking, Software-Defined Network, Optimization

Contents

List of Figures

List of Tables

1. Introduction

It is indisputable that a data center has become a crucial element of the modern information technology society. With the rise of microservices architecture, modern "cloud-native" softwares are written in a distributed fashion for cloud computing environment. The surge of IoT devices produces more data than ever. These data would require a tremendous amount of computing power to store and process. A recent breakthrough in Artificial Intelligence and Deep Learning also drive the need for a high-performance computing even further. A data center capable of providing these services is essential.

Because operating a data center is an expensive operation, efficiency is vital. It is commonly known that power consumption is one of the major contributors to the cost of data center operation. There are several approaches to raise the efficiency of a data center regarding power consumption. For example, a workload-based dynamic resource provisioning is can be utilized [10]. However, once a server is in operation, the amount of workload have little impact on the power consumption of an individual server. The utilization of the servers in a data center should be maximized for the maximum efficiency.

1.1. Data Center Optimization

Most of the existing data center optimizations focus on optimizing power consumption efficiency. Electricity is required to power all components of a data center from the server hardware and the network equipment to supporting facilities such as cooling system. It is commonly known that power consumption is a major contributor to the day-to-day operation cost of a data center. As such, power efficiency has become the primary target for data center optimization. Many works has been done to study power utilization [24, 43, 59] and optimize various components of a data center to improve power efficiency [13, 23, 42, 46, 50, 53, 67, 80]. Some example includes: an energy efficient resource management system with dynamic allocation [13], power consumption reduction of the components during idle period [50], and idle power consumption reduction with improved power state transition time [46]. Many improvements to the data center architecture are developed [23, 42, 67, 80]. In addition to a single-component optimization, a power management solution coordinating various optimization techniques for different components is also presented [53].

While power consumption is a major contributor to the day-to-day operation cost of a data center, another large portion, sometimes the biggest, of the amortized cost of the data center operation goes to the hardware costs including servers and network equipment. The amortized cost is calculated over the data center lifetime, allowing it to be a common measure for both one time purchases (e.g., equipments) and recurring spending (e.g., electricity). According to the study by A. Greenberg et al., roughly 60 percent of the amortized cost of a data center operation goes to server hardware and network equipment combined [22].

Given the scale of a data center nowadays, optimizing computation efficiency is also an effective strategy and a very compelling approach. To meet with the ever increasing demands, modern data centers are bigger than ever. The computation requirements of a data center have also grown immensely. Optimizing the computation efficiency of a data center reduces the amount of the computation resources (e.g., server hardware, network equipment) required to meet a data center computation requirements. Since the effectiveness of the computation efficiency optimization scales directly with the size of the data center, it has become a compelling strategy. More efficient computation also help reduce power consumption and improve power efficiency, since the fewer hardware is necessary. By improving the efficiency of an individual server, more servers and network equipments could be turned off or transited to a low-powered idle state when not needed. For this reason, the efficiency gained from the dynamic resource provisioning is also increased.

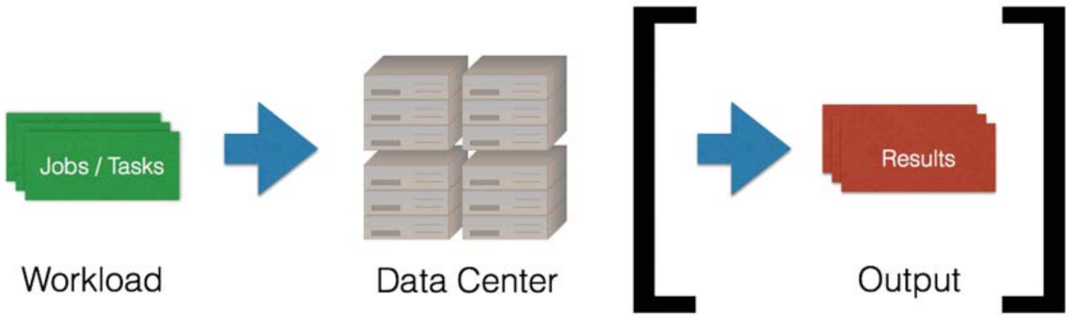


Figure 1.1.: Data center operation model

1.2. Contribution

The primary objective of this research is to maximize the efficiency, specifically the "computation efficiency", of a data center. We define the "computation efficiency" as the utilization of computation resources. Computational resources are logical resources used to execute workload. For example, CPU time, memory usage, and usage of specialized hardware such as GPU allocation are considered computation resources.

There are two factors that have the major impact on the efficiency of a data center: task scheduling efficiency and communication efficiency. Figure 1.1 illustrates a typical operation model of a data center. A data center processes the workload, a set of jobs to be executed, and may produce resulting outputs. A job consists of a single or multiple tasks to be executed together. A task is a fundamental execution unit of a data center. Generally, a majority of jobs contains multiple tasks communicating with one another during execution. Figure 1.2 illustrates the relationship between the workload, jobs, and tasks. There are two congestion points in the data center operation model: task distribution and execution. The two factors of major impact, task scheduling efficiency, and communication efficiency are the factors corresponding to these two congestion points. The relationship between the two factors of major impact and the two congestion points are elaborated in Subsection 1.2.1 and Subsection 1.2.2 respectively.

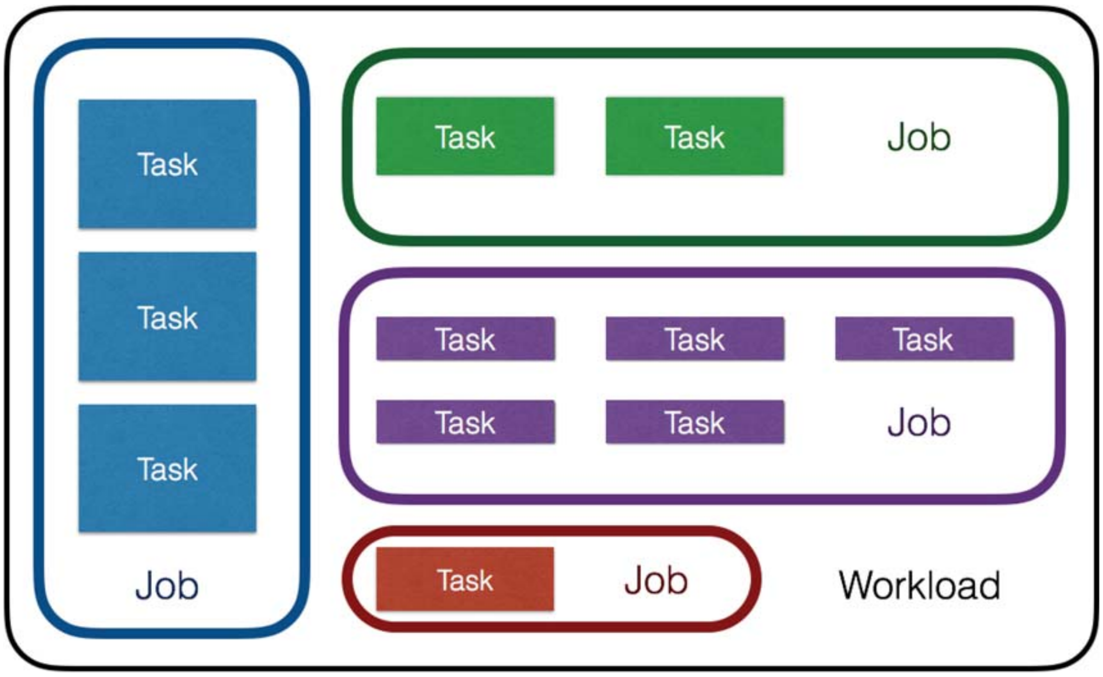


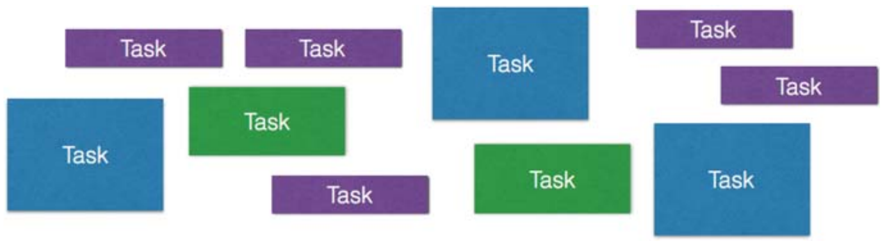
Figure 1.2.: An example workload illustrating the relationship between workload, jobs and tasks

We propose and evaluate several techniques to address the two factors of major impact on the computation efficiency of a data center: task scheduling efficiency and communication efficiency. These techniques are presented in a form of three contributions towards addressing the two factors of major impact.

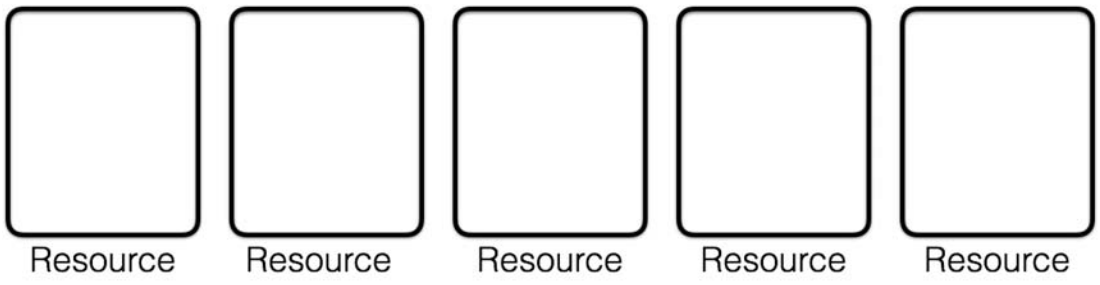
1.2.1. Task Scheduling Efficiency

An efficient task scheduling is essential for optimal task distribution. Since task distribution is a congestion point in the data center operation model, optimizing task distribution is vital to maximizing the utilization of the computation resources and the data center. Optimal task distribution minimizes the number of required resource nodes (servers) given the workload and maximizes the utilization of each resource node. Figure 1.3 shows an example of an optimal task distribution. The more efficient the task scheduling is, the higher the throughput of the data center becomes.

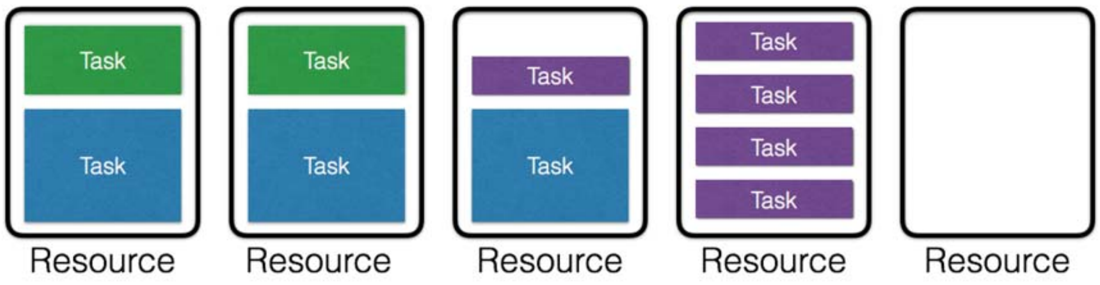
For the first contribution, we propose container rebalancing mechanism. The



(a) An example workload to be distributed (scheduled)



(b) Resource nodes available for workload distribution (scheduling)



(c) An optimal distribution of the workload given the resource nodes

Figure 1.3.: An example of an optimal workload distribution

container rebalancing method is a novel task scheduling mechanism designed specifically to take advantage of the unique property of Linux container in order to increase computation resource utilization. The container rebalancing method is described in detail in Chapter 2.

1.2.2. Communication Efficiency

More efficient communication raise the computation efficiency of a general-purpose data center. Execution is another congestion point for the data center operation model. Commonly, most jobs contain multiple tasks working in tandem. Communication between tasks is often the bottleneck of the execution. The efficiency of the communication is a dominant contributor to the execution and the computation efficiency. Improving the communication efficiency will have a direct impact on the execution and the computation efficiency of a data center. Note that communication efficiency may not be the dominant factor in some specifically-designed data center. For example, an Apache Hadoop cluster is highly optimized for map-reduce operations and could handle map-reduce-related communications very efficiently. This kind of cluster is not the focus of this work.

For the second contribution, we developed the application-aware routing and the application-aware network. The application-aware routing is a software-defined network (SDN) assisted routing method which routes each flow individually according to the characteristics of the corresponding application. A network with application-aware routing is called an application-aware network. The application-aware routing and the application-aware network are described in detail in Chapter3.

For the third contribution, we propose the concept of a self-optimizing application-aware network. We also present a deep-learning-based-network traffic classification technique which is a major development towards a self-optimizing application-aware network. The detail regarding the self-optimizing application-aware network as well as the deep-learning-based network traffic classification is described in Chapter 4.

1.3. Organization of the Thesis

The rest of the thesis is organized as followed. Chapter 2 explains container rebalancing method. Chapter 3 explains the application-aware network. Chapter 4 describes a major development toward a self-optimizing application-aware network including a deep-learning-based network traffic classification. Chapter 5 provides the conclusion and discusses future work.

2. Container Rebalancing

Virtualization technology [12, 39] allows resources on a single computer to be sliced into multiple isolated units. This technology is considered mature and is commonly used in data centers to enable more granular resource allocation resulting in a more flexible resource management, as well as a higher resource utilization. With Virtualization, resource isolation is achieved by virtualizing an entire computer with an allocated amount of resources. Each virtualized unit is called a "virtual machine" (VM). This method, however, takes inherent performance overhead [55].

Linux Containers (LXC) [5] is another technology for resource isolation. Instead of virtualizing an entire machine, thus requiring a separate "guest" operating system, LXC provides each "container" with its own separate Linux environment while still sharing the same underlying Linux kernel. Compared to VM, an LXC container takes significantly lower performance overhead [21,74]. LXC containers are also instantiate significantly faster and require less storage space compared to VM (as elaborate in Section 2.1). These advantages make many LXC-based tools such as Docker [47] and CoreOS [3] popular alternatives to full virtualization-based solutions [14]. Figure 2.1 illustrates the difference between Virtualization and LXC.

Rapid container migration is a powerful technique made possible with LXC. An LXC container (referred to here as "container" from this point onward) may take *seconds* to boot up whereas a similar VM may take *minutes*. A container also takes less disk space than a VM, since a container does not have to contain the whole operating system. These two traits combined make container migration much faster than migrating the VM and offer a feasible resource management technique.

None of the existing container orchestration solutions take advantage of rapid

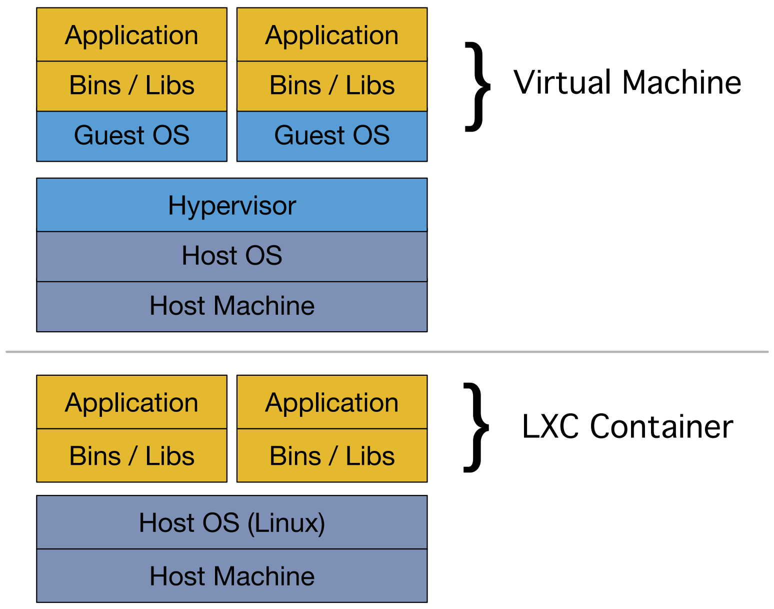


Figure 2.1.: Abstraction comparison between Virtualization and Linux Containers

container migration. With the rising popularity of LXC, many tools and technologies are being developed [4, 6, 26, 58]. However, they concentrate on resource management and do not try to take advantage of the unique capabilities of LXC such as rapid container migration.

We explores the possibility of leveraging rapid container migration as a resource management technique in conjunction with existing optimization techniques so as to increase data center efficiency. In the process, we proposes "container rebalancing", a novel mechanism with a rebalancing process constantly optimizes LXC container placement by using rapid container migration.

2.1. Background

This paper proposes container rebalancing, a novel method to increase LXC cluster efficiency by increasing the optimal overcommit ratio using rapid container migration. The approach is explained in detail in Section 2.2. This section gives a brief introduction to Linux Containers technology and the rapid container migration technique. This section also discusses existing LXC resource management techniques including overcommitting.

2.1.1. Linux Containers

Linux Containers (LXC) is a technology that combines several Linux kernel features to create a contained process called "container". Each container has its isolated view of the operating system environment with only an allocated amount of resources, virtually achieving an isolated Linux environment without the use of Virtualization technology. LXC's containers incur significantly lower performance overhead than Virtualization since LXC's containers work on an operating-system-level [21, 74]. LXC may be viewed, in a sense, as a lightweight Virtualization, though there are several differences.

LXC technology offers several advantages over Virtualization. Since creating a new container is essentially the same as spawning a new process, a container could be instantiated almost immediately. A container also typically requires less disk space, since it does not contain the whole operating system. Docker reduces the disk space requirement further by using AuFS to enable a layered file system, allowing images to be stacked on top of each other. This results in each image containing only the data which differs from base images [2].

2.1.2. Rapid Container Migration

Migrating an LXC container could be considered as a trivial operation. Total time for migrating an isolated computing unit (t_{mi}), whether it is a container or a VM, could be broken down into 3 parts: disk copying time (t_{disk}), memory copying time (t_{mem}), and instantiation time (t_{inst}). The following equation describes the model of total migration time:

$$t_{mi} = t_{disk} + t_{mem} + t_{inst}. \tag{2.1}$$

Container migration time is greatly reduced compared to migrating a VM since speedy instantiation (small t_{inst}) and a relatively smaller disk space requirement (leading to small t_{disk}) are traits of LXC. This reduction of migration time is an important feature unique to LXC.

High VM migration time is a problem for migration-based scheduling strategy [27]. With significant reduction in migration time of LXC, rapid container migration becomes a viable optimization strategy.

2.1.3. LXC Scheduling and Overcommitting

Existing scheduling solutions for LXC clusters are typically designed as a general-purpose scheduling platform [26, 58]. While some solutions are designed specifically for managing LXC clusters by providing convenient workflow for containerized application deployment [4,6], they are not taking advantage of LXC's unique capability. Even with a fairly efficient scheduling algorithm, actual resources utilization could still be at about 50–60%, while available resources (such as CPU cores and memory) are mostly allocated [56].

One common technique for increasing scheduling efficiency is overcommitting resources [1, 7]. Overcommitting is done by allowing the scheduler to allocate more resources than the actual capacity of the system on the assumption that allocated resources are typically higher than actual utilization. This method increases scheduling efficiency because it is difficult to predict accurately the required amount of resources leading to the common occurrence of an over-allocated resources request [56,58]. Overcommitting is commonly done statically by setting a static overcommit ratio for each type of resource.

Overcommiting creates a trade off between overall cluster utilization and the execution performance of an individual container. High overcommit ratio increases overall cluster utilization but also increases the chance of container failure (as there are not enough actual resources for the operations), increases the risk of resource congestion, and causes instability [16]. More busy servers may also negatively impact the performance of an individual container. Without overcommitting or by setting the overcommit ratio too low (for a given system), resources are underutilized, which means inefficient scheduling and a waste of resources. An optimal overcommit ratio maximizes overall cluster utilization while still maintains an acceptable quality of service.

2.2. Container Rebalancing

Container rebalancing takes advantage of rapid container migration to increase the efficiency of LXC cluster scheduling. This section explains the design goals and mechanism of container rebalancing.

2.2.1. Design Goals

Three goals are considered in designing a container rebalancing method: proactive-optimization, compatibility, and scalability.

2.2.1.1. Proactive-Optimization

Container rebalancing anticipates future workloads and *proactively* optimizes container placement accordingly. Common scheduling methods usually optimize container placement *reactively* in response to changes in available resources. Proactive optimization enables the system to prepare quickly for future workloads and it works especially well in a busy cluster. This approach requires rapid migration which is a costly operation with Virtualization technology. With LXC, rapid container migration is a relatively cheap operation, thus making proactive optimization viable.

2.2.1.2. Compatibility

Container rebalancing should work alongside the existing scheduling process. Task scheduling is a rigorously researched area, and existing algorithms are fairly efficient. Instead of designing a better scheduling algorithm to replace an existing one, container rebalancing should be another process that works in conjunction with the scheduling process while minimizing interference to the scheduler.

2.2.1.3. Scalability

Container rebalancing should be able to handle a large number of containers efficiently. Since a container usually requires fewer resources than a VM with a similar configuration, an LXC cluster is expected to be able to deal with a higher number of containers. Google cluster data is a 1-month trace from one of Google's LXC clusters [57, 72]. This single month of data already contains 24,281,242 distinct tasks (containers). Given the expected size of the LXC cluster, the container rebalancing technique should be able to scale up gracefully.

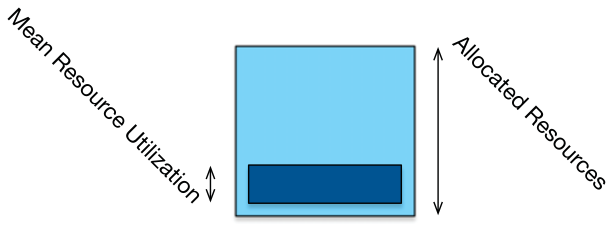
2.2.2. Implementation

The container rebalancing method consists of the rebalancing process that works alongside the scheduling process to load-balance resources between hosts of the cluster in real-time. This dynamic load-balancing process increases the probability of overcommit success as it proactively prepares rooms for overcommitting resulting in a higher optimal overcommit ratio. Figure 2.2 provides an example case illustrating the benefit of the container rebalancing method. Before the rebalancing process (Figure 2.2b), only host B has enough resources to accommodate overcommitted containers. However, both hosts have an equal chance of being overcommitted as both of them are equally allocated, making host A prone to overcommit failure. After the rebalancing process (Figure 2.2c), both hosts can accommodate overcommitted containers, thus increasing the chance of overcommit success.

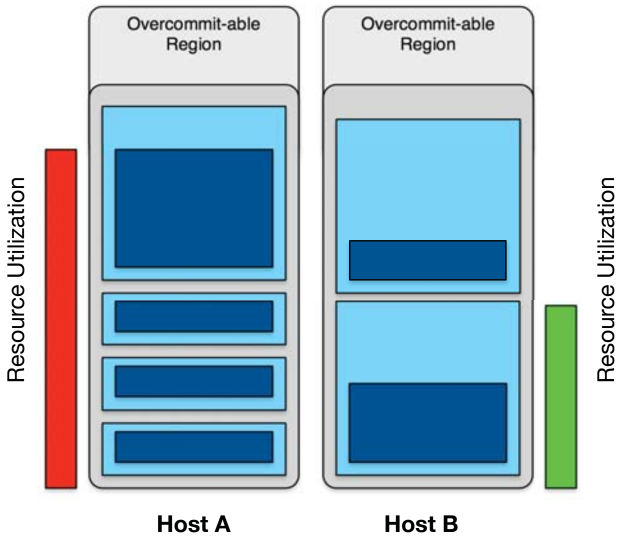
To minimize the effect on the scheduling process, only actual resource utilization is load-balanced while resources allocation is maintained. Actual resources utilization of a container is often significantly less than the allocated amount of the container, as it is difficult to predict accurately and subject to human errors [56]. This trait of container request makes it possible to load-balance actual resources utilization while still maintaining the same allocation amount on each host dictated by the scheduler.

For scalability, the container rebalancing mechanism only load-balances long-lived containers. Container requests can be classified into two groups: long-lived (service) containers and short-lived (batch) containers. Although most containers are short-lived containers, most resources of the system are consumed by long-lived containers [58]. By considering only long-lived containers, the container rebalancing process maintains significant impact on the system while drastically reducing the number of containers the process must keep track of.

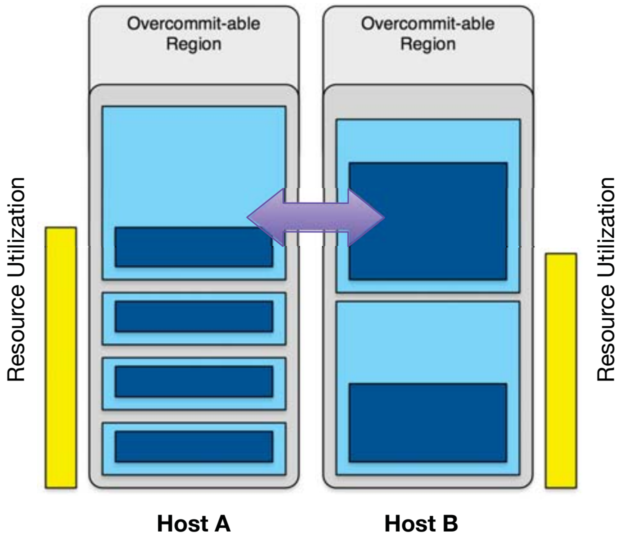
Container rebalancing mechanism may cause minor negative effects to the execution of an individual task. For example, task migration may increase task



(a) Structure of a container in terms of resource



(b) Example cluster before rebalancing process



(c) Example cluster after rebalancing process

Figure 2.2.: Example case illustrating the benefit of the container rebalancing method

Table 2.1.: An example job in the workload

Job ID	Container Index	Start Time (μs)	Duration (μs)	CPU Request (cores, normalized)	Mean CPU Usage (cores, normalized)
6252082279	0	6736765567	98000000	$6.76000e^{-3}$	$6.17230e^{-3}$
6252082279	1	6736765567	98000000	$6.76000e^{-3}$	$4.22395e^{-3}$

execution time. A task executed in a busy server may not performed as well as in a server with lower load. Several techniques can be employed to address these problems. The evaluation results also suggest that the effects are negligible. More details regarding the negative effects of container rebalancing and methods to address these problems are discussed in Subsection 2.3.4.

The container rebalancing process is divided into four repeating steps:

2.2.2.1. Container Classification

Containers are classified into long-lived containers or short-lived containers as they are inserted into the system. The classification is done using container runtime duration in each container request.

2.2.2.2. Building Comparable Container Space

Long-lived containers are grouped together according to the amounts of their allocated resources and according to their assigned hosts, creating comparable container space.

2.2.2.3. Searching Comparable Container Space

A pair of hosts with a significant resource utilization difference is selected. Comparable container space is then searched for the pair of containers from this host pair with the highest container actual utilization difference. This pair of containers is called a “swappable container pair”.

2.2.2.4. Container Swapping

Each container in the swappable container pair is migrated to the host of its counterpart.

By repeating this process, overall utilization of the cluster is load-balanced while still retaining the original allocation assigned by the scheduler on each host. Better load-balanced cluster leads to a higher overcommit success rate, a higher optimal overcommit ratio, and a higher cluster utilization.

2.3. Evaluation

An LXC cluster simulation is used to evaluate the performance and validate the feasibility of the container rebalancing mechanism. The simulation compares performance in terms of container scheduled rate and cluster utilization of a general scheduling mechanism (using only a scheduling process) to the container rebalancing mechanism (using both the scheduling process and the rebalancing process). The simulation is driven by a real-world workload from Google's cluster data. This cluster trace data is collected from Google's LXC cluster covering the period of one month and is publicly available [57, 72].

2.3.1. Workload Data Preprocessing

The workload for the simulation is extracted from Google's cluster data and organized into "jobs" and "containers". A "job" contains one "container", or multiple identical "containers" that have to be considered together while scheduling. Table 2.1 shows the organization of the workload and an example job. Google's cluster data already organizes the information into "jobs" and "tasks". A "task" is an equivalent to a "container" in the workload organization. Google's cluster data contains 672,074 jobs and 24,281,242 tasks from a 1-month period. Containers that are started before the trace period or still running after the trace period are excluded, since it is impossible to know the exact duration of these containers. 869,283 containers (3.58% of the total containers) are excluded in this process.

CPU cores are the only resources taken into account in the simulation. This is to simplify the simulation process and is required, due to time constrains, to speed up the simulation time. More investigation will be done to quantify the effect of this simplification on the accuracy of the result. The value provided by the trace data is normalized with the number of cores in the machine with the most cores in Google's cluster for obfuscation [57, 72].

2.3.2. Simulation

The simulation is an event-driven simulation with four processes running simultaneously: producer, scheduler, rebalancer, and monitor. The general scheduling mechanism is simulated using producer, scheduler, and monitor processes. The rebalancing mechanism is simulated using all four processes. Each machine in the simulated cluster has the same number of CPU cores which is equal to the number of CPU cores in the machine with the most cores in Google's cluster data. 2,000 machines in the LXC cluster are simulated in this research so that the total number of cores in the cluster is roughly equal to total number cores in Google's cluster. Although Google's cluster data contains the trace of a full month, to speed up the process, only the first week of the records was used in the simulation. SimPy 3.0.8 [9] was chosen for the implementation due to the authors' familiarity with Python.

2.3.2.1. Producer

The producer process inserts a "job" into the "job_queue" when the simulation time reaches the starting time of each "job". Algorithm 1 describes the producer process. Using the recorded duration in the trace data, this process also categorizes each "container" in a "job" as a long-lived container or a short-lived container. Containers with a duration shorter than 2,000 seconds are categorized as a short-lived containers, while the rest are categorized as a long-lived. 2,000 seconds was chosen as the threshold given that the 80th percentile runtime of a short-lived container is 12–20 minutes and that of a long-lived container is 29 days, according to the work by Schwarzkopf et al. [58].

Algorithm 1 Producer Process

```
loop
    Get $job$ from preprocessed trace data (ordered by $start\_time$)
    if $job : start\_time$ exceeds current simulation time then
        Step current simulation time to $start\_time$
    end if
    Categorize and mark $containers$ in the $job$ as long-lived container or short-lived container
    Insert $job$ to $job\_queue$
end loop
```

2.3.2.2. Scheduler

The scheduler process implements a common scheduling strategy with overcommitting. Algorithm 2 describes the scheduler process. The scheduler handles container requests in a “job” batch. The scheduler uses a random-first-fit algorithm to find an open slot for each “job” for speed and simplicity. If all “containers” in a “job” could not be scheduled simultaneously, the “job” is skipped and queued for retrying during the next scheduling iteration. For this simulation, the *rebalancing interval* is set to 3 seconds and *max retries* is set to 300 times allowing a job to stay in the queue for about 15 minutes. If the “job” could not be scheduled within this time, it is skipped. Scheduled “jobs” and “containers” may fail if they are allocated to an overcommitted host which lacks sufficient actual capacity to facilitate them.

2.3.2.3. Rebalancer

The rabalancer process searches through scheduled containers for swappable container-pairs as described in Subsection 2.2.2, and migrates each container to its counterpart’s host. Algorithm 3 describes the rebalancer process. The search is performed in the host’s utilization-difference order and the container’s mean-cpu-usage-difference order respectively. A “container” with a mean CPU usage difference lower than 0.3 is considered comparable. For this simulation, the *rebalancing interval* is set to 3 seconds, and the *cpu utilization difference threshold*

Algorithm 2 Scheduler Process

```
 1: loop Every $scheduling\_interval$
 2:     Get $job$ from $job\_queue$
 3:     if $job : retries > max\_retries$ then
 4:         Mark $job$ and corresponding $containers$ as $skipped$
 5:     else
 6:         Try scheduling all $containers$ in the $job$ into the cluster using random first-fit algorithm
 7:         if $job$ could not fit into the cluster then
 8:             $job : retries = job : retries + 1$
 9:             Put $job$ back to $job\_queue$
10:         else
11:             if Scheduled $job$ cause over-utilization in the cluster then
12:                 Mark $job$ and corresponding $containers$ as $failed$
13:             else
14:                 Mark $job$ and corresponding $containers$ as $scheduled$
15:                 for all $container$ in $job : containers$ do
16:                     if $container$ is long-lived container then
17:                         Add $(container, host)$ to $scheduled\_long\_lived\_containers\_list$
18:                     end if
19:                 end for
20:             end if
21:         end if
22:     end if
23: end loop
```

Algorithm 3 Rebalancer Process

```
1: loop Every $rebalancing\_interval$
2:     $max\_host$ = host with maximum CPU utilization
3:     $min\_host$ = host with minimum CPU utilization
4:     if $CPU(max\_host) - CPU(min\_host) \geq cpu\_utilization\_difference\_threshold$ then
5:         $container\_pairs = LongLivedContainers(min\_host) \times LongLivedContainers(max\_hosts)$
6:         Create empty $candidate\_pair$
7:         for all $pair$ in $container\_pairs$ do
8:             if $pair$ can be swap without exceeding available resources on both host then
9:                 if ($candidate\_pair$ is empty) $\vee$
10: $(CPUUtilizationDifference(pair) > CPUUtilizationDifference(candidate\_pair))$ then
11:                     $candidate\_pair = pair$
12:                 end if
13:             end if
14:         end for
15:         if $candidate\_pair$ is not empty then
16:             Swap $container$ of the $candidate\_pair$
17:         end if
18:     end if
19: end loop
```

is set to 0.1 to keep the rebalancing process from being too aggressive.

2.3.2.4. Monitor

The monitor process keeps track of resource utilization of hosts and containers in the cluster and also generates reports. After every *monitoring interval* of 60 seconds, this process logs utilization of each host in the simulated cluster and computes evaluation metrics, including *container scheduled rate* and its variances, as described in Subsection 2.3.3.

2.3.3. Results

Three metrics are used as the indicators of the performance of the simulated cluster: *container scheduled rate (CSR)*, *long-lived container scheduled rate (LCSR)*, and *short-lived container scheduled rate (SCSR)*. These metrics are defined as follows:

$$CSR = \frac{Scheduled\ Containers}{Total\ Containers}, \tag{2.2}$$

$$LCSR = \frac{Scheduled\ Long\text{-}lived\ Containers}{Total\ Long\text{-}lived\ Container}, \tag{2.3}$$

$$SCSR = \frac{Scheduled\ Short\text{-}lived\ Containers}{Total\ Short\text{-}lived\ Container}. \tag{2.4}$$

Figure 2.3 shows CSR, LCSR, and SCSR from the simulations with various overcommit ratios. The resulting CSR suggests that the optimal overcommit ratio of the simulated cluster with the general scheduling mechanism is about 1.3. The optimal overcommit ratio increased to about 1.4 with the container rebalancing mechanism. From the results, LCSRs also conform with CSRs, and there is no remarkable improvement with SCSRs. These two facts suggest that, among the factors contributing to the increase in utilization, rebalancing only the long-lived containers is significant.

There are notable CSR, SCSR, and LCSR drops at an overcommit ratio of 1.6. They are considered to be outliers and disappear when the random seed of the simulation is changed.

Figure 2.4 shows the average utilizations of the simulated clusters using the general scheduling mechanism and the container rebalancing mechanism. Figure 2.5 shows changes in utilizations of the simulated clusters over the course of the simulations. Only the overcommit ratios of 1.3 and 1.4 are shown in the figures since they are optimal overcommit ratios for general scheduling and container rebalancing, respectively. The results suggest that, at any given simulation time, the container rebalancing mechanism typically produces a higher cluster utilization compared with the general scheduling. Container rebalancing also outperforms general scheduling even at their respective optimal overcommit ratios (as indicated by comparing (a) the container rebalancing at overcommit ratio 1.4 to (b) the general scheduling at overcommit ratio 1.3).

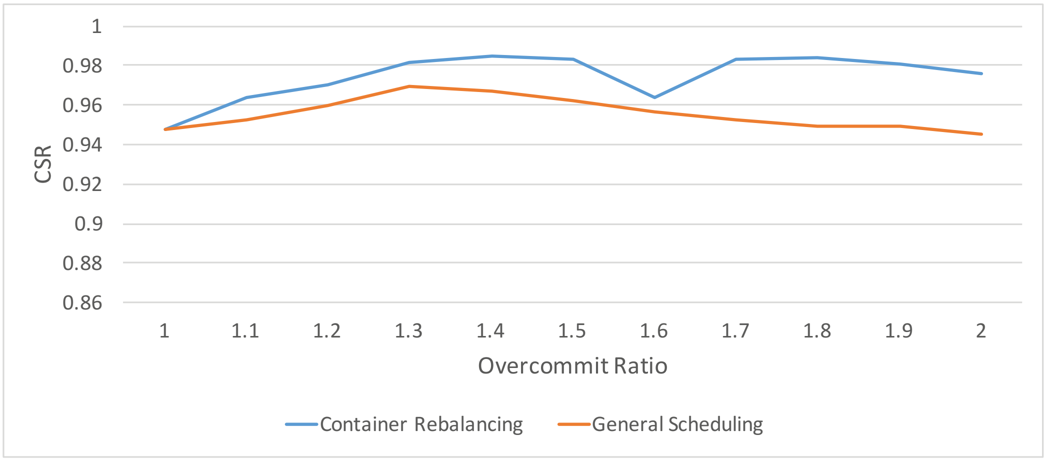
2.3.4. Discussion

From the results (Figure 2.3), container rebalancing increases the optimal overcommit ratio of an LXC cluster by a small margin. This increase still creates observable utilization improvement in Figure 2.4 and Figure 2.5. The improvement is desirable as there is no drawback to using cluster rebalancing.

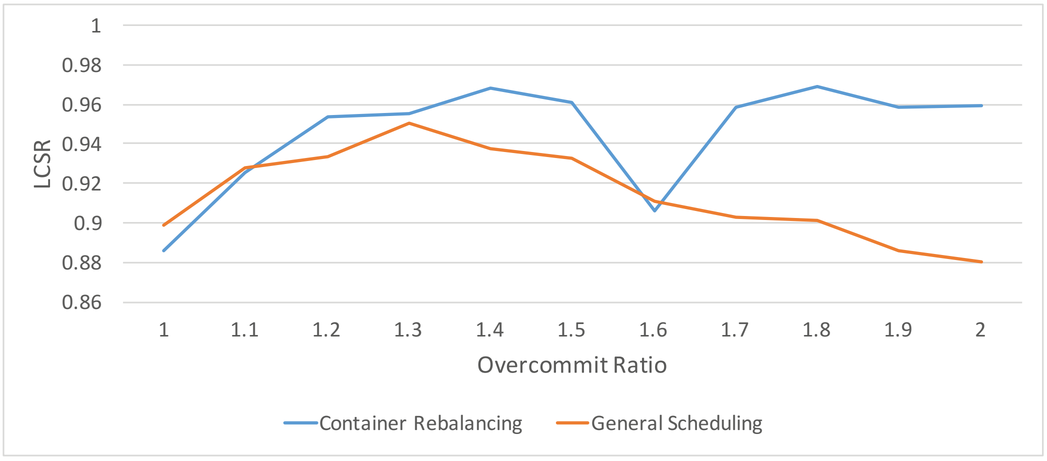
Container rebalancing also improves cluster performance (measured by container scheduled rate and cluster utilization in this research) regardless of the overcommit ratio. We believe that this is because a load-balanced cluster enables more containers to be successfully overcommitted, allowing more works to be done with the same amount of resources. Specifically, in this simulation, at overcommit ratio 1.4, 1.8% more containers are executed (an increase from 4,958,643 containers to 5,050,295 containers).

The overcommit ratio should be determined specifically for each cluster. Although the results suggest that the container rebalancing mechanism increases the overcommit ratio, multiple factors are influencing the optimal overcommit ratio of a particular cluster. For example, the number of hosts, the capacity of each host, and the scheduling algorithm can drastically change the value. Cluster simulation could be used to calculate the optimal overcommit ratio by using a cluster configuration similar to the targeted cluster and the real captured workload.

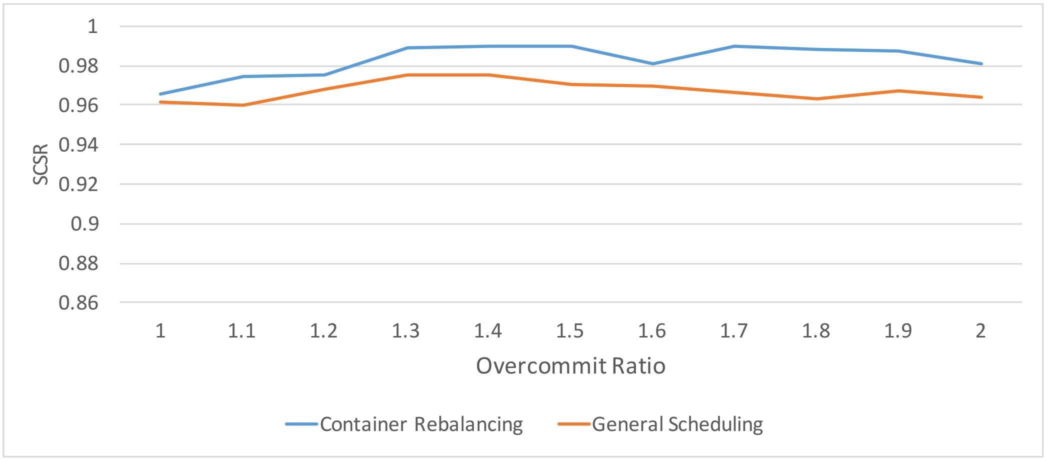
Rapid container migration adds little or no overhead to the execution. At overcommit ratio 1.4, 67,718 unique containers are migrated. This number is



(a) Container Scheduled Rate



(b) Long-lived Container Scheduled Rate



(c) Short-lived Container Scheduled Rate

Figure 2.3.: CSR, LCSR and SCSR from the simulations

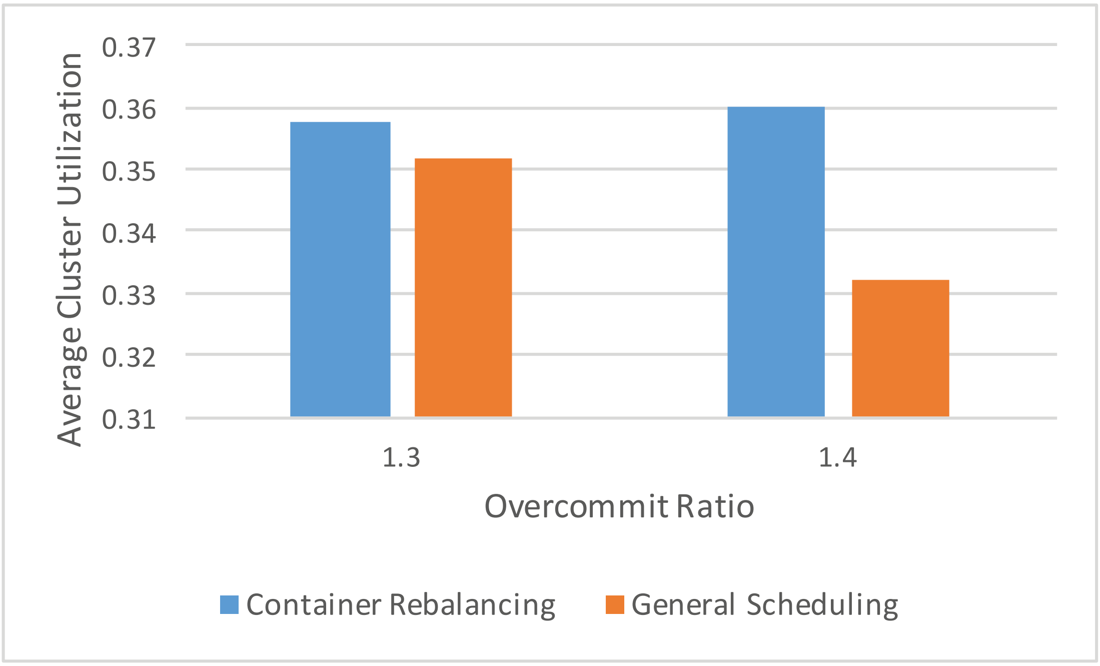


Figure 2.4.: Average cluster utilizations from the simulations

1.32% of all containers in the simulation (5,127,542 containers) and 5.96% of all long-lived containers in the simulation (1,136,517 containers). Figure 2.6 shows the distribution of unique containers by their migration count at overcommit ratio 1.4. Around half of the affected containers are migrated only once. Almost all of the affected containers are migrated less than 20 times. Migrating a container is also considered a trivial operation (as is discussed in Section 2.1). The other overcommit ratios display similar results; only the result at overcommit ratio 1.4 is shown, since it is the optimal overcommit ratio of this simulation.

Service degradation or disruption due to rapid container migration are considered negligible. Container migration may causes service degradation or disruption as a task execution may be interrupted during migration. Since the increase to total migration count from rapid container migration is minimal, the chance of a container being effected by rapid container migration is also minimal. For the containers are affected, an increase to the migration count of an individual container is also minimal, suggesting that the effect to an individual container is negligible.

There are also multiple techniques to mitigate the impact to the performance of an individual task from rapid container migration. Live migration could be utilized to reduce service disruption while the container is being migrated. For critical tasks, task prioritization could be used to prevent critical containers from being overcommited or scheduled to a high-load server.

While it is possible to over-optimize for utilization, there is also a mitigation. As the utilization approach the theoretical maximum value, execution performance may drop sharply. Taking the trade off between utilization and execution performance into account, it is generally not preferable to target the maximum theoretical utilization. Instead, targeting a slightly lower utilization allows the system to maintain effective execution performance. Usage spike may also cause the utilization to approach theoretical maximum and negatively impact the execution performance. This problem could be compensated by adjusting target utilization threshold, together with properly calculated overcommit ratio.

Due to time constraints, various compromises are taken to keep each simulation runtime to less than 24 hours. Although Google's cluster data contains one month of trace data, the simulation only simulates a single week, using the data of the first week in the trace. The results from Figure 2.5 are approaching stable values, which suggests that simulating only the first week is enough to get meaningful results. The results may be considered preliminary, because the simulation only takes into account a single resource, the CPU cores. More work has to be done to validate if cluster rebalancing is also feasible for multi-objective optimization. As a start, fuzzy-sets and weight-averaging fuzzy operators could be used to approach multi-objective optimization. This is similar to the approach in the work of Xu et al. [75].

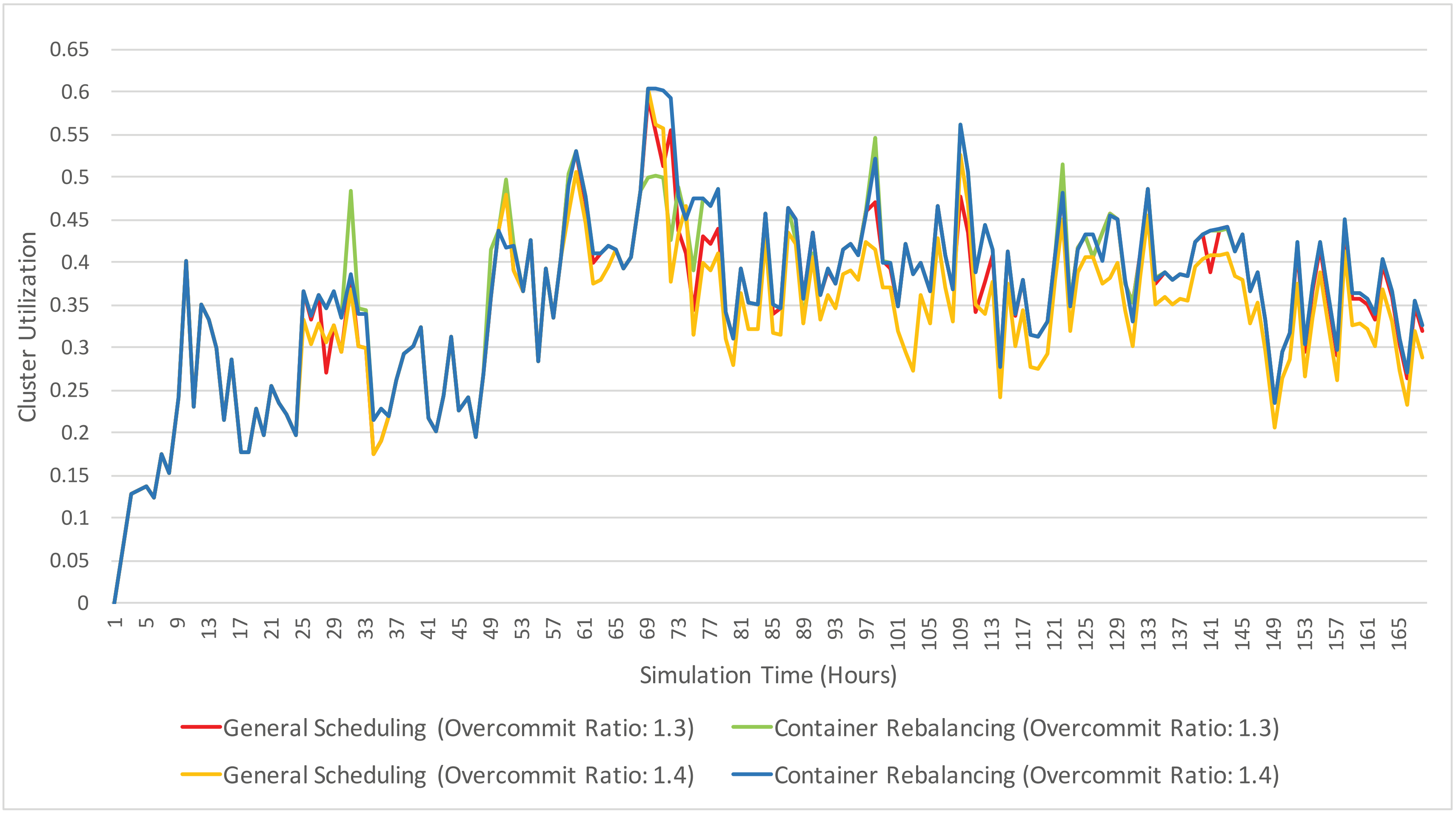


Figure 2.5.: Cluster utilizations from the simulations

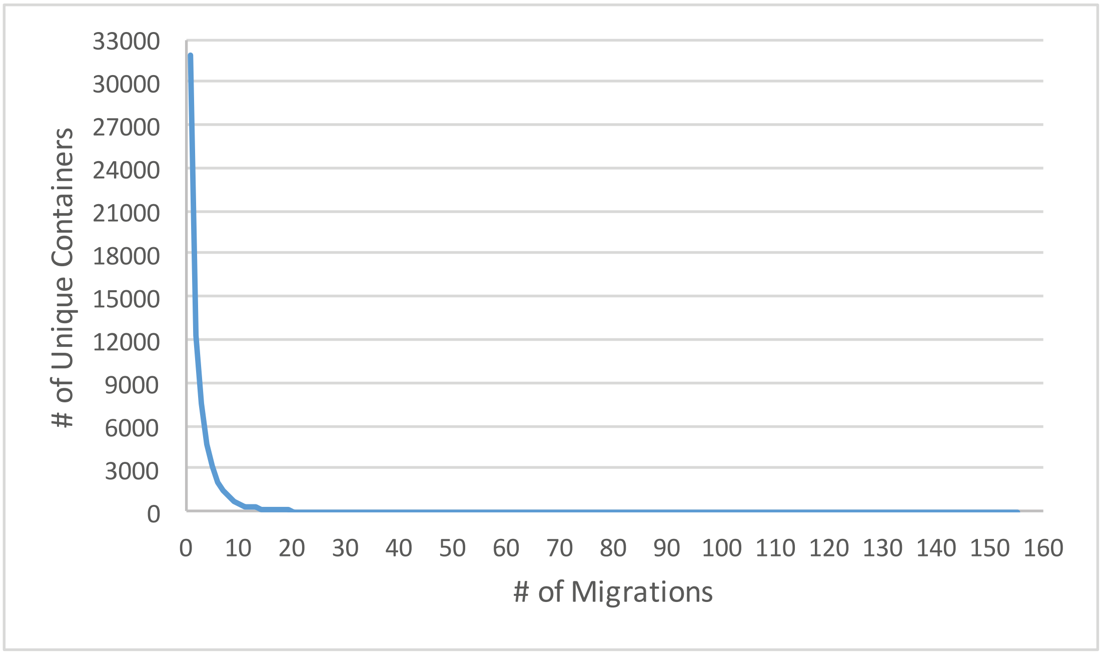


Figure 2.6.: Distribution of unique containers by their migration count throughout the container rebalancing simulation at overcommit ratio 1.4

3. Application-Aware Routing and Application-Aware Network

The Software-Defined Network (SDN) is a widely adopt technology for optimizing the network efficiency of a data center. SDN introduces the concept of a software-programmable network. This is a powerful concept that cuts across the traditional OSI layer separation [78]. A programmable network also enables tighter integration between network hardware and software. SDN also allows more granular network flow management. As a result, SDN-assisted routing has become a prominent technique for improving network efficiency.

However, there are still issues with the current approaches in SDN-assisted routing. First, conventional approaches in SDN-assisted routing search for optimal routes for a small (usually single) targeted class of applications, or a certain situation. In spite of the generality of the concepts, many of the researched approaches are only applicable to the targeted applications or situations [28, 49, 71]. Second, some approaches also require specific setups including client-side modifications which makes them difficult to adopt [49].

In this research, we propose using SDN-assisted application-aware routing to address these two issues. Application-awareness allows different optimal routes for different classes of applications, thus expanding the range of applications an optimized network can support. SDN makes possible this granularity of flow management. In realizing application-aware routing, application classification, determining the network flows of an application, is an important process. This process is done entirely within an application-aware network, eliminating the need for client-side modifications. To evaluate the feasibility and practicality of the

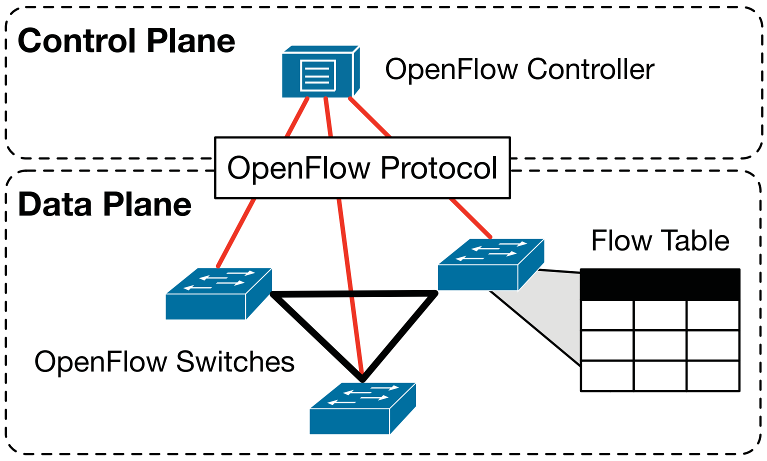


Figure 3.1.: Structure of an OpenFlow Network

proposed concept, we created Overseer, an OpenFlow controller for implementing an SDN-assisted application-aware network.

3.1. Background

This section briefly introduces the Software-Defined Network (SDN), OpenFlow and the issues with existing SDN-assisted routing approaches.

A Software-Defined Network (SDN) is a network architecture that allows programmable network behavior, which in turn enables more granular route optimization. OpenFlow is a widely-accepted de-facto standard implementation of an SDN [45]. Figure 3.1 illustrates the structure of an OpenFlow network. The OpenFlow networking model decouples a network into a *control plane* and a *data plane*. The *control plane* is the layer that governs the behavior of the network. The OpenFlow controller regulates the behavior of this plane by remotely managing rules in the flow tables of all OpenFlow switches in the OpenFlow network using the OpenFlow protocol. The *data plane* is the layer that handles data transfer. OpenFlow switches obey rules in the flow table allowing the *control plane* to “control” the *data plane*. By matching flows in the network with rules in the flow tables, OpenFlow switches perform associated actions such as forwarding or dropping communications.

3.1.1. Existing works involving SDN-assisted routing

Due to the limited scope of the research, existing work on SDN-assisted routing often has limited applicability [19, 28, 29, 37, 49, 62, 71]. For example, in the work by Egilmez et al., SDN-assisted routing was used to create a QoS-enabled adaptive video streaming service [19]. However, their solution could not coexist with the other SDN-assisted routing techniques. As another example, in the work by Watashiba et al., a system allowing a job scheduler to communicate with the OpenFlow network was introduced to improve the performance of a network-intensive high-performance computing (HPC) application [71]. However, the system assumes a fat-tree topology which only works with a traditional job-based parallel HPC application.

Many approaches also require specific setups or modifications to the client, which makes them more difficult to deploy [19, 49, 62]. For example, in the work by Maeda et al. and Nakasan et al., SDN-assisted routing was used in conjunction with Multipath TCP (MPTCP) to increase the data transfer throughput of a network [49, 62]. These works rely on MPTCP for handling multiple concurrent communications. Deploying MPTCP is a complicated process involving modification to the client's OS kernel [54].

In the paper by Ichikawa et al., they investigated the possibility of allocating routes specific to each connection according to network properties of each path [29]. Their implementation is geared toward improving virtual cluster performance and only works with Rocks cluster. Their technique, however, could be generalize to apply to many situations rather than targeted one.

In the work by Huang et al., GridFTP transfer speed is increased by using SDN to manage multiple parallelized connection to avoid congestion in a multi-path network [28]. Increasing bandwidth using multiple connections with distinct path is applicable to many situation. However, their implementation only work with GridFTP parallelized communication.

In the work by Watashiba et al., a system allowing job scheduler to communicate with OpenFlow network is introduced to improve the performance of network-intensive high-performance computing (HPC) application [71]. However, the system assume fat-tree topology is only works with traditional job-based parallel HPC application.

In the work by Tatsunori et al. and Nakasan et al., SDN-assisted routing is used in conjunction with MPTCP to increase data transfer throughput of a network [49, 62]. These works rely on MPTCP for handling multiple concurrent communications. Deploying MPTCP is a complicated process involving modification to client's OS kernel [54].

In the work by Egilmez et al., SDN-assisted routing is used to create QoS-enabled adaptive video streaming service [19]. However, their solution could not coexist with the other SDN-assisted routing technique.

CORONET try to make SDN more reliable by using tried-and-through methods such as VLAN in conjunction with SDN-assist routing [37]. Naturally, CORONET requires the switches to support VLAN. CORONET also lacks application classification thus make custom route planning a difficult task.

Although the mentioned research studies repeatedly share similar characteristics, it is difficult to apply them in situations different from their targeted use cases. A more general SDN-assisted routing approach or a framework that provides a common interface to organize and apply network optimizations could be a solution. It would not only increase the applicability of existing SDN-assisted routing methods, but also help avoid repeated efforts in the future.

3.2. Design and Implementation

This section describes the proposed application-aware routing approach, our general approach to SDN-assisted routing, as well as our implementation to clarify this concept.

3.2.1. Application-Aware Routing

Application-aware routing is a routing method which finds an optimal route specific to an individual application according to a routing strategy suited to its application class. We designed application-aware routing to directly address two issues with the current approach in SDN-assisted routing: limited applicability and requirements for client-side modifications. A network with application-aware routing is called an application-aware network.

The novelty of application-aware routing is the individual application routing. Current approaches in SDN-assisted routing apply the same routing strategy to every communication in the network. This approach substantially limits the applicability of the network. Routing each application individually allows multiple routing strategies to be applied simultaneously, thus expanding the range of applications the network can support.

The application-aware routing process also was designed to be self-contained, so that an application-aware network requires no client-side modification. There are two steps in realizing application-aware routing. The first step is identifying the application to determine its class. The second step is routing the application using the appropriate strategy for the application class. Efficient routing strategies also require near-realtime network information, such as topology, link bandwidth and link latency, which is difficult to monitor [15].

3.2.2. Application Class

The research considered two major application classes: *bandwidth-oriented applications* and *latency-oriented applications*. This classification was used because, although there are multiple factors influencing network application performance, bandwidth and latency are the two factors that have the most noticeable impact. A more sophisticated classification is being developed using an unsupervised machine learning technique. More details about this technique are discussed in Section 3.4.2.1.

Bandwidth-oriented applications is a class of applications that achieves higher performance when more bandwidth is available. Most applications that involve transferring a large amount of data fall into this category. Examples of applications in this category include HTTP, FTP and media (audio/video) streaming.

Latency-oriented applications is a class of applications that performs better with low latency communications. Usually, applications that involve remote controlling, real-time communications or applications that uses a lot of short communications for synchronization fall into this class. For instance, SSH is considered a latency-oriented application. Most online games, although using specialized protocols, often belong to this category as well.

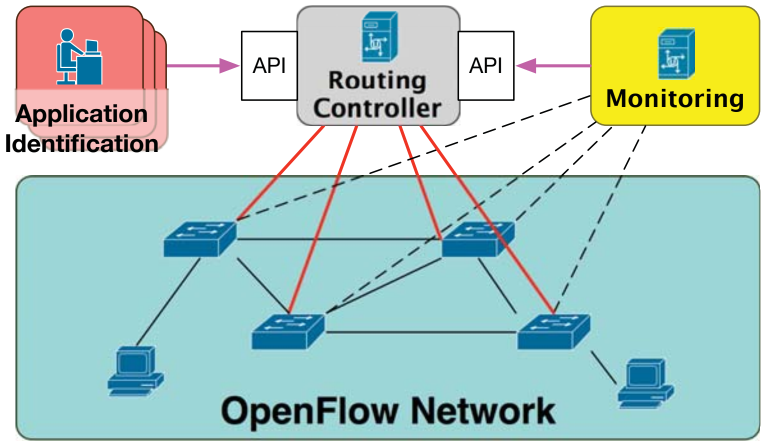


Figure 3.2.: Structure of an application-aware network

3.2.3. Architecture

Application-aware routing consists of three elements: application identification, monitoring, and the routing controller. Figure 3.2 illustrates the relationship between these elements and the OpenFlow network.

3.2.3.1. Application Identification

Applications are differentiated on a network flow (connection) level in the OpenFlow architecture. Applications' flows are identified with a *flow identifier* which is a quartet of source IP address, destination IP address, source TCP/UDP port, and destination TCP/UDP port. Identified flows are then classified as *bandwidth-oriented flows* (for bandwidth-oriented applications), *latency-oriented flows* (for latency-oriented applications), or *default flows* (no special routing preference) by matching with the *flow classification rules*. A *flow classification rule* is a pair of *flow identifier* and the corresponding application class. Each field in a *flow classification rule* could be either an exact value matching the targeted flows or a wildcard. Table 3.1 shows some example *flow classification rules*. With the current implementation, *flow classification rules* are provided manually by user input (by administrator or domain experts) or a network application through the routing component's API.

Table 3.1.: Example flow classification rules

Flow Identifier (SrcIP, SrcPort, DstIP, DstPort) (* denotes wildcard match)	Flow Class
(10.0.0.1, 80, *, *)	Bandwidth-oriented
(*, *, 10.0.0.1, 80)	Bandwidth-oriented
(10.0.0.1, 7000, *, *)	Latency-oriented
(*, *, 10.0.0.1, 7000)	Latency-oriented
(10.0.0.1, 7199, *, *)	Latency-oriented
(*, *, 10.0.0.1, 7199)	Latency-oriented
(10.0.0.1, 9160, *, *)	Latency-oriented
(*, *, 10.0.0.1, 9160)	Latency-oriented
(10.0.0.1, 9042, *, *)	Latency-oriented
(*, *, 10.0.0.1, 9042)	Latency-oriented
. . . Omitted to save space . . .	
(*, *, *, *)	Default

3.2.3.2. Monitoring

The monitoring element retrieves information on topology, link bandwidth, and link latency from the network. As routes are allocated reactively, the information needs to be monitored in a near-realtime manner so that the controller can react to changes in the network as soon as possible. Any monitoring solution can be used as long as it can communicate with the routing component's API.

For this research, Overlord [36] is used as the monitoring solution. Overlord implements a direct measurement method to achieve reasonably accurate measures while still retaining a near-realtime property, making it suitable for our application-aware network. Overlord consists of monitoring agents installed on each node and a central collecting server. These monitoring agents regularly communicate with each other to collect network measurements and report back to the central collecting server. The direct measurement method measures metrics by directly benchmarking values, producing very precise results. However, the benchmarking process also affects link performance during the measurement. Overlord minimizes this interference by intelligently deciding when each link should be benchmarked, and which links could be benchmarked simultaneously [36].

3.2.3.3. Overseer: Routing Controller

The routing controller element classifies flows with the *flow classification rules* and routes them with the corresponding routing strategy. Overseer, our implementation of the routing controller, is an OpenFlow controller. In this study, POX [8] was used as an OpenFlow controller development framework due to its maturity and simplicity.

For its operation, Overseer gathers information from the other elements. *Flow classification rules* received from the application identification element are stored in a *flow classification table.* For each flow in the application-aware network, Overseer classifies those flows and routes them using the corresponding routing strategy along with information from the monitoring element.

Information is gathered from the other elements by exposing a set of APIs. The APIs were implemented using the JSON-RPC protocol [33], since POX already provided support for this protocol. The current implementation is still very primitive and does not take security into account. Figure 3.3 shows an example request and response from a successful JSON-RPC API call, adding the first *flow classification rule* shown in Table 3.1.

Overseer implements three routing strategies: maximize-bandwidth, minimize-latency, and minimize-path-length.

1. The *maximize-bandwidth strategy* for bandwidth-oriented applications routes flows through the path with the maximum bandwidth.

2. The *minimize-latency strategy* for latency-oriented applications routes flows through the path that has the minimum latency.

3. The *minimize-path-length strategy* is the default strategy, used for unmatched flows, which routes flows through the shortest path.

Routes between all host pairs are pre-calculated for all routing strategies using a slightly modified Dijkstra algorithm. Figure 3.4 shows some example pre-

```
{
  "method": "set_entry",
  "params": {
    "path_identifier": [
      "10.0.0.1",
      "80",
      "*",
      "*"
    ],
    "class": "bandwidth_oriented"
  },
  "id": 1
}
```

(a) Request

```
{
  "result": "",
  "id": 1
}
```

(b) Response

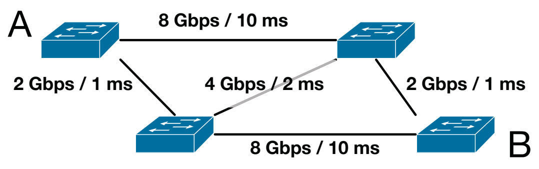
Figure 3.3.: Example JSON-RPC API call for adding a flow classification rule

calculated paths. Figure 3.5 illustrates the flow routing process. ld

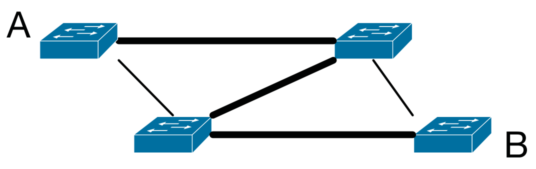
3.2.4. Example Use Case

Let us consider a 3-node Apache Cassandra cluster with one node also running a HTTP service. The IP addresses of each node range from 10.0.0.1 to 10.0.0.3. The HTTP service is running on node 10.0.0.1. Figure 3.6 illustrates the network topology of this cluster.

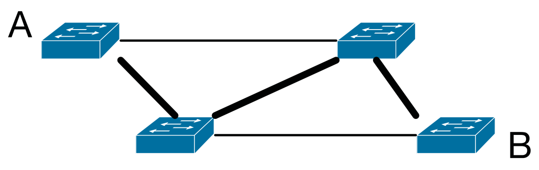
In order to enable the application-aware network to optimize the routing, the administrator must decide on the classification of the flows in the network and



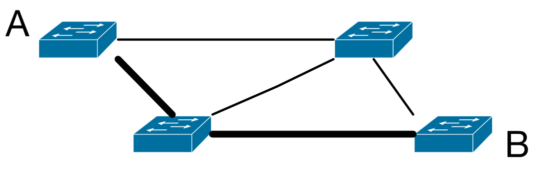
(a) Example network with bandwidth and latency information



(b) Path with **maximum bandwidth** from A to B



(c) Path with **minimum latency** from A to B



(d) Path with **minimum path length (hops)** from A to B

Figure 3.4.: Example network with corresponding calculated paths

design suitable flow classification rules to enforce such classification. The communication pattern of Apache Cassandra and the HTTP service are known to the administrator. Apache Cassandra is a distributed database management system that exhibits a latency-oriented communication pattern. More information about Apache Cassandra is discussed in Section 3.3.2.2. The HTTP service is used primarily to serve large files to the other hosts. With this knowledge, the administrator decided to classify Apache Cassandra's communication as latency-oriented and the HTTP service as bandwidth-oriented.

One method to achieve this classification is to configure the flow classification table as shown in Table 3.1. The first two rules in the tables match HTTP communications from and to 10.0.0.1 and classify them as bandwidth-oriented. The rest of the rules except the final one match Apache Cassandra's communications from and to all Apache Cassandra's hosts and classify them as latency-oriented. Rules for the other ports used by Apache Cassandra and the other Apache Cassandra's hosts are omitted to reduce the paper length. The final rule classifies everything else as default. This rule is not required for matching HTTP and Apache Cassandra communications but is useful in case more applications are deployed or more hosts are connected to the network in the future. The admin-

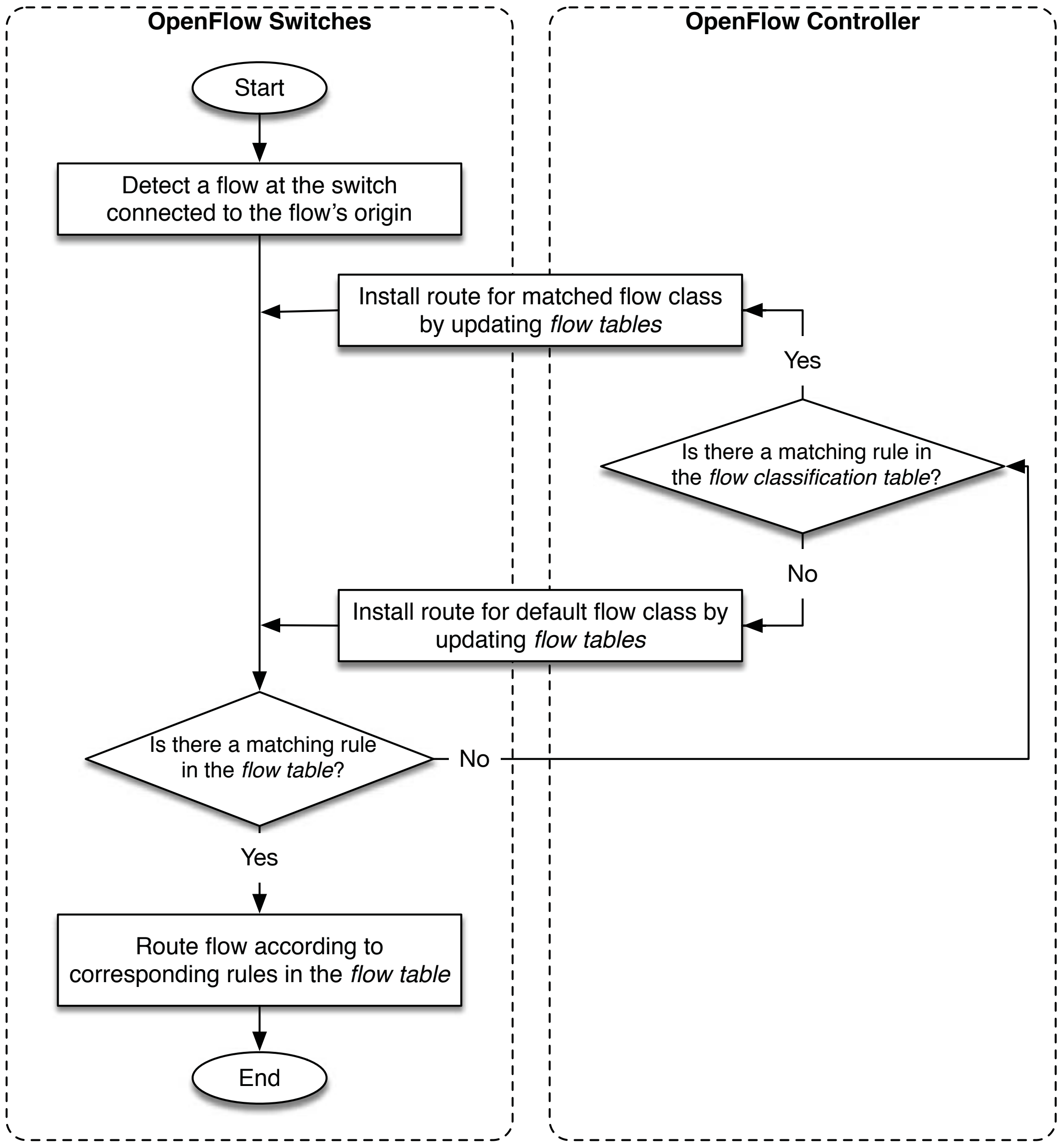

OpenFlow Switches
OpenFlow Controller
Start
Detect a flow at the switch connected to the flow's origin
Install route for matched flow class by updating *flow tables*
Yes
Is there a matching rule in the *flow classification table*?
No
Install route for default flow class by updating *flow tables*
Is there a matching rule in the *flow table*?
No
Yes
Route flow according to corresponding rules in the *flow table*
End

Figure 3.5.: Flow routing process

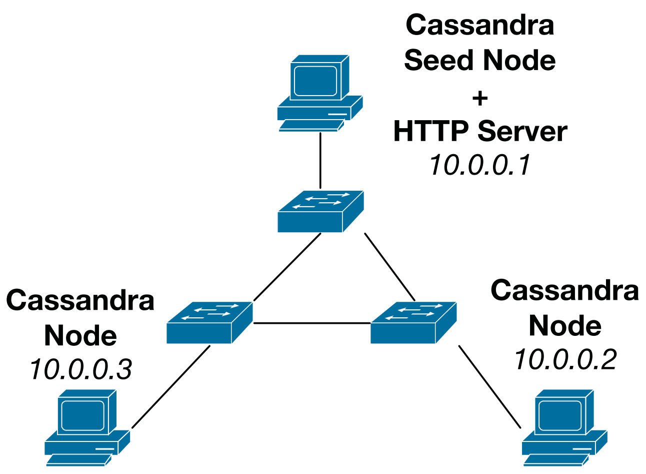


Figure 3.6.: Network topology of the example use case

istrator input these rules into the flow classification table using the JSON-RPC API.

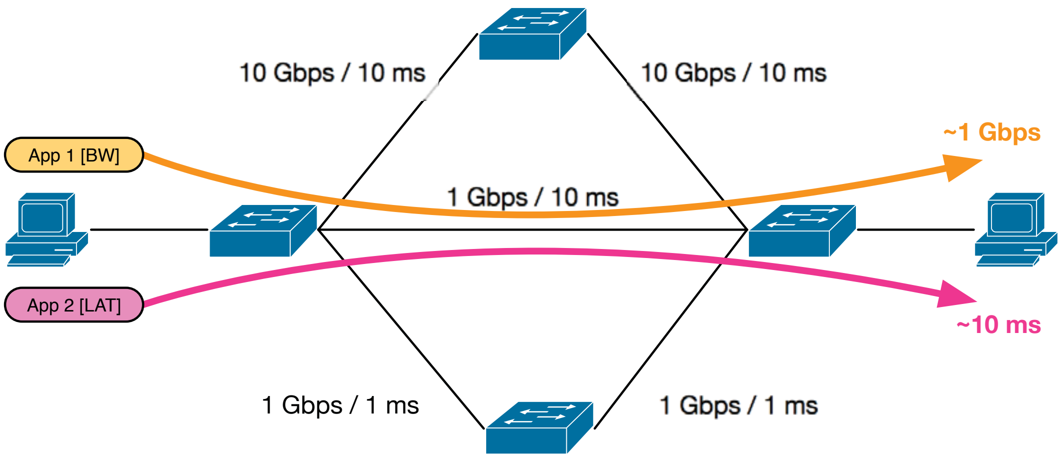
3.3. Evaluation

A series of evaluations are performed to assert the correctness of Overseer's implementation of the application-aware routing and to validate the feasibility and practicality of the application-aware network.

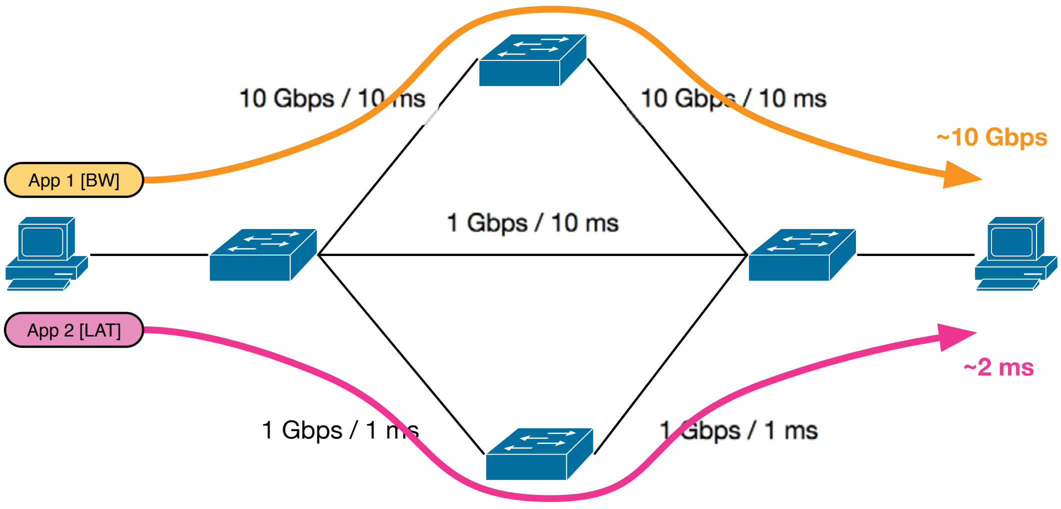
3.3.1. Overseer's Functionality Assertion

An emulation is used to assert that Overseer's functionalities, especially the application-aware routing, are correctly implemented. That is, Overseer must be able to properly determine and use the most appropriate paths when there are multiple paths with different properties to choose from according to information in the flow classification table.

Let us consider an example network scenario in Figure 3.7. This is an overlay network. Each link in the network is connected together not physically with real cables but virtually through multiple tunnels over the Internet. Numerous factors can affect performance of communication over the Internet in an unpredictable manner. These factors make tunnel performance vary in an unpredictable



(a) Shortest-Path Routing



(b) Application-Aware Routing

Figure 3.7.: Emulated network topology and the paths selected by shortest-path routing and application-aware routing

manner. The variation of tunnel performance then, in turn, propagates to the unpredictability of performance of each link in the overlay network and finally results in the uneven performance of each link as shown in the figure.

With the shortest-path routing (with paths selected by the spanning-tree protocol), all communication between leftmost host and rightmost host in this network take the middle path, which is the shortest. This is not an optimal path both in term of bandwidth and latency. Using these paths result in the less than optimal performance for both application 1 and application 2.

In application-aware routing, each network flow, although sharing the same source and destination, is routed separately according to its requirement. With the network performance information and application characteristics information, the network flows corresponded to the application 1 are routed through the upper path while the network flows corresponded to the application 2 are routed through the lower path. Utilizing these two paths should result in higher performance in term of bandwidth and latency for both application 1 and application 2.

3.3.1.1. Emulation Technique

The scenario is emulated Mininet, a commonly accepted network emulator within the OpenFlow research community [41]. Since only Overseer was being evaluated, the monitoring element was bypassed. Instead, the properties of each link are manually set through Overseer's API. Figure 3.8 shows the script that was used to emulate the network topology.

The emulation result indicated that Overseer performed as designed and the expected routes were taken as in the scenario.

3.3.2. Validity Assessment

Although our proposed application-aware routing approach was designed to increase the applicability of SDN-assisted routing as well as eliminating the requirement for specific client-side modifications, the validity of this method was still unproven.

To validate our proposed approach, we evaluated the feasibility and practicality of application-aware routing. We evaluated the feasibility, quantitatively

```python
from mininet.topo import Topo


class Usecase(Topo):
    def __init__(self):
        # Initialize topology
        Topo.__init__(self)

        # Add hosts and switches
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")
        s1 = self.addSwitch("s1")
        s2 = self.addSwitch("s2")
        s3 = self.addSwitch("s3")
        s4 = self.addSwitch("s4")

        # Add links
        self.addLink(h1, s1)
        self.addLink(h2, s4)
        self.addLink(s1, s2)
        self.addLink(s1, s3)
        self.addLink(s1, s4)
        self.addLink(s2, s4)
        self.addLink(s3, s4)

topos = {
    "usecase": (lambda: Usecase())
}
```

Figure 3.8.: Mininet script for creating topology in Figure 3.7

assessing whether application-aware routing significantly improved network performance, by comparing the performance of each implemented routing strategy with the shortest-path routing. Shortest-path routing was used as a baseline because it represents a common routing strategy in a traditional network. We evaluated practicality, assessing whether an application-aware network could work in a real situation, by using a real application and a real wide-area network environment.

For the evaluations, an OpenFlow testbed was created within the private cloud of Nara Institute of Science and Technology (NAIST). The cloud consisted of 6 virtual machine hosts. Figure 3.9 illustrates the structure of the testbed. Each host has a gigabit Ethernet connection that is shared among all virtual machines (VMs) running on the same host. To have the communication go through the physical network, six VMs were placed on six separate hosts. Each VM runs CentOS 6.5 with Open vSwitch 1.11 and the Overlord monitoring agent. The Open vSwitch on each VM is connected to the other VMs using GRE tunnels. Traffic control policies are applied to these tunnels during each experiment to simulate bandwidth and latency fluctuation as well as network congestion during the experiment. An extra tap interface was also connected to the Open vSwitch on each VM to allow IP address assignment to enable network-level connectivity required by Overlord. Overseer (the controller) and Overlord's central server were installed in a separate VM running Ubuntu 14.04 with the latest version of POX checked out from the "carp" branch as of July 2014.

3.3.2.1. Feasibility Evaluation with a Controlled Virtual Environment

The feasibility of the application-aware network was validated by measuring the significance of its performance gain compared to a traditional network. The performance gain was measured using the average bandwidth and latency of communication within an application-aware network, and then compared with the same measurements from a traditional network with the same topology.

There are some testbed configurations unique to this experiment. To simulate an extreme condition, a mesh topology is employed. Also, when no controller is available, the Open vSwitch can be configured so that it falls back to behaving like a traditional learning switch. The Open vSwitch also comes with a Spanning

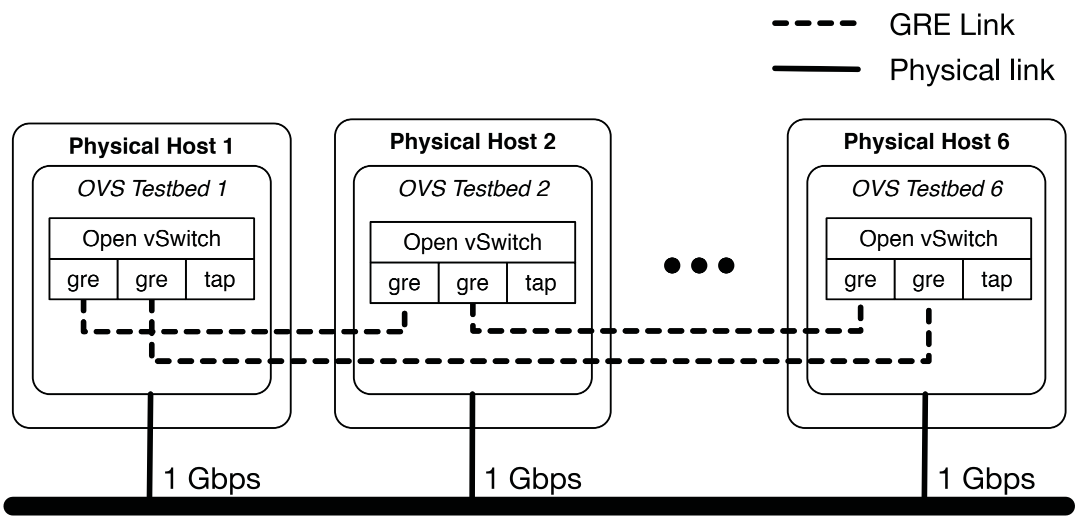


Figure 3.9.: Structure of the OpenFlow testbed in the NAIST private cloud

Tree Protocol (STP). These functions of the Open vSwitch were used to represent the traditional network.

Bandwidth Evaluation

Since the network of the cloud was shared across multiple running VMs, the bandwidth of all links in the testbed was limited to 100 Mbps to reduce interference from external factors. However, the limit enforced by the Linux kernel is not absolute and some fluctuation may occur. Netperf [32] was used to measure the bandwidth between all pairs of switches, each pair in both directions. The average value of all the measured bandwidth is used to represent the overall bandwidth of the network.

To simulate bandwidth fluctuation due to path dynamics, 5 links were selected at random. On these selected links, the bandwidth was further reduced to 40 Mbps. All pairs bandwidth measurement was performed, then average values were calculated again for both the traditional network and the application-aware network. For consistency, this process was repeated for 100 iterations. Finally, average results were calculated from all iterations. Table 3.2 and Table 3.3 show measures bandwidth between each pair of hosts from the first iteration of the

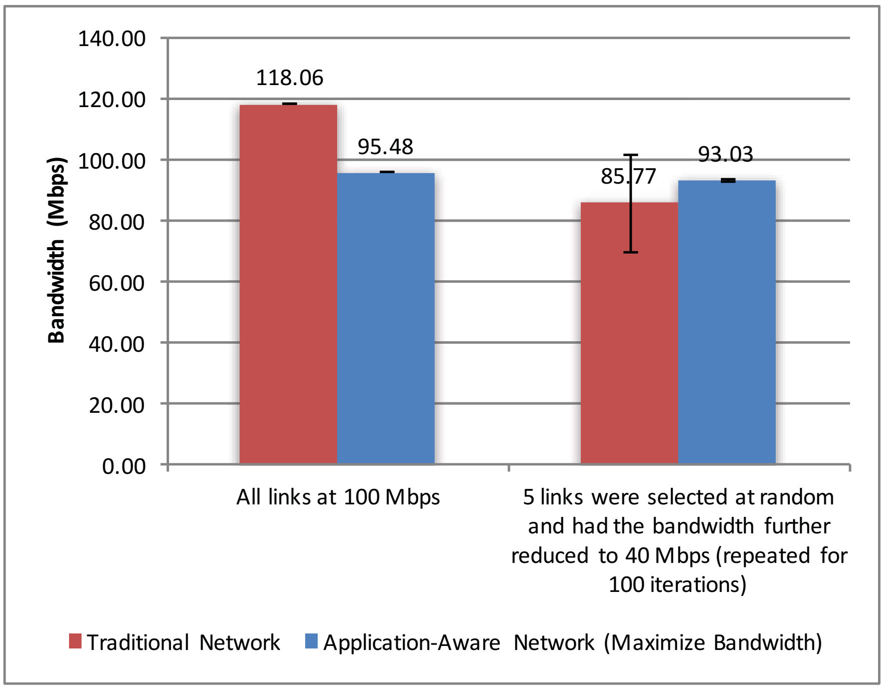


Figure 3.10.: Average bandwidth of the networks from bandwidth evaluation

bandwidth evaluation with fluctuation simulation. Figure 3.10 shows the average bandwidth measured for each case.

For the baseline case where all links are limited to 100 Mbps, the traditional network performed better than the application-aware network. This is due to the differences in bandwidth between each link not being substantial enough for the system to benefit from deliberately searching for a better route. Application-aware routing becomes an unnecessary overhead in this case. However, we believe that our implementation is imperfect as it was implemented as a proof of concept and there is still room for improvement.

In the case where 5 links were bandwidth limited, the application-aware network outperformed the traditional network because lower bandwidth links were avoided. As shown in Figure 3.10, with the traditional network, the results vary as depicted by the standard deviation. With the traditional network, the average bandwidth varies due to the number of links in the spanning tree that are being bandwidth limited. The application-aware network helps avoid this problem.

Table 3.2.: Bandwidth between all hosts from the first iteration of bandwidth evaluation with fluctuation simulation using shortest-path routing (Mbps)

From \ To	Host 1	Host 2	Host 3	Host 4	Host 5	Host 6
Host 1	N/A	46.5128	48.4373	46.504	48.438	48.4288
Host 2	46.5044	N/A	48.44	46.5068	48.4677	48.4271
Host 3	46.5049	46.5059	N/A	46.5107	121.693	116.2493
Host 4	46.4972	46.5092	48.4058	N/A	48.43	48.4391
Host 5	48.4451	48.4698	121.7273	48.4342	N/A	121.7179
Host 6	46.51	46.5044	116.2437	46.4991	121.7125	N/A

Table 3.3.: Bandwidth between all hosts from the first iteration of bandwidth evaluation with fluctuation simulation using application-aware routing (Mbps)

To / From	Host 1	Host 2	Host 3	Host 4	Host 5	Host 6
Host 1	N/A	93.6063	91.4219	93.4562	93.6367	92.5326
Host 2	92.5669	N/A	94.9994	91.0898	93.5846	91.3951
Host 3	93.547	95.039	N/A	97.4394	91.2292	95.0051
Host 4	93.6499	91.2594	97.4378	N/A	93.5199	93.5494
Host 5	93.4267	91.4581	91.4737	93.5258	N/A	95.041
Host 6	91.115	94.9889	91.2215	91.2452	97.4351	N/A

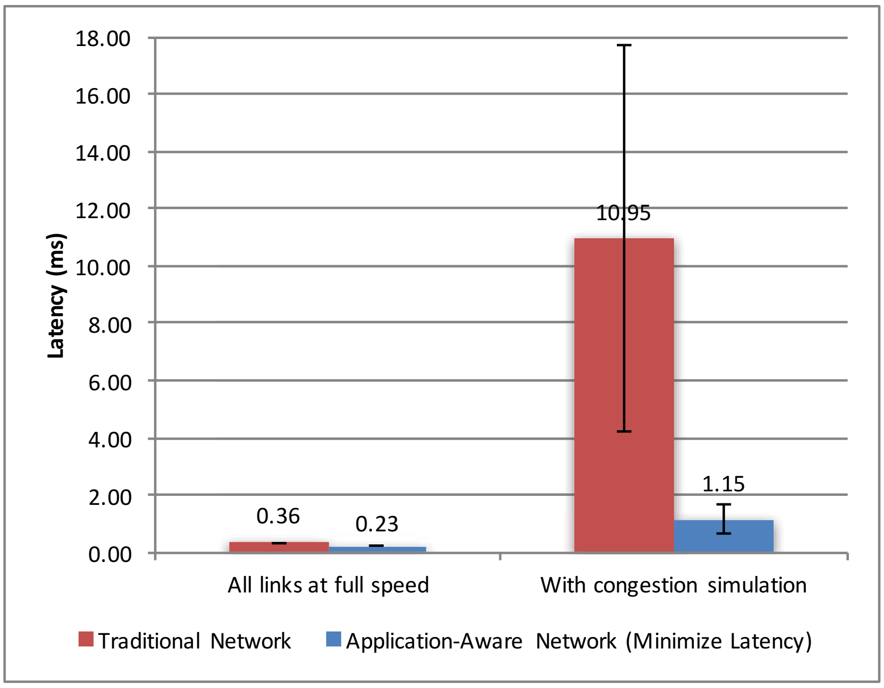


Figure 3.11.: Average latency of the networks from latency evaluation

Latency Evaluation

The average latency of a network is measured by averaging the mean latency between each pair of switches in the network. The mean latency between each pair of switches is obtained from the average produced by pinging ten times. Although the ping utility measures round-trip-time, this value directly corresponds to latency and is used in this experiment to represent latency.

The average communication latency of the application-aware network and the traditional network were measured. This experiment was performed in the same testbed as in Section 3.3.2.1 with no baseline latency added. For congestion simulation, again 5 links were selected at random which had 10 ms latency added. Again, the average results from 100 randomized iterations were used. Table 3.4 and Table 3.5 show measured latency between each pair of hosts from the first iteration of the latency evaluation with congestion simulation. Figure 3.11 shows the measured average latency of each routing technique in the two different configurations.

Table 3.4.: Latency between all hosts from the first iteration of the latency experiment with congestion simulation using shortest-path routing (ms)

From \ To	Host 1	Host 2	Host 3	Host 4	Host 5	Host 6
Host 1	N/A	40.58	20.448	20.439	20.35	20.442
Host 2	40.589	N/A	20.497	20.526	20.314	20.546
Host 3	20.515	20.531	N/A	0.434	0.273	0.394
Host 4	20.432	20.551	0.508	N/A	0.189	0.492
Host 5	20.336	20.327	0.191	0.205	N/A	0.317
Host 6	20.52	20.528	0.411	0.595	0.265	N/A

Table 3.5.: Latency between all hosts from the first iteration of the latency experiment with congestion simulation using application-aware routing (ms)

To / From	Host 1	Host 2	Host 3	Host 4	Host 5	Host 6
Host 1	N/A	0.276	0.346	0.204	0.338	0.601
Host 2	0.314	N/A	0.202	0.21	0.486	0.259
Host 3	0.401	0.199	N/A	0.433	0.211	0.364
Host 4	0.241	0.169	0.454	N/A	0.264	0.545
Host 5	0.473	0.353	0.171	0.21	N/A	0.216
Host 6	0.508	0.207	0.222	0.418	0.224	N/A

At full speed, the application-aware network performed better than the traditional network in terms of latency. This is because the application-aware network always uses the shortest paths whereas the traditional network always routes through the root node of the spanning tree, increasing the traveling distance of packets. With congestion simulation, the difference in term of latency between the application-aware network and the traditional network was even more extreme. The application-aware network noticed the congested links and avoided using those links. The traditional network, on the other hand, continued using the congested links.

With traditional routing, the results are highly variable, depending on which links are being latency-increased. The variation was even more drastic with added latency from congestion simulation, resulting in a severe penalty for not taking the best route. However, the application-aware network achieved very low and stable latency.

3.3.2.2. Practicality Evaluation with a Real-world Experiment

We evaluated practicality by measuring the performance of a real application in an application-aware network, and the performance of an application-aware network in a real wide-area environment.

A Real Application in an Application-aware Network

To validate the practicality of using an application-aware network with a real application, we evaluated the performance of Apache Cassandra [40] in an application-aware network. Apache Cassandra is a distributed database management system, which was chosen because it is a good example of a complex communication-heavy network application. Datastax's distribution of Apache Cassandra comes bundled with cassandra-stress, a stress testing tool for an Apache Cassandra cluster. This tool was used for the evaluation.

For this evaluation, a 6-node mesh network testbed was created in the same private cloud, in the same manner as for the feasibility evaluation. Apache Cassandra 3.0.6 was installed on each node, creating a 6-node cluster with 2 seed nodes. As shown in Figure 3.12, the network topology of the cluster was set up

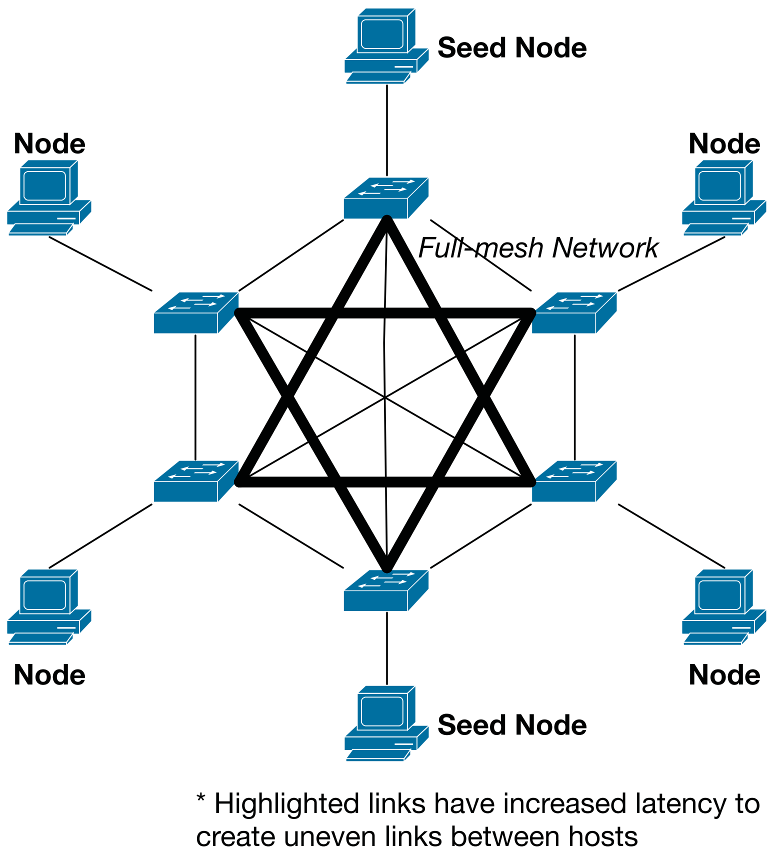

Seed Node
Node
Node
Full-mesh Network
Node
Node
Seed Node
* Highlighted links have increased latency to create uneven links between hosts

Figure 3.12.: Network topology of the experimental Apache Cassandra cluster

to create uneven links within the network while still having an equal number of full quality links and low quality links for each node. Each link has a bandwidth of 10 Gbps and a latency of 1 ms, with the highlighted links in the figure having their latency raised to 20 ms. An application-aware network was created in the same testbed as used for the feasibility evaluation.

Using default data populated by the *multiple concurrent writes* test, the cluster was stress tested with the *multiple concurrent reads* benchmark test. Each benchmark was performed multiple times with a different number of concurrent worker threads. The reads benchmark was used because reading from Apache Cassandra involves searching through the entire cluster, which involves a lot of short communications between hosts. This communication pattern is classified as latency-oriented.

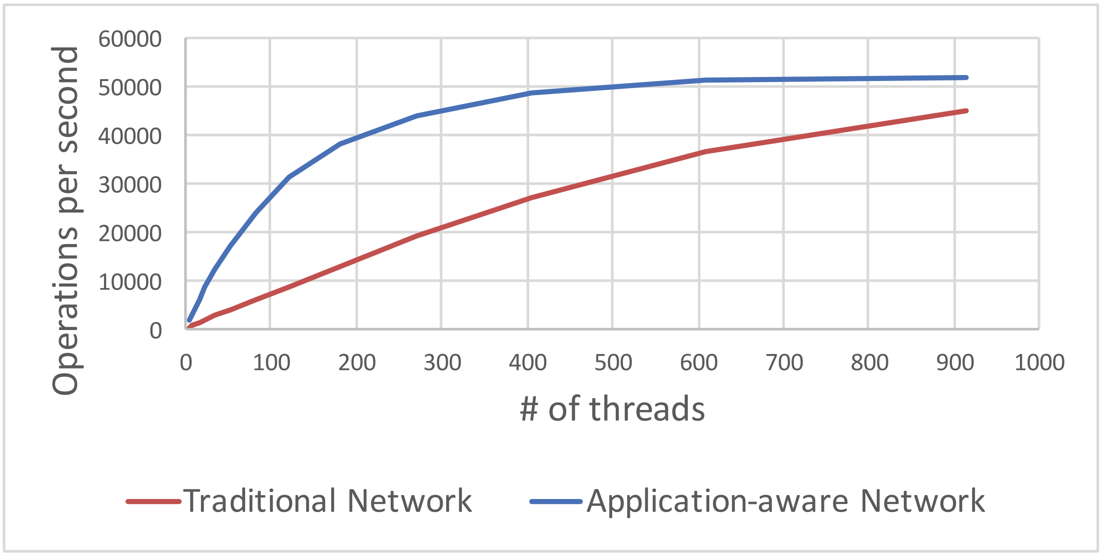


Figure 3.13.: Apache Cassandra read frequency of the test cluster in an application-aware network and a traditional network

The benchmark test was done on an application-aware network with the flow of Apache Cassandra set to *minimize_latency* and a custom routing that always used the shortest-path between hosts representing the best solution from the traditional network. From the results shown in Figure 3.13, the application-aware network outperformed the shortest-path network in every case. At a higher thread count, we observed a lower performance gain from the application-aware network. We suspect this result is due to the bottleneck of the communications shifting from latency-oriented to bandwidth-oriented as the number of threads increases.

An Application-Aware Network in a Real Wide-Area Environment

We tested an application-aware network created with Overseer on a portion of PRAGMA-ENT, a global-scale OpenFlow network testbed [30]. This testbed was built and is mainly used by the Pacific Rim Application and Grid Middleware Assembly (PRAGMA) [11,61]. Several institutions, mostly from Japan and the United States, contribute resources to this project. The testbed consists of both physical OpenFlow switches and Open vSwitch. Connections between the switches are in the form of both physical connections using a dedicated VLAN

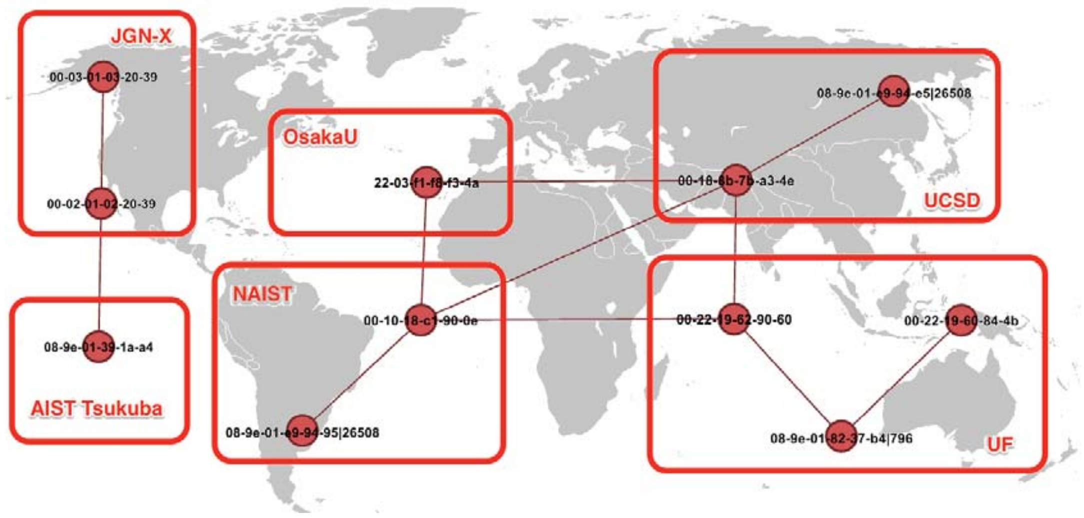


Figure 3.14.: Network topology of PRAGMA-ENT as of October 10, 2014

and virtual connections through a GRE tunnel. Figure 3.14 shows participating switches and network topology of PRAGMA-ENT as of October 10, 2014.

In this experiment, there were 4 participating hosts located at Nara Institute of Science and Technology (NAIST), Osaka University (OsakaU), University of California San Diego (UCSD) and University of Florida (UF). Each host ran Open vSwitch. All connections between hosts were created using GRE tunnels through the Internet. The performance of each connection is only limited by the actual Internet connection quality of each location. Figure 3.15 and Figure 3.16 illustrate the logical network topology of the portion of PRAGMA-ENT utilized in this experiment along with the measured latencies and bandwidths of each link.

This experiment focused on the uplink from OsakaU to UCSD as it provided the most explicit results. The bandwidth and latency from OsakaU to UCSD were compared for the application-aware network with a corresponding routing strategy and the traditional network using shortest path routing. In the case of application-aware routing, two *flow classification rules* were set. The first rule matched Netperf's flows and classified them as bandwidth-oriented. The other rule matched everything else, including ping's ICMP packet, and classified them as latency-oriented. Table 3.6 compares the result of all tests. From the results,

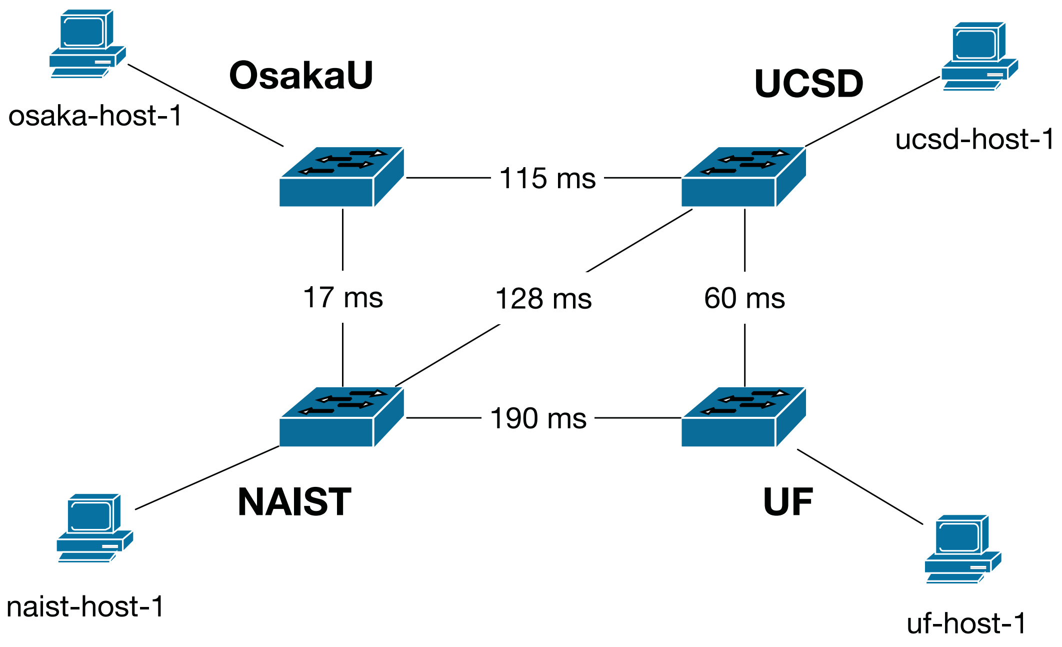


Figure 3.15.: Logical network topology and latency of each link in the experimental network portion of PRAGMA-ENT

the application-aware network (Overseer) achieved a higher bandwidth than the traditional network (Shortest Path) when it routed through NAIST, approaching the theoretical maximum value, although latency is the same for both cases as the shortest path is already the best route for minimum latency. From these results, we conclude that an application-aware network outperforms a traditional network in a real wide-area environment and is practical.

3.4. Discussion

From the evaluations, We concluded that an application-aware network is both feasible and practical. The rest of this section discusses advantages of Overseer as well as its drawbacks and possible mitigations for those drawbacks.

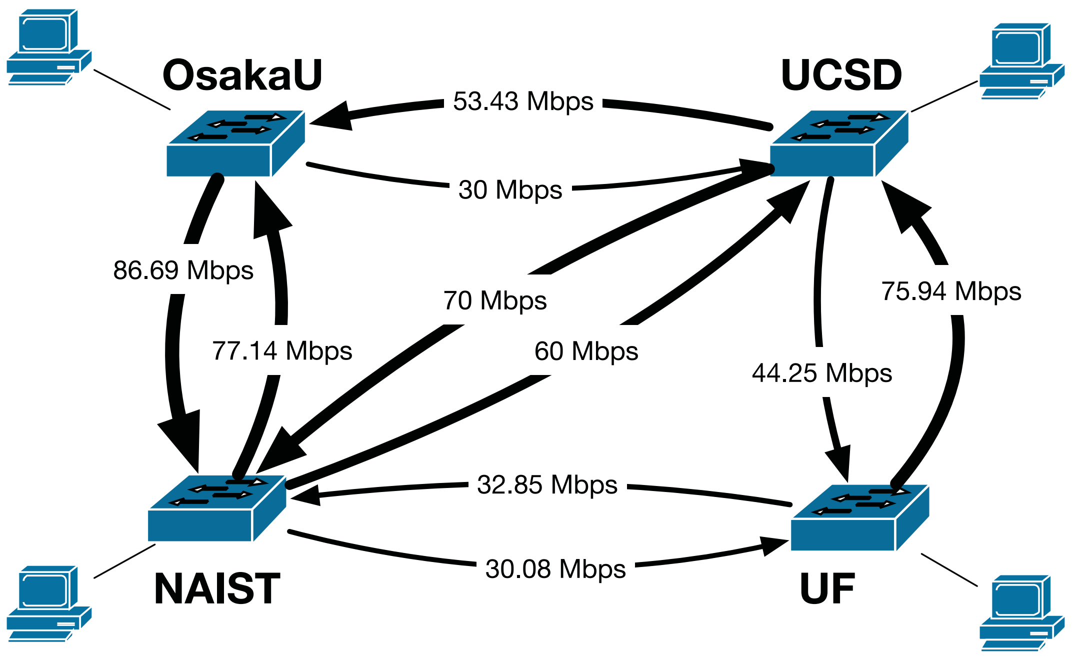


Figure 3.16.: Bandwidth of each link in the experimental network portion of PRAGMA-ENT

3.4.1. Advantages of Overseer

Due to the backward compatibility of its implementation, Overseer could be used as a drop-in replacement in a traditional network. If no rule is specified, Overseer uses shortest-path routing for all communications. Although Overseer adds some overhead to the routing process, a configuration with preemptive *flow classification rules* could be employed to reduce the overhead from reactive routing. A technique for automatic flow identification and classification technique is also being researched. Section 3.4.2 discusses more details regarding automatic flow identification and classification.

The components of an application-aware network communicate through APIs, allowing them to be replaced at will. For example, although Overlord provides fairly accurate information, it may be preferable to use another monitoring solution, such as PerfSONAR [25], for more stable monitoring and to reduce the impact on the network, at the cost of accuracy.

Overseer's API also allows tight integration with applications. As an example,

Table 3.6.: Measured bandwidth and latency from OsakaU to UCSD

Metric	Shortest Path	Overseer
Bandwidth	31.26 Mbps	59.25 Mbps
Latency	115 ms	115 ms

it is possible to live-modify *flow classification rules* to match desired communication characteristics at different stages of an application. The *flow classification rules* of an application flow could be set to bandwidth-oriented and then changed to latency-oriented at a later stage or vice-versa.

Overseer also could also be used as a building block for other SDN-assisted routing projects, since adding new network metrics, path types, and routing strategies is a relatively simple process.

3.4.2. Limitations and Mitigations

There are still several limitations with Overseer. This section discusses those issues and provides some possible mitigations and solutions.

3.4.2.1. Manual Application Identification

Currently, application identification requires manual input, with *flow classification rules* added manually through Overseer's API by the administrator or domain experts. This limitation greatly reduces the applicability of Overseer.

Automatic flow identification and categorization based on observed communication patterns is a possible solution. Research is being conducted on this approach. For example, AppMon is an OpenFlow proxy that provides flow identification and categorization using simple rules based on observations from network traffic analysis [69].

We have also been developing a deep-learning-based automatic flow identification and categorization technique. This technique is unsupervised so that it could extract natural classification of the network communication as opposed to using a defined categorization designed by the administrator or domain experts. However, preliminary results from the new technique suggested that bandwidth/latency is still a very good classification for route optimization.

3.4.2.2. Unfair Flow Distribution

Overseer allocates routes to each flow on a first-come-first-serve basis. However, this greedy approach does not guarantee Pareto optimal network utilization as the effect of flow order will modify the resulting flow routes. It may also lead to a longer time to approach convergence.

To mitigate this issue, flow prioritization, resource accounting, along with online optimization or flow rebalancing could be used. One of the major causes of unfair flow distribution is differences in flow lifetimes. There are two major flow types, long-lived flows and short-lived flows. A study has shown the importance of managing both types of flows, especially in data center networks [34]. It is possible for long-lived flows to be allocated to superior paths for an extended period, leaving other short-lived flows to compete with each other for inferior paths. To combat this, flow prioritization could occasionally de-prioritize long-lived flows and reallocate them to inferior paths, allowing short-lived flows to be allocated to superior paths and complete their communications faster. This prioritization method should result in higher throughput of the network.

It is also possible to take advantage of the monitored network properties of the entire network and calculate paths for all live flows collectively with an algorithm such as max-flow. All flows can be redistributed and updated periodically to achieve Pareto optimal utilization.

3.4.2.3. Scalability

Overseer, our proposed OpenFlow controller, could become a bottleneck limiting the scale of an application-aware network. Scalability is still an open issue of OpenFlow [31, 77]. Overseer does not take scalability into account. However, there are several techniques that could be used to increase the scale of an application-aware network [77].

To a certain degree, an efficient highly-available OpenFlow controller can be scaled. A load-balancing model could be utilized to scale a single OpenFlow network [76]. However, an application-aware network requires frequent information

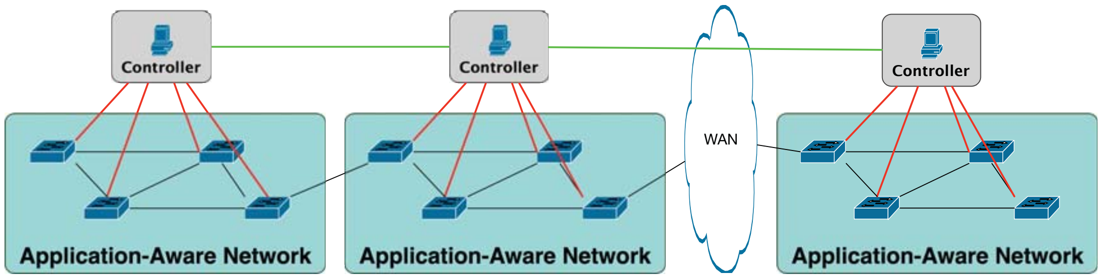

Controller
Controller
Controller
WAN
Application-Aware Network
Application-Aware Network
Application-Aware Network

Figure 3.17.: Interconnected multiple application-aware networks

updates which leads to data dependency between OpenFlow controller processes, which limits the effectiveness of this approach. Controller fail-over techniques also can increase the resiliency of the controller [18]. However, this does not increase processing throughput, and still suffers from the same problem as controller load-balancing.

It is more efficient to connect multiple application-aware networks instead of scaling a single application-aware network to support a large network. An application-aware network can be split into multiple smaller application-aware networks with their own OpenFlow controllers, instances of Overseer, communicating between each other to propagate *flow classification rules*. This approach is similar to that of Hyperflow [63]. Figure 3.17 illustrates the structure of interconnected multiple application-aware networks.

3.4.3. Utilizing Software-Defined Network and Application-Aware Routing to Enhance Communication Efficiency of a Data Center

The application-aware routing and the application-aware network is highly suitable for enhancing communication efficiency of a general-purpose cluster. An example of a general-purpose cluster includes a Mesos cluster. This technology could be used to accelerate network communication of a variety of applications in the cluster. In the environment with a wide range of applications, each with their own distinct communication characteristics, the application-aware network could detects network flows associated with each application and provide an optimal route for each individual flow.

In the case of a specifically-designed cluster where the network communication is highly efficient, SDN technology could still be applied to enhance the communication efficiency even further. This kind of cluster may not benefit as much from the application-aware routing and the application-aware network compared to a general-purpose cluster. Even in this case, the other SDN-based network optimizations could still be utilized to improve the communication efficiency. For example, SDN could be used to accelerate MPI_Reduce operation by eliminating redundant communications [44].

4. A self-optimizing application-aware network with deep-learning-based traffic classification

An automatic network flow categorization and identification is essential for realizing a self-optimizing network with an application-aware network. Network traffic categorization and identification, in general, is also useful for network monitoring, quality of service (QoS) control, network security, as well as data mining. Deploying an application-aware network involves configuring a predetermined flow classification and flow optimization rules for network flows corresponding to each application using the network. An automated method to create (and update) flow classification and flow optimization rules using communication data from the network would complete a cycle and create an automatic self-optimizing network. Figure 4.1 illustrates the concept of a self-optimizing network.

Generating a meaningful classification of network flows together with a corresponding class identifier (classifier) is challenging problem. A class of a network flow is defined as a non-linear representation of the network flow. The classifier model creation process is formulated as an unsupervised clustering of a raw network traffic capture. A raw network traffic capture is commonly available as a large unlabeled lightly-preprocessed dataset. While there are many clustering techniques, a deep-learning-based approach was chosen as it is a technique suitable for this kind of dataset.

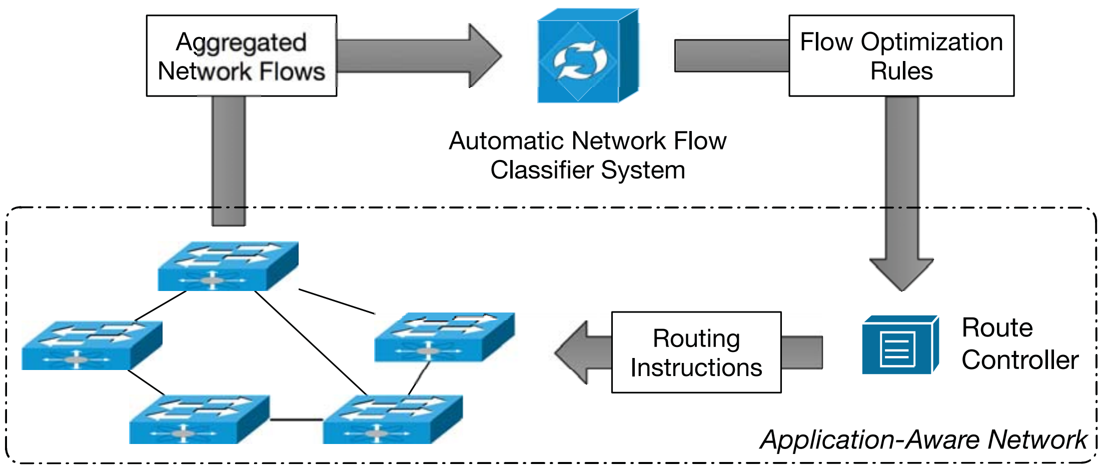


Figure 4.1.: The concept of a self-optimizing network

4.1. Background

In the scope of this work, network flows categorization and identification are defined as follow. Network flow categorization is a process of generating a meaningful network flow classification. The classification also has to be useful to the application-aware network meaning that the classification should be based on network properties that could be optimized with network routing. Network flow identification is to identify which class a network flow belongs to, corresponding to the classification.

4.1.1. Existing Approach

There were issues with the previous approaches to automatically categorize and/or identify network flows from network traffic data. Most of the existing works use an application-level classification which is too specific for route optimization with the application-aware network [20,35,48,51,60,70,73,79]. In the work by X. Wang, a decision tree with the limited number of predetermined features and thresholds were used [68]. While their work is applicable to the application-aware network, they used a predetermined bandwidth-bound and latency-bound classification with the assumption that the classification is appropriate for any network traffic without providing proper justification. A single-model-fit-all approach also limits the applicability of the model.

4.1.2. Deep Learning based Approach

A deep learning technique refers to a machine learning technique using deep neural network (DNN) model. While the neural network is not a new technology, a recent breakthrough in parallel computation allowed for a training of a much more complex network within a reasonable time. Multiple types of deep neural network are developed for various applications ranging from (unsupervised) clustering fix-sized input vectors to (supervised) classification of sequential time series data.

Using deep learning for network flows categorization and identification has several advantages. Deep learning is a technique that is highly suitable for capturing a non-linear representation from a large unlabeled lightly-preprocessed dataset. Since the classification of a network flow is a non-linear representation of the flow and a large unlabeled lightly-preprocessed network traffic data set is also readily available, deep learning is a natural choice for this task.

While there are some existing neural-network-based approaches, they do not work very well for creating a self-optimizing network with the application-aware network. Specifically, most of the approaches classify a network flow into a predetermined application-level classification [60, 70] which is too specific for application-aware network and does not guarantee to reflect a natural classification of the dataset. Some works also require labeled input [70] which is not readily available.

In this work, a deep neural network clustering is selected as it is a suitable deep-learning-based approach to network flows categorization and identification (for the application-aware network) with the following characteristics. The approach is unsupervised thus allowing the model to capture natural classification of the dataset without requiring prior knowledge which could bias the results. The resulting classification represents application's communication characteristic and is useful for route optimization.

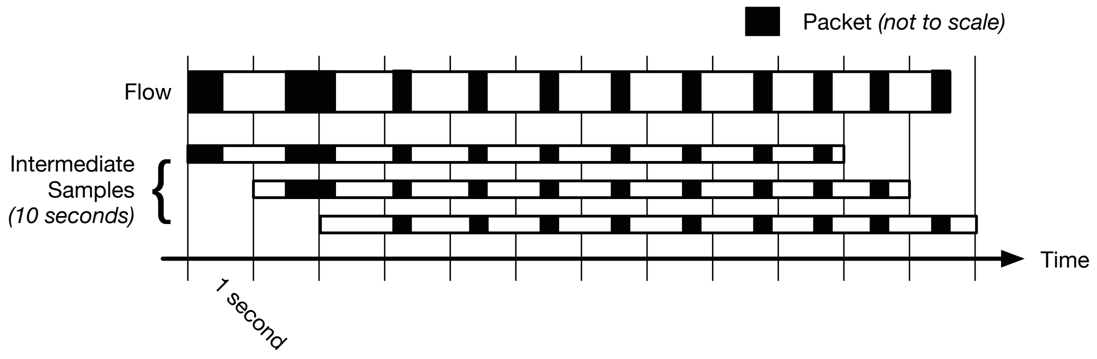


Figure 4.2.: Example intermediate training samples sliced from a flow

4.2. Model Creation Process

This section describes training samples preparation and the deep neural network clustering method to generating a network flow classification and a classifier.

4.2.1. Training Samples Preparation

Raw network traffic capture is transformed into training samples format suitable for clustering with a deep neural network model. Since the objective is to create an online network flow classifier for an application-aware network, the input format have to be highly scalable to be able to process the large amount of network flows in a near-real-time manner. A fixed-size input vector is chosen as the input format as it allows for parallel inferencing with multiple instances of the classifier. Protocol (TCP or UDP), source TCP/UDP port, and destination TCP/UDP port are also added to the input vector as these are common features that are known to be related to network application performance characteristic.

The training samples preparation process is divided into three steps as followed.

1. Raw network traffic is grouped by flows. Each flow is identified by a unique combination of source IP address, destination IP address, protocol (TCP or UDP), source TCP/UDP port, and destination TCP/UDP port.

2. A flow is sliced into multiple intermediate samples with a 1-second sliding window with 10 seconds in length. Each intermediate sample contains 10

Field	Value
Protocol	TCP
Source Port	80
Destination Port	56931
Transferred Size (second 0-1, Bytes)	796
. . . Omitted to save space . . .	
Transferred Size (second 9-10, Bytes)	$2,073$
Packet Count (second 0-1)	6
. . . Omitted to save space . . .	
Packet Count (second 9-10)	4

Table 4.1.: An example complete training sample

seconds of packets from the flow. Figure 4.2 illustrates the flow slicing window.

3. An intermediate sample is restructured with a transfer size binning and a packet count binning into a complete training sample. Packets in the intermediate sample are counted and binned into 10 1-second transfer size bins and 10 1-second packet count bins. Protocol (TCP or UDP), source TCP/UDP port, and destination TCP/UDP port are also added. Ultimately, each complete training sample is a vector with 23 dimensions. Table 4.1 shows an example complete training sample.

4.2.2. Training Deep Neural Network Model

A 3-layer stacked denoising autoencoder network with sigmoid activation functions is used in this work. Softmax functions are applied at the final layer to “squash” the output into the probability distribution. The error function is a standard mean square error function comparing the clean input and the reconstructed output. Adam optimizer, an optimization algorithm based on stochastic gradient descent, is used for the optimization with all parameters set to its default value [38]. Figure 4.3 illustrates the structure of the model.

A stack denoising decoder is an unsupervised deep neural network model struc-

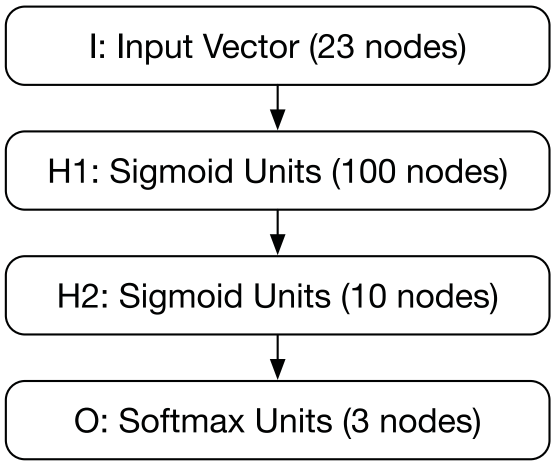

I: Input Vector (23 nodes)
H1: Sigmoid Units (100 nodes)
H2: Sigmoid Units (10 nodes)
O: Softmax Units (3 nodes)

Figure 4.3.: The 3-layer stacked denoising autoencoder model

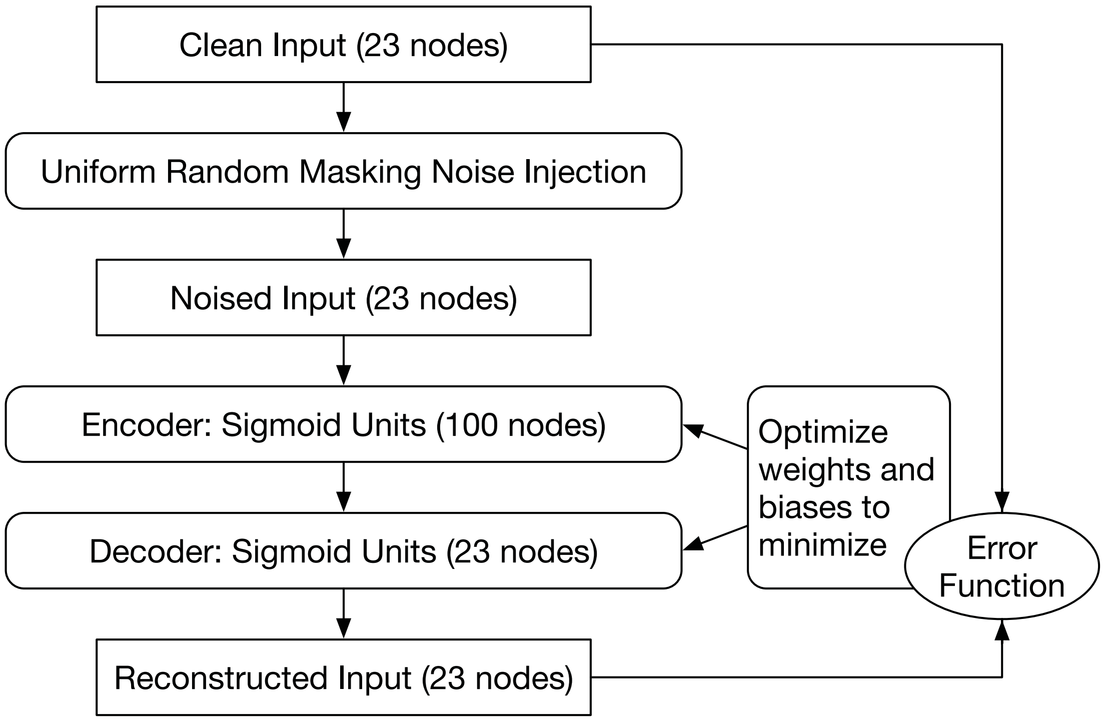

Clean Input (23 nodes)
Uniform Random Masking Noise Injection
Noised Input (23 nodes)
Encoder: Sigmoid Units (100 nodes)
Decoder: Sigmoid Units (23 nodes)
Reconstructed Input (23 nodes)
Optimize weights and biases to minimize
Error Function

Figure 4.4.: Training process of the first layer of the stacked denoising autoencoder model used in this work

ture that is commonly used for automatic feature extraction [65, 66]. The denoising process is introduced to the standard autoencoder to address overfitting problem which is prevalent in case the size of the encoded representation is larger than the size of the input vector. A stack denoising autoencoder model is trained on a layer-by-layer basis. For the training of each layer, the model takes a clean input vector and generate a noised input from the clean input. The noised input is fed to the encoder in the encoding step to create an encoded representation of the noised input. The encoded representation is then fed to the decoder in the decoding step to create a reconstructed input. The quality of the encoded representation is defined as the inverse of the error function. The error function is a distance function between the clean input and the reconstructed input. An optimizer is used to minimize the value of the error function as the training step goes on. After the training of each layer, the decoder is discarded and the training continues with the output from the trained encoder as an input to the next layer. Figure 4.4 illustrates the training process of the first layer of the stack denoising autoencoder model used in this work.

We have also experimented with the other networks such as deep denoising autoencoder. However, they did not produce useful results. Only the stacked denoising autoencoder is discussed in this paper.

4.3. Experiment Results

Several stacked denoising autoencoder models were trained with varying numbers of hidden layers, size of each hidden layers, and size of the output vector (numbers of output nodes). All models were trained with CAIDA Internet traffic dataset [17].

The following 3-layer stacked denoising autoencoder models produces the most promising results. The first two hidden layers contain 100 and 10 nodes respectively. The output layer has 3 nodes. Uniform random masking noise is used to create the noised input from the clean input. The model took around 9 hours to learn. Figure 4.3 illustrates the structure of the model.

While the model with 3-nodes at the output layer (3-outputs model) produces the best results, the results from the model with 4-nodes at the output layer (4-outputs model) are also shown for comparison. Since similar reasoning is also applied in comparing the 3-outputs model to the other models, the results from

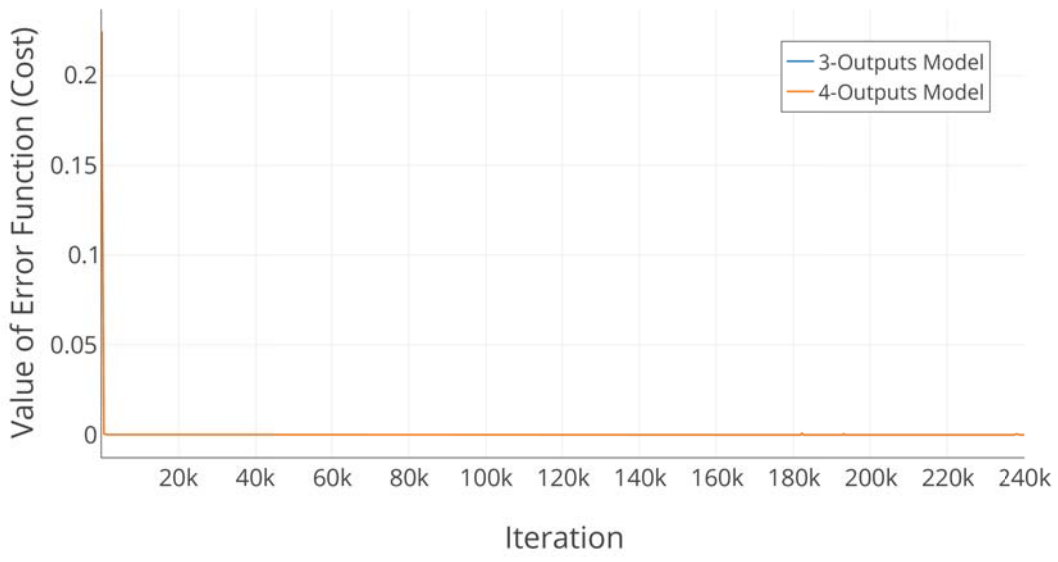


Figure 4.5.: Value of error function (cost) of the first layer of the model as the training progress

the other models are omitted to save space.

Value of error function (cost) after learning alone is not enough to evaluate the quality of the model. Figure 4.5, Figure 4.6, and Figure 4.7 show the cost associated with each layer of the 3-outputs model and the 4-outputs model as the training progress. The model was learned for the total of 240,000 iterations. The first layer was learned during the first 80,000 iterations of the training. The second and third layers were learned during the subsequent 80,000 iterations of the training respectively. The cost of the first and second layer of both model are identical as they share the same structure. The cost of the last layer of the two model do not significantly different.

The 4-outputs model does not produce useful classification. The model cluster network communication into four classes. Figure 4.8 and Figure 4.9 show properties distribution of classes generated with this model. The resulting classification is similar to the classification produced by the 3-outputs model. We could not find a meaningful interpretation of the class 2. The class 2 of this model also contains very little flows. These two points suggest that it is better to use the

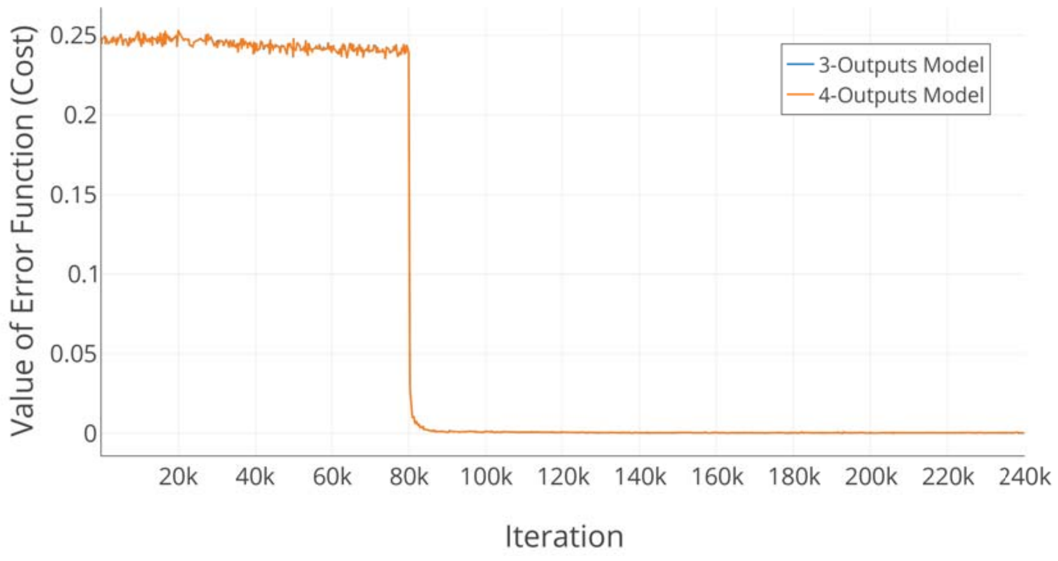


Figure 4.6.: Value of error function (cost) of the second layer of the model as the training progress

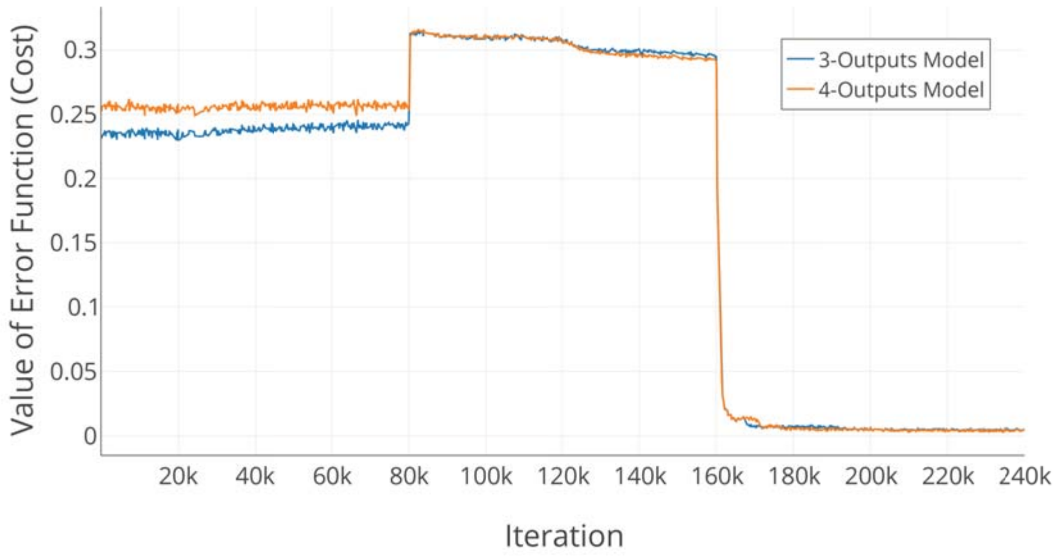


Figure 4.7.: Value of error function (cost) of the third layer of the model as the training progress

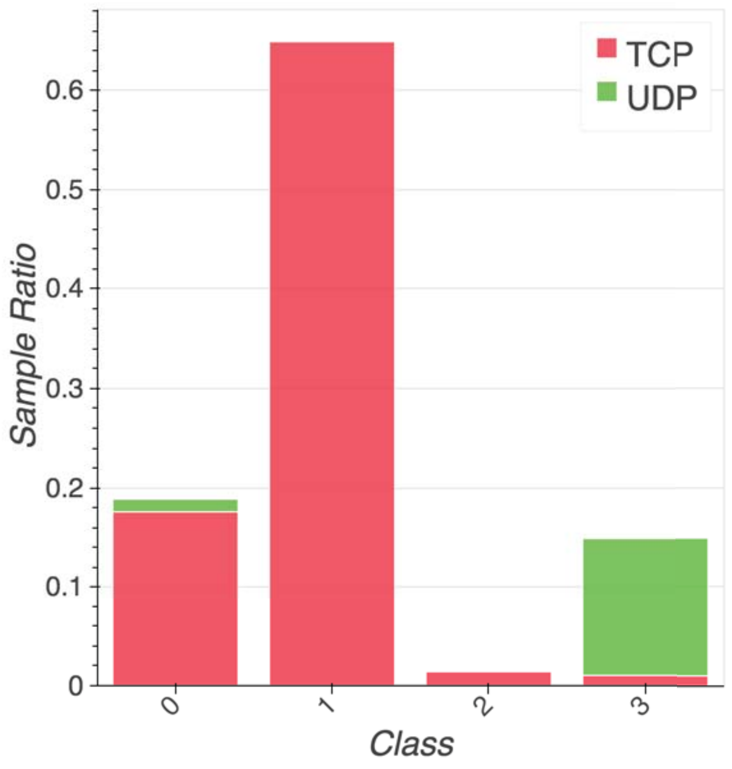
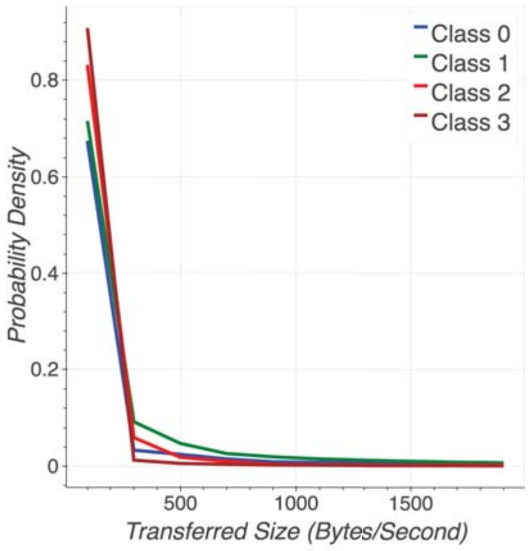


Figure 4.8.: Class size distribution of the 4-outputs model

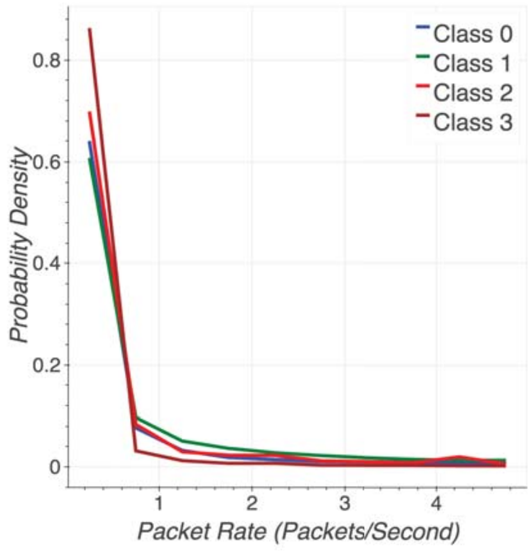
3-class classification of the 3-outputs models. Aside from class 2, it is possible to interpret the other classes in the similar fashion with the classification of the 3-outputs model.

The 3-outputs model cluster network communication into three classes. Figure 4.10 and Figure 4.11 show properties distribution of classes generated with this model.

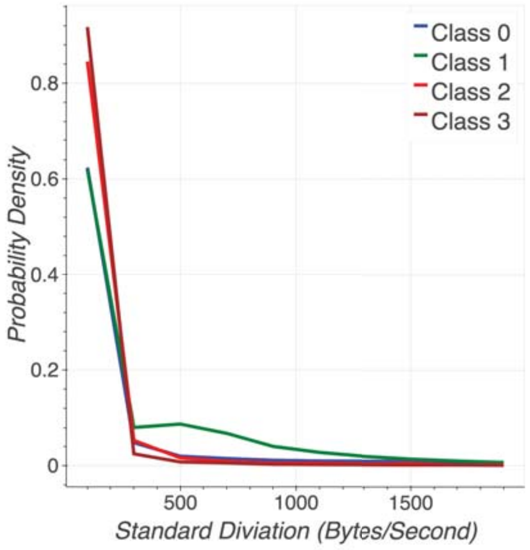
Information from Figure 4.10 and Figure 4.11 is used to interpret the meaning of the classes. From the Figure 4.11, we observe that all distributions of class 2 are grouped at the very small value compared to the other classes. According to the Figure 4.10, all UDP communications are also clustered into this class 2. We interpret these observations as class 2 representing a low-frequency communication considered as an irregular communication pattern. Comparing only class 0 and class 1, class 1 have a relatively higher transfer size and packet rate across the board. Transferred size standard deviation distribution of class 1 also



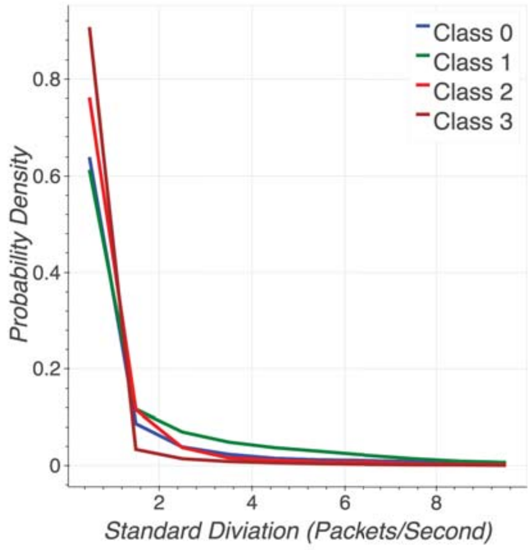
(a) Transferred size distribution per class



(b) Packet rate distribution per class



(c) Transferred size standard deviation distribution per class



(d) Packet rate standard deviation distribution per class

Figure 4.9.: Properties distribution of the resulting classification from the 4-outputs model

significantly higher the other classes. We interpret these observations as class 0 represents a regular-frequency communication with relatively lower packet size

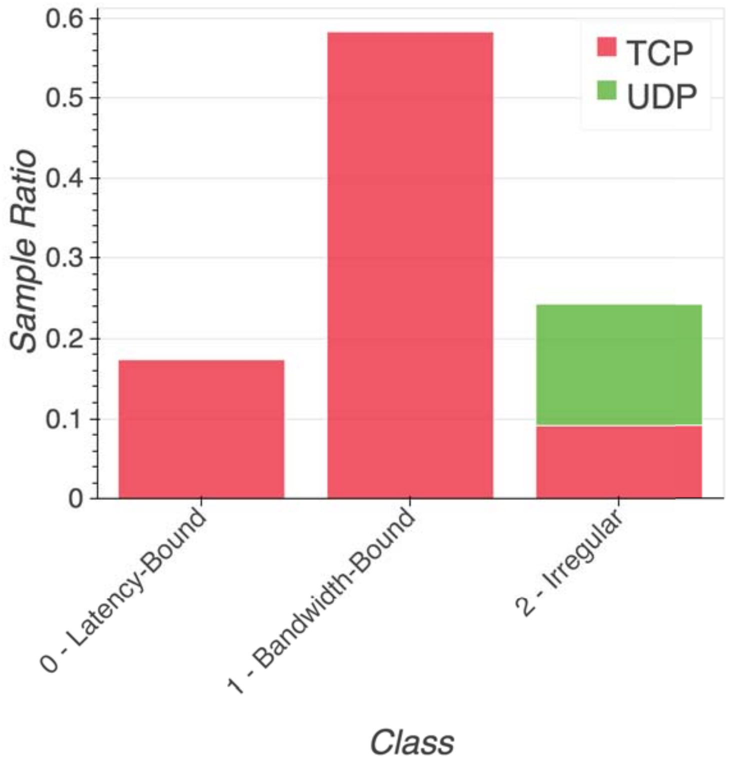
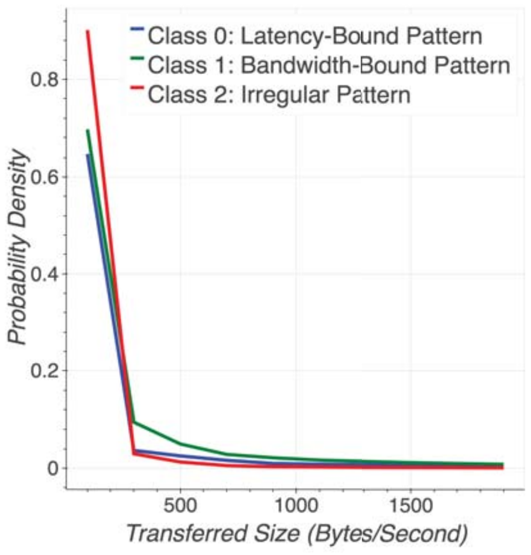


Figure 4.10.: Class size distribution of the 3-outputs model

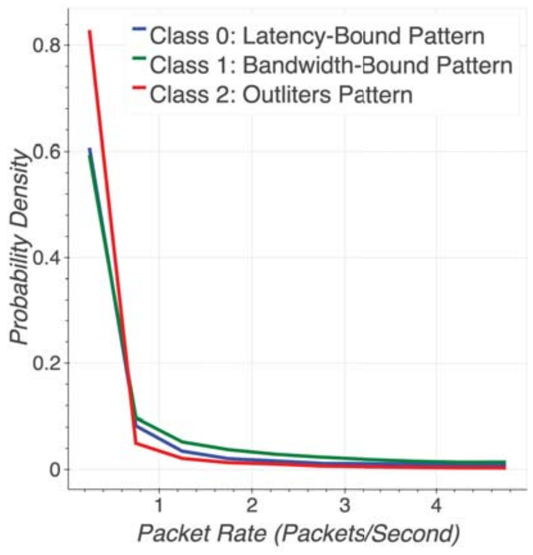
(latency-bound communication pattern) whereas class 1 represents a regular-frequency communication with relatively higher packet size (bandwidth-bound communication pattern). With these interpretations, the classes are interpreted as followed.

1. Class 0: Regular-frequency communication with relatively low packet size (latency-bound pattern).
2. Class 1: Regular-frequency communication with relatively high packet size (bandwidth-bound pattern).
3. Class 2: Low-frequency communication (irregular pattern).

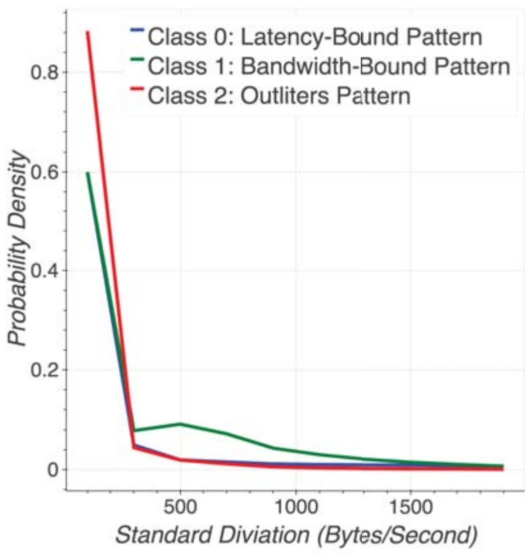
It is also important to note that the three classes discovered by this model are similar to our prior curated classification (bandwidth-bound/latency-bound classification) used in the development of the application-aware network [64].



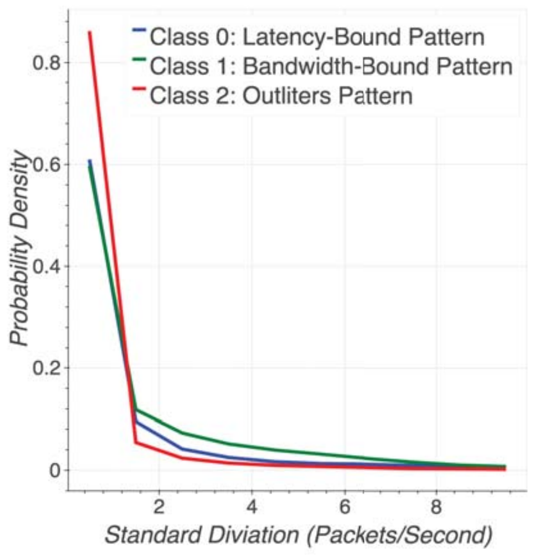
(a) Transferred size distribution per class



(b) Packet rate distribution per class



(c) Transferred size standard deviation distribution per class



(d) Packet rate standard deviation distribution per class

Figure 4.11.: Properties distribution of the resulting classification from the 3-outputs model

4.4. Self-Optimizing Network

The classifier model presented in this work, together with the application-aware network, can be used to develop an automatic self-optimizing network. Figure 4.1 illustrates the concept of a self-optimizing network.

An implementation of a self-optimizing network has to be able to operate in a high-throughput environment. Raw network traffic contains an enormous amount of data. Ideally, during each optimization cycle, every network flows have to be identified and categorized Then, a rule for each flow are created or updated to reflect the current classification of the corresponding flow. The flow categorization and identification process has to be highly scalable to cope with the amount of the raw network traffic data. Since it is not practical to create a rule for every single flow due to the limitation of OpenFlow, the implementation requires a method to prioritize the creation and maintenance of optimization rules for high-impact flows.

We propose the following design for an implementation of a self-optimizing network. At every edge switch, the ingress communication is mirrored out to a collector port. sFlow [52] could be used to output a uniform sampling of the communication at each switch instead of mirroring everything to reduce the throughput if required. The communication is gathered at the collector and bucketed into each identified flow. The network flow samples are constructed from the latest collected communication together with corresponding duration counter. To prioritize high-impact flows, only flow sample from the flows with the duration longer than a specific threshold value are sent to the classifier model for classification. Flow class identification (classification) can be parallelized for higher throughput with parallel model inferencing. The result of the classification is used to create and update the flow optimization rules for the application-aware network. Figure 4.12 illustrates the structure of this implementation.

This design is feasible and highly scalable. The sFlow sampling rate and the flow duration threshold can be optimized the achieve the acceptable balance between classification accuracy and scalability unique to each network. Parallel model inferencing also allows for scaling-out by increasing the numbers of the classifiers.

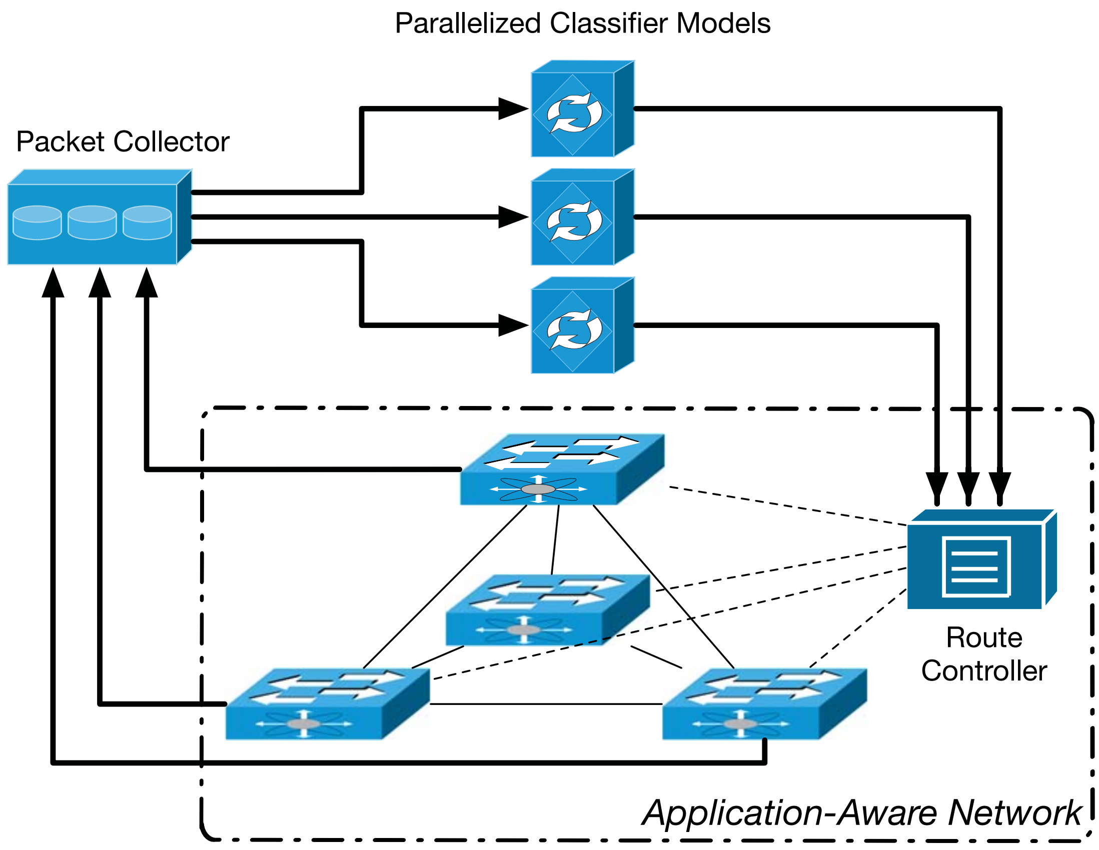


Figure 4.12.: Structure of the proposed design of a self-optimizing network

5. Conclusion

We present three contributions towards addressing the two factors of major impact on data center computation efficiency: the container rebalancing method, the application-aware routing and the self-optimizing application-aware network including a deep-learning-based network traffic classification system.

5.1. Container Rebalancing

Container rebalancing method is a novel scheduling mechanism with a rebalancing process working in conjunction with an existing scheduling process of Linux Containers (LXC) cluster. Container rebalancing takes advantage of LXC's rapid container migration to increase the optimal overcommit ratio of an LXC cluster. This improvement, in turn, increases overall resources utilization of the data center.

The simulation is used to evaluate the performance and validate the feasibility of container rebalancing. Although many simplifications and compromises have been made due to time constraints, the results still suggest that the container rebalancing is a promising method.

5.2. Application Aware-Routing and Application-Aware Network

An application-aware network enables route optimization for communication between tasks during group execution. This is done by identifying and routing each network flow independently. It also eliminates the need for client-side modification by classifying the applications on the network side. Overseer is an example of an OpenFlow controller implementing an application-aware network.

Feasibility and practicality of the application-aware network are validated by comparing an application-aware network with a spanning-tree-protocol-based network and a shortest-path routing network in multiple situations. From the results, the application-aware network achieved higher overall performance than the competitors. The reason being that by providing different routes for different classes of application, the network can support a wider range of applications. We conclude that the application-aware network is a promising technology. While there are still several issues with our implementation, Overseer, a list of possible mitigations is also presented.

5.3. A self-optimizing application aware network with deep-learning-based network traffic classification

A self-optimizing application-aware network eliminates the requirement of a manual application classification by using an automatic network traffic classification model. A deep-learning-based clustering technique is used to develop the classification model. We describe the development of a network flow categorization and identification model. The model was developed using a 3-layer stacked denoising autoencoder and trained with CAIDA Internet traffic dataset. 3-classes classification is discovered from the dataset. The three classes are a latency-bound pattern, a bandwidth-bound pattern, and an irregular pattern. This result suggested that deep learning is a feasible approach for network flow categorization and identification in the context of developing a classification model for a self-optimizing application-aware network.

The design of the self-optimizing network is also proposed. The design has to be highly scalable to be feasible given the size of the network traffic workload it was expected to handle. All design decision are described and rationalized in detail to demonstrate its scalability.

5.4. Future Work

Going forward, there are still more works to be done towards multiple aspects of our contributions.

Regarding the container rebalancing, more work is being done to investigate the effectiveness of this method and to improve the accuracy of the simulation. We are also looking to expand the work towards multi-objective optimization by including other optimization targets (i.e. memory usage, response time, and operation performance) and constraints (i.e. data locality).

Regarding the self-optimizing application-aware network including the deep-learning-based network traffic classification model, we are continually working to improve our deep-neural-network classification model as well as looking into the other model creation techniques. To keep pace with the dynamic of the network communication as time passes, we are working on finding optimal model update frequency and reducing model learning time if required. We are also looking to incorporate adaptive learning to continuously update the model as well. The real implementation of the self-optimizing application-aware network is also being developed.

Finally, while it is straightforward to implement both container rebalancing and a self-optimizing application-aware network together, we have yet to implement them together in a single environment. We are working towards implementing both container rebalancing and a self-optimizing application-aware network together in a real data center and performing a more comprehensive evaluation.

Appendices

A. Overseer's JSON-RPC API

Overseer communicates with other element of the application-aware routing suite by exposing a JSON-RPC API [33]. This API is used by the applications to register flow classification rules as well as monitoring element to continuously feeding measured network properties. The following methods were implemented:

A.1. get_table

Parameter : *None*
Output : All entries in flow preference table

A.2. get_entry

Parameter : Path identifier (JSON quadruple array)
Output : A single entry in flow preference table

A.3. set_entry

Parameters :

- Path identifier (JSON quadruple array)
- Path preference (possible options are `minimum_hop`, `maximum_bandwidth` and `minimum_latency`)

Output : *None*

A.4. remove_entry

Parameter : Path identifier (JSON quadruple array)
Output : *None*

A.5. update_bandwidth

Parameters :

- Source DPID (Hexdecimal)
- Destination DPID (Hexdecimal)
- Bandwidth (Mbps)

Output : *None*

A.6. update_latency

Parameters :

- Source DPID (Hexdecimal)
- Destination DPID (Hexdecimal)
- Latency (ms)

Output : *None*

Acknowledgements

I would like to thank the following people for their wisdom, guidance, and support, without whose help this work would never have been possible:

First and foremost, to Professor Hajimu Iida for providing a great research environment. His laboratory, Software Design and Analysis Laboratory, is a great place to pursue research.

To Professor Kazutoshi Fujikawa and Shoji Kasahara. Their constructive comments and feedback push my works come so far. Without them, my research works and the thesis would not be possible.

I would like to express deep appreciation to my supervisor, Associate Professor Kohei Ichikawa for his continuous support and guidance in my research work as well as my life in Japan. It was his valuable suggestions and comments that bring this research to fruition. Without them, I would not have successfully accomplished the Master course.

To Assistant Professor Putchong Uthayopas, who was also my advisor during my time as an undergraduate student at Kasetsart University. He gave me invaluable knowledge in research methodology and widened my vision insight in the area of high-performance computing. Part of this research was originally started during my time in Kasetsart University. His insightful suggestion helped shape this research in its initial stage. Without him, I could not come this far.

To Assistant Professor Yasuhiro Watashiba. His informative feedbacks and suggestions always helps raise the quality of this research.

To Professor Shinji Shimojo, Professor Susumu Date and Associate Professor Hirotake Abe for the great internship opportunity at Cybermedia Center, Osaka University. This valuable opportunity has led me to pursue the master course. They also gave a lot of beneficial points that I can use in order to improve my research.

Professor Philip M. Papadopoulos, Professor Nadya Williams, Dr. Teri Simas, Luca Clementi, Shava Smallen and PRAGMA community members for giving me an opportunity to get involved with PRAGMA research activities as well as an internship opportunity at University of California San Diego. PRAGMA community provides me access to their resources that is vital to this research. Without the privilege, this research would be impossible. The internship was also a highly worthwhile experience for me.

I would like to acknowledge my thesis committee. Thank you so much for reviewing my thesis and for the insightful comments and suggestions that helped me to improve the overall quality of this thesis.

Finally, I would like to thank the Monbukagakusho (MEXT) Scholarship for their monetary support. This scholarship enables me to live and pursue my research in Japan comfortably. It is a huge honor to be a recipient of this scholarship.

References

[1] 33.4. Overcommitting Resources. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Virtualization/sect-Virtualization-Tips_and_tricks-Overcommitting_with_KVM.html.

[2] AUFS storage driver in practice. URL: https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/.

[3] CoreOS is Linux for Massive Server Deployments. URL: https://coreos.com/.

[4] Kubernetes - What is Kubernetes? URL: http://kubernetes.io/docs/whatisk8s/.

[5] Linux Containers - LXC - Introduction. URL: https://linuxcontainers.org/lxc/introduction/.

[6] Swarm Overview. URL: https://docs.docker.com/swarm/overview/.

[7] IBM Knowledge Center | Platform Resource Scheduler 2.2.0 > Administering > Resource over commit allocation ratios, jan 2013. URL: http://www.ibm.com/support/knowledgecenter/SS8MU9_2.2.0/Admin/concepts/resourceovercommit.dita.

[8] POX Wiki, 2014. URL: https://openflow.stanford.edu/display/ONL/POX+Wiki.

[9] Overview — SimPy 3.0.8 documentation, 2015. URL: http://simpy.readthedocs.io/en/3.0.8/.

[10] Sameera Abar, Pierre Lemarinier, Georgios K. Theodoropoulos, and Gregory M.P. Ohare. Automated dynamic resource provisioning and monitoring in virtualized large-scale datacenter. *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, pages 961–970, 2014. `doi:10.1109/AINA.2014.117`.

[11] Peter Arzberger and Grace S. Hong. The Power of Cyberinfrastructure in Building Grassroots Networks: A History of the Pacific Rim Applications and Grid Middleware Assembly (PRAGMA), and Lessons Learned in Developing Global Communities. *2008 IEEE Fourth International Conference on eScience*, pages 470–470, 2008. `doi:10.1109/eScience.2008.55`.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, volume 37, page 164, New York, New York, USA, oct 2003. ACM Press. URL: `http://dl.acm.org/citation.cfm?id=945445.945462`, `doi:10.1145/945445.945462`.

[13] Anton Beloglazov and Rajkumar Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831, 2010. URL: `http://ieeexplore.ieee.org/document/5493373/`, `doi:10.1109/CCGRID.2010.46`.

[14] David S. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, sep 2014. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7036275`, `doi:10.1109/MCC.2014.51`.

[15] Y. Breitbart, Chee-Yong Chan Chee-Yong Chan, M. Garofalakis, R. Rastogi, and A. Silberschatz. Efficiently monitoring bandwidth and latency in IP networks. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 2, pages 933–942. Ieee, 2001. `doi:10.1109/INFCOM.2001.916285`.

[16] David Breitgand, Zvi Dubitzky, Amir Epstein, Alex Glikson, and Inbar Shapira. Sla-aware resource over-commit in an iaas cloud. *Proceedings of the 8th International Conference on Network and Service Management*, pages 73–81, oct 2012. URL: `http://dl.acm.org/citation.cfm?id=2499406.2499415`.

[17] CAIDA. The CAIDA Anonymized Internet Traces 2016 Dataset, 2016. URL: `http://www.caida.org/data/passive/passive_2016_dataset.xml`.

[18] Xiaoliang Chen, Bin Zhao, Shoujiang Ma, Cen Chen, Daoyun Hu, Wenshuang Zhou, and Zuqing Zhu. Leveraging master-slave OpenFlow controller arrangement to improve control plane resiliency in SD-EONs. *Optics Express*, 23(6):7550, 2015. `doi:10.1364/OE.23.007550`.

[19] Hilmi E. Egilmez, Seyhan Civanlar, and A. Murat Tekalp. An optimization framework for QoS-enabled adaptive video streaming over openflow networks. *IEEE Transactions on Multimedia*, 15(3):710–715, 2013. `doi:10.1109/TMM.2012.2232645`.

[20] Jeffrey Erman, Anirban Mahanti, Martin Arlitt, Ira Cohen, and Carey Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9-12):1194–1213, 2007. `doi:10.1016/j.peva.2007.06.014`.

[21] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. *Technology*, 25482, 2014. URL: `http://domino.research.ibm.com/library/CyberDig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf`.

[22] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The Cost of a Cloud : Research Problems in Data Center Networks. *ACM SIGCOMM Computer Communication Review*, 39(1):68–73, 2009. URL: `http://portal.acm.org/citation.cfm?doid=1496091.1496103`, `doi:10.1145/1496091.1496103`.

[23] Laszlo Gyarmati and Tuan Anh Trinh. How can architecture help to reduce energy consumption in data center networking? In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 183–186, 2010. URL: `http://portal.acm.org/citation.cfm?doid=1791314.1791343`, `doi:10.1145/1791314.1791343`.

[24] Ali Hammadi and Lotfi Mhamdi. A survey on architectures and energy efficiency in Data Center Networks. *Computer Communications*, 40:1–21, 2014. URL: `http://dx.doi.org/10.1016/j.comcom.2013.11.005`, `doi:10.1016/j.comcom.2013.11.005`.

[25] Andreas Hanemann, Jeff W. Boote, Eric L. Boyd, Jerome Jérôme Durand, Loukik Kudarimoti, Roman Łapacz, D. Martin Swany, Szymon Trocha, Jason Zurawski, Roman Lapacz, D. Martin Swany, and Jason Zurawski. PerfSONAR: A Service Oriented Architecture for Multi- Domain Network Monitoring. In *The Third international conference on Service-Oriented Computing*, pages 1–23. Springer-Verlag, 2005. `doi:10.1007/11596141_19`.

[26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, A Platform, Fine-Grained Resource, and Matei Zaharia. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, 2011. URL: `http://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdfhttps://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center`, `doi:10.1109/TIM.2009.2038002`.

[27] Jinhua Hu, Jianhua Gu, Guofei Sun, and Tianhai Zhao. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. *Proceedings - 3rd International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2010*, pages 89–96, 2010. `doi:10.1109/PAAP.2010.65`.

[28] Che Huang, Chawanat Nakasan, Kohei Ichikawa, and Hajimu Iida. A Multipath Controller for Accelerating GridFTP Transfer over SDN. *2015*

IEEE 11th International Conference on e-Science, pages 439–447, 2015. `doi:10.1109/eScience.2015.37`.

[29] Kohei Ichikawa and Hirotake Abe. A network performance-aware routing for multisite virtual clusters. In *19th IEEE International Conference on Networks (ICON)*, pages 1–5. Ieee, dec 2013. `doi:10.1109/ICON.2013.6781935`.

[30] Kohei Ichikawa, Pongsakorn U-Chupala, Che Huang, Chawanat Nakasan, Te-Lung Liu, Jo-Yu Chang, Li-Chi Ku, Whey-Fone Tsai, Jason Haga, Hiroaki Yamanaka, Eiji Kawai, Yoshiyuki Kido, Susumu Date, Shinji Shimojo, Philip Papadopoulos, Mauricio Tsugawa, Matthew Collins, Kyuho Jeong, Renato Figueiredo, and Jose Fortes. PRAGMA-ENT: An International SDN testbed for cyberinfrastructure in the Pacific Rim. *Concurrency and Computation: Practice and Experience*, (February), 2017. URL: `http://doi.wiley.com/10.1002/cpe.4138`, `doi:10.1002/cpe.4138`.

[31] Sally Johnson. Exploring OpenFlow scalability in cloud provider data centers, 2013. URL: `http://searchtelecom.techtarget.com/feature/Exploring-OpenFlow-scalability-in-cloud-provider-data-centers`.

[32] Rick Jones. NetPerf: a network performance benchmark, 1996. URL: `http://www.netperf.org/netperf/`.

[33] JSON-RPC Working Group. JSON-RPC 2.0 Specification, 2013. URL: `http://www.jsonrpc.org/specification`.

[34] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference - IMC '09*, page 202, 2009. `doi:10.1145/1644893.1644918`.

[35] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '05*, 35(4):229, 2005. URL: `http://portal.acm.org/citation.cfm?id=1090191.1080119%`

5Cnhttp://portal.acm.org/citation.cfm?doid=1080091.1080119, doi:10.1145/1080091.1080119.

[36] Nawawit Kessaraphong, Putchong Uthayopas, and Kohei Ichikawa. Building a Network Performance Benchmarking System Using Monitoring as a Service Infrastructure. In *The 18th International Computer Science and Engineering Conference*, pages 2–5, 2014.

[37] Hyojoon Kim, Mike Schlansker, Jose Renato Santos, Jean Tourrilhes, Yoshio Turner, and Nick Feamster. CORONET: Fault tolerance for software defined networks. *Proceedings - International Conference on Network Protocols, ICNP*, pages 1–2, 2012. doi:10.1109/ICNP.2012.6459938.

[38] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations 2015*, pages 1–15, 2015. arXiv:1412.6980.

[39] Avi Kivity, Uri Lublin, Anthony Liguori, Yaniv Kamay, and Dor Laor. kvm: the Linux virtual machine monitor. *Proceedings of the Linux Symposium*, 1:225–230, 2007. URL: https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=225, doi:10.1186/gb-2008-9-1-r8.

[40] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, 2010. doi:10.1145/1773912.1773922.

[41] Bob Lantz, Brandon Heller, and N McKeown. A network in a laptop: rapid prototyping for software-defined networks. *... Workshop on Hot Topics in Networks*, pages 1–6, 2010. URL: http://dl.acm.org/citation.cfm?id=1868466.

[42] Huang Lei, Jia Qin, Wang Xin, Yang Shuang, and Li Baochun. PCube: Improving Power Efficiency in Data Center Networks. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 65–72, 2011. doi:10.1109/cloud.2011.74.

[43] P. Mahadevan, S. Banerjee, P. Sharma, A. Shah, and P. Ranganathan. On Energy Efficiency for Enterprise and Data Center Networks. *Communications Magazine, IEEE*, 49(8):94–100, 2011. doi:10.1109/MCOM.2011.5978421.

[44] Pisit Makpaisit, Kohei Ichikawa, and Putchong Uthayopas. MPI_Reduce Algorithm for OpenFlow-Enabled Network. In *15th International Symposium on Communications and Information Technologies (ISCIT)*, pages 261–264, 2015.

[45] N McKeown and Tom Anderson. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[46] David Meisner, Brian T. Gold, and Thomas F. Wenisch. The PowerNap Server Architecture. *ACM Transactions on Computer Systems*, 29(1):1–24, 2011. URL: http://portal.acm.org/citation.cfm?doid=1925109.1925112, doi:10.1145/1925109.1925112.

[47] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, mar 2014. URL: http://dl.acm.org/ft_gateway.cfm?id=2600241&type=html.

[48] Andrew W Moore and Denis Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques Categories and Subject Descriptors. *Sigmetrics*, pages 50–60, 2005. doi:10.1145/1071690.1064220.

[49] Chawanat Nakasan, Kohei Ichikawa, Hajimu Iida, and Putchong Uthayopas. A simple multipath OpenFlow controller using topology-based algorithm for multipath TCP. *Concurrency and Computation: Practice and Experience*, (February), 2017. URL: http://doi.wiley.com/10.1002/cpe.4134, doi:10.1002/cpe.4134.

[50] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Symposium A Quarterly Journal In Modern Foreign*

Literatures, volume 21, pages 323–336, 2008. URL: `http://portal.acm.org/citation.cfm?id=1387612`, `doi:10.1.1.143.3471`.

[51] T T T Nguyen and G Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys and Tutorials, IEEE*, 10(4):56–76, 2008. `doi:10.1109/SURV.2008.080406`.

[52] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. sep 2001. URL: `http://dl.acm.org/citation.cfm?id=RFC3176`.

[53] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No " Power " Struggles : Coordinated Multi-level Power Management for the Data Center. *Solutions*, 36:48–59, 2008. URL: `http://portal.acm.org/citation.cfm?id=1346289`, `doi:http://doi.acm.org/10.1145/1346281.1346289`.

[54] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, number 1, pages 29–42, 2012. URL: `http://elf.cs.pub.ro/soa/res/lectures/mptcp-nsdi12.pdf`.

[55] Nathan Regola and Jean-Christophe Ducom. Recommendations for Virtualization Technologies in High Performance Computing. *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 409–416, 2010. `doi:10.1109/CloudCom.2010.71`.

[56] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael a. Kozuch. Heterogeneity and dynamicity of clouds at scale. *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*, pages 1–13, 2012. URL: `http://dl.acm.org/citation.cfm?id=2391229.2391236`, `doi:10.1145/2391229.2391236`.

[57] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, nov 2011.

[58] Malte Schwarzkopf and Andy Konwinski. Omega: flexible, scalable schedulers for large compute clusters. *EuroSys '13 Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013. URL: `http://dl.acm.org/citation.cfm?id=2465386`, `doi:10.1145/2465351.2465386`.

[59] Junaid Shuja, Sajjad A. Madani, Kashif Bilal, Khizar Hayat, Samee U. Khan, and Shahzad Sarwar. Energy-efficient data centers. *Computing*, 94(12):973–994, 2012. `doi:10.1007/s00607-012-0211-2`.

[60] Murat Soysal and Ece Guran Schmidt. Machine learning algorithms for accurate flow-based network traffic classification: Evaluation and comparison. *Performance Evaluation*, 67(6):451–467, 2010. URL: `http://dx.doi.org/10.1016/j.peva.2010.01.001`, `doi:10.1016/j.peva.2010.01.001`.

[61] Yoshio Tanaka, Naotaka Yamamoto, Ryousei Takano, and Akihiko Ota. Building Secure and Transparent Inter- Cloud Infrastructure for Scientific Applications. Technical report, 2013.

[62] Kato Tatsunori, Maeda, Hirotake, Abe, Kazuhiko. MPTCP with path selection mechanizm based on predicted throughput on OpenFlow-enabled environment. *IPSJ SIG Notes*, 2013(7):1–5, 2013.

[63] Amin Tootoonchian and Y Ganjali. HyperFlow: A distributed control plane for OpenFlow. *INM/WREN'10 Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3, 2010.

[64] Pongsakorn U-chupala, Yasuhiro Watashiba, Kohei Ichikawa, Susumu Date, and Hajimu Iida. Application-aware network: network route management using SDN based on application characteristics. *CSI Transactions on ICT*, 5(4):1–11, 2017. URL: `http:https://doi.org/10.1007/s40012-017-0171-y`, `doi:10.1007/s40012-017-0171-y`.

[65] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 1096–1103, 2008. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.149.8111%5Cnhttp://portal.acm.org/citation.cfm?doid=1390156.1390294%5Cnhttp://portal.acm.org/citation.cfm?doid=1390156.1390294`, `arXiv:arXiv:1412.6550v4`, `doi:10.1145/1390156.1390294`.

[66] Pascal Vincent Pascalvincent and Hugo Larochelle Larocheh. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion Pierre-Antoine Manzagol. *Journal of Machine Learning Research*, 11:3371–3408, 2010. `arXiv:0-387-31073-8`, `doi:10.1111/1467-8535.00290`.

[67] Lin Wang, Fa Zhang, Jordi Arjona Aroca, Athanasios V. Vasilakos, Kai Zheng, Chenying Hou, Dan Li, and Zhiyong Liu. GreenDCN: A general framework for achieving energy efficiency in data center networks. *IEEE Journal on Selected Areas in Communications*, 32(1):4–15, 2014. `arXiv:arXiv:1304.3519v2`, `doi:10.1109/JSAC.2014.140102`.

[68] Xuliang Wang. A Flow-level Monitoring Middleware for Automatic Flow Categorization. Master's thesis, Nara Institute of Science and Technology, 2016.

[69] Xuliang Wang, Pongsakorn U-chupala, Kohei Ichikawa, Yasuhiro Watashiba, Chantana Chantrapornchai, Putchong Uthayopas, and Hajimu Iida. Design of a flow-level monitoring middleware for automatic flow categorization. In *IEICE Technical Report*, 2016.

[70] Zhanyi Wang. The Applications of Deep Learning on Traffic Identification. In *Black Hat USA*, 2015.

[71] Yasuhiro Watashiba, Susumu Date, and Hirotake Abe. Efficacy Analysis of a SDN-enhanced Resource Management System through NAS Parallel Benchmarks. *The Review of Socionetwork Strategies*, 8(2):69–84, 2014.

[72] John Wilkes. More Google cluster data. Google research blog, nov 2011.

[73] Charles Wright, Fabian Monrose, and Gm M Masson. HMM profiles for network traffic classification. *. . . of the 2004 ACM workshop on . . .*, pages 9–15, 2004. URL: `http://dl.acm.org/citation.cfm?id=1029211`, `doi:10.1145/1029208.1029211`.

[74] M G Xavier, M V Neves, F D Rossi, T C Ferreto, T Lange, and C a F De Rose. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, 2013. `doi:Doi10.1109/Pdp.2013.41`.

[75] Jing Xu and Jose A B Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *2010 IEEE/ACM International Conference on Green Computing and Communications, GreenCom 2010, 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing, CPSCom 2010*, pages 179–188, 2010. `doi:10.1109/GreenCom-CPSCom.2010.137`.

[76] Chiba Yasunobu and Sugyou Kazushi. OpenFlow Controller Architecture for Large-Scale SDN Networks. *NEC Technical Journal*, 8(2):41–45, 2014.

[77] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, 2013. `doi:10.1109/MCOM.2013.6461198`.

[78] Yechiam Yemini. The OSI Network Management Model. *Communications Magazine, IEEE*, 31(5):20–29, 1993.

[79] Jun Zhang, Yang Xiang, Yu Wang, Wanlei Zhou, Yong Xiang, and Yong Guan. Network traffic classification using correlation information. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):104–117, 2013. `doi:10.1109/TPDS.2012.98`.

[80] Yan Zhang, Student Member, Nirwan Ansari, and United States. HERO: Hierarchical Energy Optimization for Data. *IEEE Systems Journal*, 9(2):2957–2961, 2015. `doi:10.1109/JSYST.2013.2285606`.

Publication List

Early version of the work in this thesis as well as the other work directly related to this thesis were published as listed below.

[1] P. U-Chupala, Y. Watashiba, K. Ichikawa, S. Date, and H. Iida, "Container Rebalancing: Towards Proactive Linux Containers Placement Optimization in a Data Center," *Computer Software and Applications Conference (COMPSAC)*, 2017, vol. 1, pp. 788–795. (Chapter 2)

[2] P. U-chupala, K.Ichikawa, P. Uthayopas, S.Date and H.Abe "Designing of SDN-Assisted Bandwidth and Latency Aware Route Allocation," in *IPSJ SIG Techinical Report*, Vol. 2014-HPC-145, No. 2, pp. 1–7, 2014. (Chapter 3)

[3] P. U-chupala, K. Ichikawa, H. Iida, N. Kessaraphong, P. Uthayopas, S. Date, and H. Abe, "Application-Oriented Bandwidth and Latency Aware Routing with OpenFlow Network," in *The 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2014. (Chapter 3)

[4] P. U-chupala, Y. Watashiba, K. Ichikawa, S. Date, and H. Iida, "Application-aware network: network route management using SDN based on application characteristics," *CSI Transactions on ICT*, vol. 5, no. 4, pp. 1–11, 2017. (Chapter 3)

[5] K. Ichikawa, M. Tsugawa, J. Haga, H. Yamanaka, T. Liu, Y. Kido, P. U-Chupala, C. Huang, C. Nakasan, J. Chang, L. Ku, W. Tsai, S. Date, S. Shimojo, P. Papadopoulos, and J. Fortes, "PRAGMA-ENT: Exposing SDN Concepts to Domain Scientists in the Pacific Rim," *PRAGMA Workshop on International Clouds for Data Science (PRAGMA-ICDS 2015)*, Oct. 2015. (Chapter 3)

[6] K. Ichikawa, P. U-Chupala, C. Huang, C. Nakasan, T. Liu, J. Chang, L. Ku, W. Tsai, J. Haga, H. Yamanaka, E. Kawai, Y. Kido, S. Date, S. Shimojo, P. Papadopoulos, M. Tsugawa, M. Collins, K. Jeong, R. Figueiredo, and J. Fortes, "PRAGMA-ENT: An International SDN testbed for cyberinfrastructure in the Pacific Rim," *Concurrency and Computation: Practice and Experience*, February, 2017. (Chapter 3)

[7] P. U-chupala, Y. Watashiba, K.Ichikawa and H.Iida "Towards Self-Optimizing Network: Applying Deep Learning to Network Traffic Categorization and Identification in the Context of Application-Aware Network," in *IPSJ SIG*

Techinical Report, Vol. 2018-IOT-40, No. 6, pp. 1–6, 2018. (Chapter 4)

The following publications are not directly related to the material in this thesis but were produced in parallel to the research performed for this thesis.

[1] S. Date, H. Abe, D. Khureltulga., K. Takahashi, Y. Kido, Y. Watashiba, P. U-Chupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, “An Empirical Study of SDN-accelerated HPC Infrastructure for Scientific Research,” *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, 2015, pp. 89–96.

[2] S. Date, H. Abe, D. Khureltulga, K. Takahashi, Y. Kido, Y. Watashiba, P. U-chupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, “SDN-accelerated HPC Infrastructure for Scientific Research,” *International Journal of Information Technology*, Vol. 22, No. 1, April 2016