# Proactive Configuration of Data Centre Networks for Big Data Processing

by

**Harpreet Singh, B.Sc.**

**Dissertation**

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

**Master of Science in Computer Science**

**University of Dublin, Trinity College**

September 2016

# Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Harpreet Singh

August 30, 2016

# Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Harpreet Singh

August 30, 2016

# Acknowledgments

Firstly, I would like to thank my supervisor Stefan Weber for making this dissertation possible with his invaluable guidance and support. I would also like to thank my family and friends, for their encouragement. And finally, I would like to thank Alexia, for her infallible moral support and encouragement.

Harpreet Singh

*University of Dublin, Trinity College*

*September 2016*

# Proactive Configuration of Data Centre Networks for Big Data Processing

Harpreet Singh, M.Sc.
University of Dublin, Trinity College, 2016

Supervisor: Stefan Weber

Various studies have determined that the network is a performance *bottleneck* in Big Data processing applications running in the cloud such as Hadoop. Numerous attempts have been made to alleviate this network bottleneck by traffic engineering during execution of the applications, using Software-Defined Networking. Such measures of traffic engineering are overwhelmingly *reactive* in nature and are bound to induce control traffic overhead in the network. In this project, we propose a *proactive* approach for configuring Data Centre Networks as the means to optimize application traffic, specifically Hadoop; thereby accelerating the execution of applications in the cloud.

We configure the network before execution of the application, to determine if there is a performance gain when there is no control overhead in the network. The network is configured *proactively*, by logging the flow decisions made by the *reactive* algorithms from previous studies. These flow rules are subsequently installed in the routing devices before the execution of the application, after which, the flows are routed *reactively*. We

demonstrate an *average gain in network bandwidth utilization* between 11.9% to 59.9% in comparison to *reactive* approaches, while Hadoop job completion times are reduced by 10% to 33.5%.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The ever growing organisations involved in search engines, social networking as well as applications for mobile devices, require efficient analysis of the massive collections of the data generated, which is achieved using thousands of commodity servers, resulting in the growing trend of *Big Data Analytics* [41]. Big data applications process data by distributing it across the data centre clusters, and after each piece has been analysed in parallel, the final data set is transferred and merged across the data centre network.

To support such communication patterns, data centre topologies make use of *multi-rooted tree* topologies in order to achieve higher speed links among the clusters and bypass the limitations caused by *limited port densities* in commodity switches [10]. The goal of *multi-rooted tree* topologies is to enable efficient communication among the network hosts by increasing the number of paths that interconnect them, thereby creating redundancy. Path multiplicity in *multi-rooted tree topologies* is achieved by horizontal scaling of hosts [9, 26, 30, 29] in order to cater to the increasing processing demands of big data applications.

Traditional network traffic routing protocols are designed for much simplistic communication patterns and topologies, with limited paths between host destination pairs, where the small number of redundant paths are used for fault tolerance, making them not suitable to be used for routing traffic in topologies with path diversity. Additional

obstacles in routing of big data application traffic are possible escalation in network traffic volumes, multiplicity of traffic patterns and high duration of data processing jobs. As a result, various studies [10, 26, 30] have highlighted that the network is a bottleneck in the performance of big data applications.

A significant amount of research [21, 56, 10, 45, 48, 57] has been conducted with the sole purpose of optimizing traffic in a data centre network for large-scale data analysis by devising new flow scheduling mechanisms and programming the entire network stack, so as to maximize throughput in the network, enabling better performance of big data applications. All work done in this field employs the emergent technology of Software-Defined Networking (SDN), in order to configure the network in accordance with big data application communication patterns.

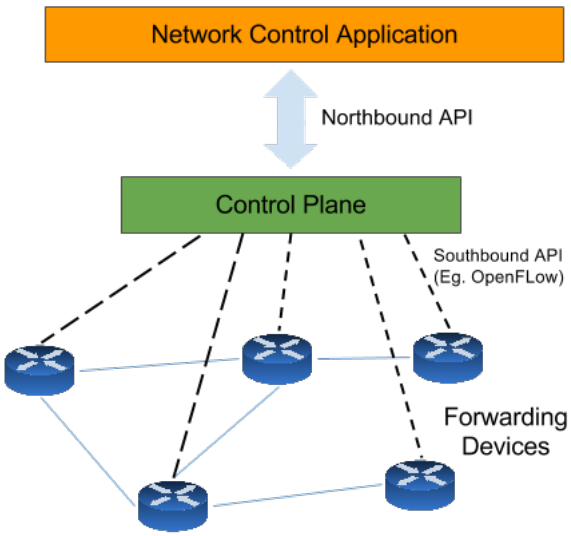


Figure 1.1: Fundamental abstractions of SDN.

SDN allows the control of all forwarding devices in the network from a global vantage point; it achieves this by separation of the control plane from the data plane of the forwarding devices, enabling centralized control of the network [42]. Moreover, it maintains the global state of the network and is responsible for the routing protocols being used in the network, enabling prototyping of new flow scheduling mechanisms. A high level overview of the SDN architecture is illustrated in Figure 1.1. SDN allows for network

programmability by installing flows in the switches, based on control logic that sits in a centralized controller which has a global view of the network. Hence, network applications can be developed which aim to maximize throughput by creating a tight-coupling between network flow scheduling and application communication patterns.

Consequently, using SDN, the data centre network can be optimized to handle the traffic patterns of big data applications in topologies with path diversity, without being limited by traditional routing protocols which are insufficient to handle such communication patterns.

## 1.1 Project Motivation

The motivation behind the current project is to device a flow scheduling approach in accordance with *big data* application patterns, in particular MapReduce [22], which is *proactive* in nature, so that the data centre network achieves maximum throughput and the application performs better. Existing research focuses on dynamic configuration of data centre networks by *reactively* scheduling flows at application runtime. Our goal is to investigate if there are performance gains in terms of the total bisection bandwidth achieved by hosts running big data applications in the network, when the network is optimized *proactively* based on application traffic patterns.

By taking a global view of the network using SDN, we preconfigure the scheduling system in order to avoid bottlenecks and minimize control traffic overhead in the network. We use Hadoop [31] as the application for proactive network integration due to it's widespread use and popularity.

## 1.2 Project Aims

We propose a *Proactive* approach for the configuration of a data centre network, which installs flow rules in the network before the big data application starts. In particular, we

intend to

- Determine whether there is an increase in the total bisection bandwidth achieved by the hosts in the network when flows are scheduled *proactively*.
- Investigate the effectiveness of *proactive* network configuration by comparing the Hadoop job completion times with *reactive* flow scheduling approaches and static Equal Cost Multi-Path (ECMP [33]) routing.
- Determine if there is an inverse correlation between total bisection bandwidth achieved and Hadoop job completion times for different flow scheduling strategies.

## 1.3 Project Approach

We present a design of our *Proactive* flow scheduler based on python, which leverages the flow scheduling decisions made by Global First-Fit [10] flow scheduling algorithm. Subsequently, we analyse the performance of our *Proactive* flow scheduler against ECMP routing and Global First-Fit flow scheduling on an emulation based testbed.

## 1.4 Document Structure

Chapter 2 explores the current state-of-the-art in efforts to optimize data centre network traffic for big data applications using SDN, and the emulators used for prototyping novel network control logic. Chapter 3 provides background for the project, explaining the working of MapReduce and its implementation in Hadoop, subsequently describing current state-of-the-art in data centre topologies such as the *fat-tree topology* [9], and finally providing an overview of ECMP routing and Global First-Fit flow scheduling which are used for evaluating our *Proactive* flow scheduling. Chapter 4 describes the design of our *Proactive* flow scheduler and gives a functional architectural overview of our experimental setup. The actual implementation of the design is described in Chapter 5. Chapter 6 de-

tails the benchmark tests used for evaluating our *Proactive* flow scheduler against ECMP routing and Global First-Fit flow scheduling and provides a brief analysis on the results obtained. Finally, Chapter 7 contains our concluding remarks.

# Chapter 2

# State-of-the-Art

In the previous chapter, we introduced our project, including our project motivation and aims. In this chapter, we describe the current state-of-the-art in the research area where our project lies and discuss relevant technologies and concepts upon which the current project is built. In Section 2.1, we discuss the concept of Software-Defined Networking, which is the fundamental building block of our project. Subsequently, in Section 2.2, we analyse and contrast studies that have used Software-Defined Networking to optimize Hadoop application traffic, on the basis of which, the current project is based. And finally, in Section 2.3, we describe emulators used for emulating data centre networks and Hadoop jobs respectively, which have made this project possible.

## 2.1 Software-Defined Networking

Fast paced innovations in Computing have accelerated the growth and adoption of technology throughout the world in every sphere of life. However, innovations in Computer Networking have not been very steady, part of it was because of the unwillingness to conduct experiments in a network carrying production traffic. Moreover, there were a large number of protocols and network equipment already deployed in the network, which made it more difficult to innovate [43].

Software-Defined Networking (SDN) enables controlling of the entire network through a centralized controller which maintains state of the entire network and is responsible for functions such as the routing protocols used by the forwarding devices in the network [38]. SDN achieves this by separating the control plane from the forwarding plane, where the control plane is housed in the centralized controller, and the forwarding plane is controlled via a narrow channel such as the OpenFlow protocol [43], as illustrated in Figure 2.1. Communication between the OpenFlow switches and the centralized controller is achieved over a secure channel which is encrypted via SSL.

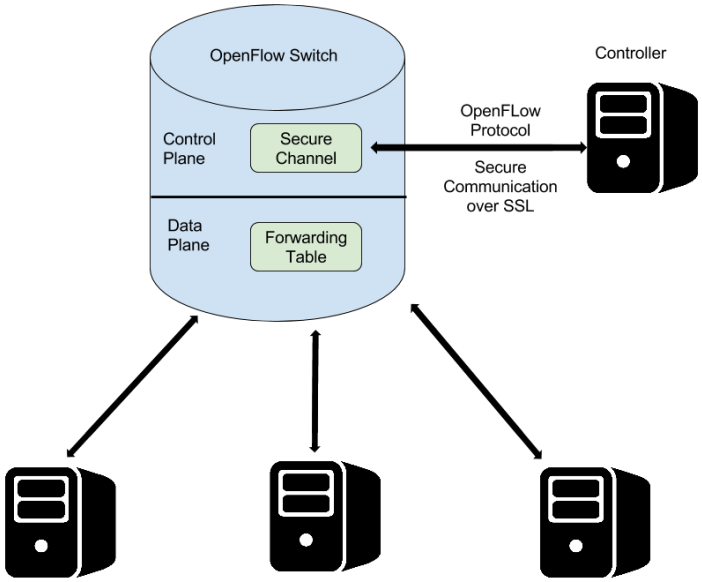


Figure 2.1: Overview of the operation of the OpenFlow protocol, where forwarding behaviour of switches is controlled by a centralized controller via a secure channel.

Consequently, new functionality can be added in networks that are already deployed without any modification to the switches, and prototyping of new ideas can be conducted without disrupting production traffic, enabling innovation in networking at software speeds [38].

SDN has resulted in a significant amount of research conducted with the aim of optimizing routing protocols and managing the network more efficiently [37]. Examples of SDN Networks include FlowVisor [52], PortLand [49] and Ethane [14]. In this project, we leverage SDN to optimize data centre networks for Hadoop [31] traffic.

## 2.2 Hadoop Traffic Optimization using SDN

Table 2.1: Comparison of various studies that attempt to optimize data centre traffic of applications such as Hadoop using SDN

| Study | FlowComb [21] | Wang *et al.* [56] | HybridTE [57] | Pythia [48] | Hedera [10] |
|---|---|---|---|---|---|
| Approach | Reactive | Reactive | Semi-Proactive | Reactive | Reactive |
| Methodology | Software agents on every node report demands, used to route traffic | Interface SDN controller with Hadoop master,which reports demands to controller | Install mice flows proactively and elephant flows reactively | Hadoop middleware predicts future transfers, network arranged accordingly | Devised new flow scheduling algorithms as an extension to ECMP |
| Uses Hadoop Traffic | Yes | Yes | No | Yes | Yes |
| Result | Reduced Hadoop Runtime by 33% | Proposal, No Results | Outperforms Hedera and ECMP in flow completion times | Reduced Hadoop runtime by 43% compared to ECMP | 39% better bisection bandwidth achieved for Hadoop shuffle phase, compared to ECMP |
| Limitations | Agents cause computational overhead, require domain specific knowledge | Requires creating demand estimator, computational overhead on master node | Underutilizes path diversity for mice flows, elephant flow detection suffer latencies | Domain-specific, intrusive to app data, induces computational overhead | 100ms latency in Scheduling due to control overhead, might increase when scaled to millions of hosts |
| Type | Application-Aware | Application-Aware | Application-Aware | Application-Aware | Traffic-Aware |

In this section, we review various recent studies [21, 56, 10, 45, 48, 57] that have been

conducted with the aim of optimizing application traffic of Big Data processing applications such as Hadoop [31] (described in detail in 3.1.2), in a data centre network using SDN; in order to achieve faster data processing times through high network utilization. We give a high level overview and comparison of these studies in Table 2.1. Most of the work done in this area uses *reactive* measures for data centre network configuration, which motivated us to explore a *proactive* approach, as described in Chapter 4.

All the studies compared in Table 2.1 employ *multi-rooted* tree topologies, which are described in detail in 3.2, while they compare their performance against Equal Cost Multi-Path (ECMP) [33] routing, which is the industry standard protocol for routing traffic in a multi-rooted tree topology having path diversity. ECMP is described in detail in 3.3.1. All approaches to optimize Hadoop traffic using SDN can be broadly classified into *Application-Aware networking* and *Traffic-Aware Networking*. Studies employing the two approaches are discussed in the next subsections.

### 2.2.1 Application-Aware Network Optimization

Figure 2.2 illustrates the approach of *Application-Aware* scheduling in order to optimize the network for Big Data applications. It attempts to tightly integrate Big Data application controllers with the SDN controller responsible for scheduling of flows in the network. The application controller reports traffic demands to the SDN controller on the bases of information obtained from the hosts, which is used by the SDN controller to configure forwarding devices in the network in a *reactive* manner. We further discuss this approach of network optimization by analysing studies that are based on this methodology for network optimization.

Das *et al.* [21] attempted to optimize Hadoop by introducing a network management framework called *FlowComb* that leverages SDN and monitoring of all servers running Hadoop, to perform dynamic scheduling of flows.

The main objectives of *FlowComb* were to predict the network demand of an ap-

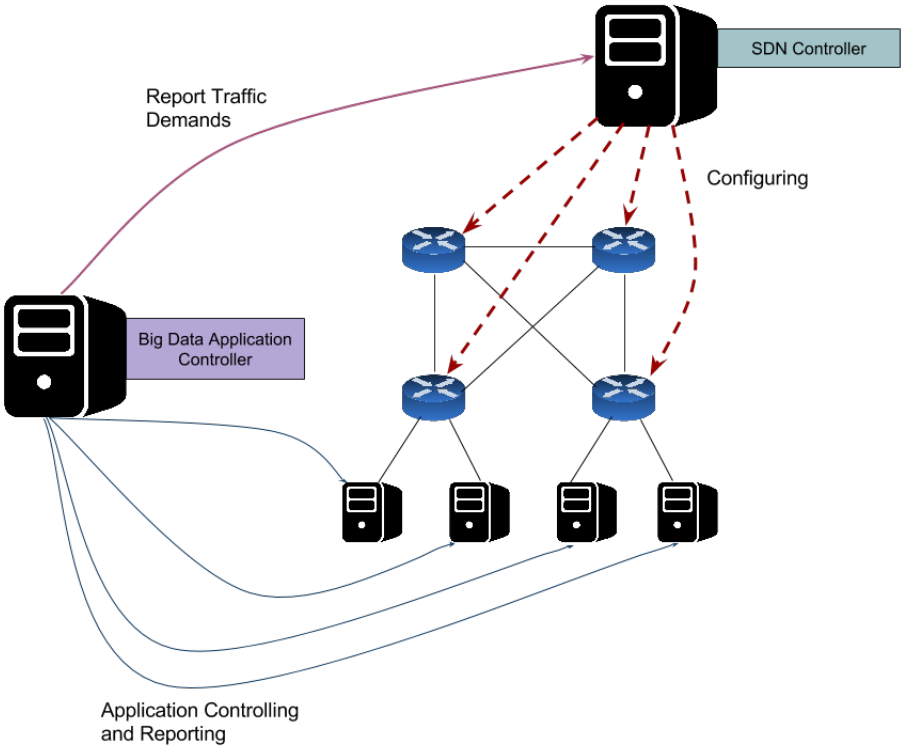


Figure 2.2: The big data application controller reports traffic demands to the network controller, enabling *Application-Aware* flow scheduling.

plication in advance and dynamically install flows in the network switches based on its prediction. Das *et al.* argued that studies monitoring the network to detect changes such as *Hedera* [10], do so only after a change has occurred; moreover network monitoring is an expensive operation. In order to anticipate network demands, Das *et al.* developed software agents which were installed on all servers running Hadoop in a cluster. The Software agents essentially performed two functions, namely, scanning Hadoop logs to extract information about map tasks and network transfers; and sending this information to the scheduling component of *FlowComb*, which is the SDN controller, thereby making the SDN controller *application-aware*.

Similar to Hedera [10] as described in 3.3.2, *FlowComb's* Scheduling engine dynamically allocates and installs paths in the network using the OpenFlow protocol [43]; in order to avoid traffic congestion and provide sufficient bandwidth to all flows. *FlowComb* was found to reduce the average running time of sorting 10 GB of data in a 14 node

Hadoop cluster by *33%* [21].

*FlowComb* has a number of drawbacks, such as it depends on domain-specific knowledge to operate, hence the same implementation of *FlowComb* cannot be applied to different MapReduce implementations apart from Hadoop [31], such as Dryad [35] and Spark [53]. Furthermore, since reducers start transfers at random, *FlowComb* cannot determine the exact timing of the start of a transfer [21]; extracting information at the hosts by scanning Hadoop logs causes a computation overhead and finally, *FlowComb* helps in optimizing Hadoop jobs only when the network is congested.

Similarly, Wang *et al.* [56] have proposed to tightly integrate Big Data applications with network control by leveraging the programmability of networks provided by SDN and the high network bandwidth provided by optically switched networks, making the network *application-aware*. Since applications such as Hadoop [31] have a master node that maintains an overall control over the other nodes in a cluster, as described in 3.1.2, Wang *et al.* suggest to interface the SDN controller with the master node, thereby enabling application controllers to report traffic demands to the SDN controller and issue topology configuration commands to the SDN controller, on the basis of which, the network is configured *reactively*.

The approach suggested by Wang *et al.* [56] would have to overcome a number of challenges. A demand estimator engine would need to be implemented for every application that has to be interfaced with the SDN controller for the master node, which would introduce a significant computational overhead for the master node [56]. The *reactive* nature of configuring the topology at runtime, as suggested by Wang *et al.* imposes significant computational responsibilities on the SDN controller to re-configure the network in response to traffic demands with low latency, and imposes a challenge on the SDN controller's scalability.

Wette *et al.* [57] argue that *reactive* approaches such as Hedera [10], which try to optimize data centre networks by reactively installing flow entries, overwhelm the switching hardware by the number of flows that they tend to install. In order to alleviate

overwhelming of switching hardware, Wette *et al.* propose a *semi-proactive* approach of flow scheduling called *HybridTE*. By exploiting explicit knowledge about elephant flows, *HybridTE* is able to perform flow scheduling using very few flow entries in the switches. *HybridTE* treats mice flows and elephant flows differently, by installing flow entries *proactively* for mice flows and *reactively* for elephant flows.

Using a data centre architecture as described in 3.2, *HybridTE* installs one forwarding tree per ToR switches using OpenFlow wild-card flow table entries in a *proactive* manner, which, Wette *et al.* argue is sufficient for small flows. As far as the elephant flows are concerned, Wette *et al.* investigate the effect of different techniques for elephant flow detection such as HadoopWatch [51], which monitor Hadoop logs to predict upcoming file transfers and packet sampling, which has been demonstrated by Choi *et al.* [16] to be effective in determining elephant flows when using a reasonable sample. Using elephant flow detection schemes as previously described, *HybridTE* re-routes elephant flows to the shortest path which is the least congested, in a *reactive* manner.

*HybridTE* was found to outperform ECMP [33] and Hedera [10] in terms of flow completion times, with ECMP taking 29.1% longer times for flow completion in a network with increasing congestion than *HybridTE* [57]. However, *HybridTE* has certain limitations such as it does not exploit path diversity for routing mice flows, and detecting elephant flows to avoid congestion has inherent high-level latencies.

*Pythia* [48] is another study that aims to optimize Hadoop job completion times by predicting Hadoop data transfers at runtime and avoiding congestion during the Hadoop shuffle phase by dynamically routing network flows. *Pythia* has two components

- Hadoop instrumentation middle-ware that runs on all of the Hadoop worker nodes, responsible for predicting future shuffle transfers by extracting information from Hadoop logs, similar to software agents employed by FlowComb [21].

- An Orchestration entity that dynamically allocates reactive paths on runtime on the basis of future shuffle transfers predicted by the middle-ware which is sent to it.

The Hadoop instrumentation middle-ware is able to determine the size of a data transfer, since after a map task finishes, it writes the intermediate <key, value> pairs to disk [45], which is read by the instrumentation middle-ware [48]. The size of the future transfer, along with the map task ID is subsequently transmitted to the orchestration controller, which *reactively* configures the network based on the supplied prediction.

Neves *et al.* evaluated *Pythia* on a cluster of 10 servers with a total RAM of 128GB and found that *Pythia* outperforms ECMP which is the industry standard for multipath routing, as explained in 3.3.1; by lowering Hadoop job completion times by 43% [48]. However, *Pythia* has certain limitations, such as the Hadoop instrumentation layer is able to access Hadoop logs, making it intrusive and difficult to implement in a multi-tenant data centre, with different Hadoop jobs running simultaneously causing a computational overhead at the worker nodes. *Pythia* is very specific to Hadoop and would not function for a broader range of communication patterns.

### 2.2.2 Traffic-Aware Network Optimization

In contrast to *application-aware* network optimization, as discussed in 2.2.1, *traffic-aware* network optimization seeks to optimize the network by continuously monitoring forwarding devices and reporting the changing traffic information to the SDN controller, as illustrated in Figure 2.3, which subsequently performs flow scheduling on the bases of the reported information. This monitoring can be done via the SDN controller itself, since it can request traffic statistics from OpenFlow switches.

*Hedera* [10] is one such study that employs *traffic-aware* scheduling. The study proposes an extension to ECMP, whereby, it routes flows exceeding a certain threshold in a *reactive* manner. We do a more comprehensive discussion on Hedera's Global First-Fit flow scheduling algorithm in 3.3.2 and evaluate implementations of ECMP and Global First-Fit against our *proactive* approach in Chapter 6. *Hedera* was found to significantly reduce Hadoop shuffle transfer times in comparison to ECMP routing, achieving 39%

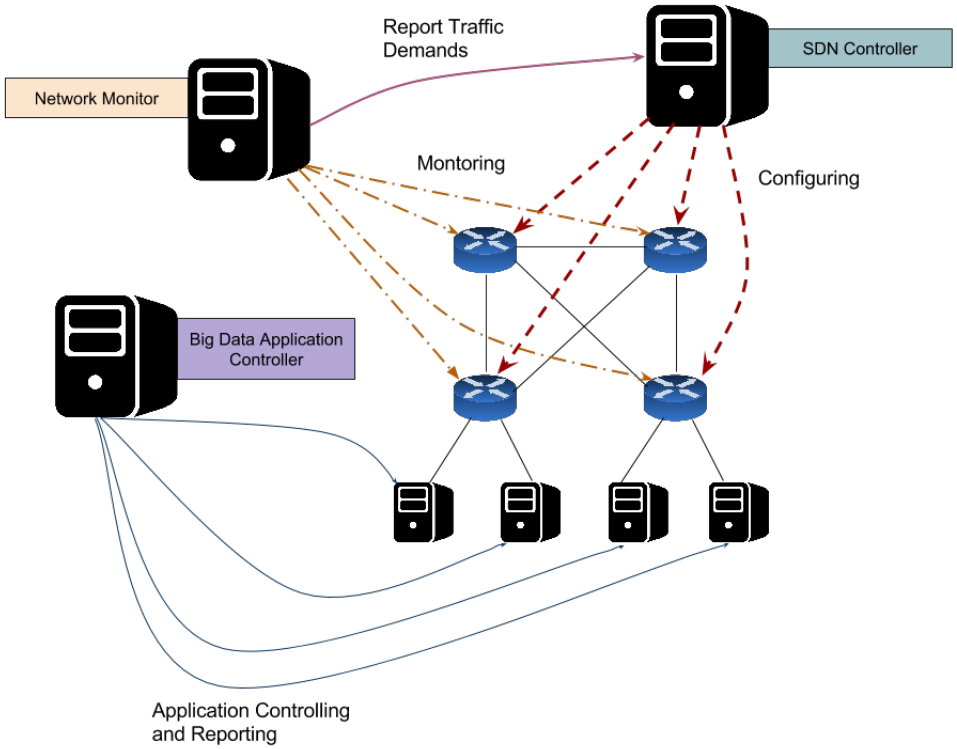


Figure 2.3: The network is monitored continuously to keep track of traffic demands, enabling *Traffic-Aware Scheduling*.

more of the total bisection bandwidth available in the network, as compared to ECMP.

## 2.3 Emulators for Data Centre Experimentation

In this section, we describe the emulators that we used for the implementation of our current project, detailed in Chapter 5, since we did not have access to a cluster of servers for running our experiments. Firstly, in 2.3.1, we describe the emulator used for emulating data centre networks and subsequently, in 2.3.2 we describe an emulator that emulates the working of Hadoop [31] using real Hadoop traces.

### 2.3.1 Data centre Network Emulation

In order to exploit the full potential of SDN as described in 2.1, it should be possible to prototype novel approaches for network control without the need of running experiments

in a data centre, since all researchers do not have access to such resources. Lantz *et al.* [38] filled this gap by introducing *Mininet*, which is a lightweight virtualization platform for prototyping SDN Networks. *Mininet* can work on the constraint resources of a single laptop. It is built with the following attributes

- Allows defining of new topologies and functionality using familiar programming languages, specifically, python [55].
- Network can be deployed from virtual to physical hardware without any change in code, with high degree of fidelity in behaviour.
- Allows managing the network in real time, scaling to thousands of switches on a single physical host.

Code created in other similar emulators like Opnet [3] and ns-2 [1] for network simulation cannot be ported directly to real hardware; moreover, they don't provide the functionality of interacting with the network in real time [38]. On the other hand, *Mininet* leverages Linux features such as network namespaces and virtual Ethernet pairs, enabling it to offer support for networks with bandwidth in Gigabits and hundreds of nodes such as controllers, switches and hosts; all of which are emulated with ease on a single laptop. All these attributes made *Mininet* the ideal choice of network emulator for our current project since we did not have access to a cluster of servers.

*Mininet* combines its lightweight Linux virtualization with an interactive command line interface and an extensible API written in python to experiment with SDN. The interactive CLI can be used to manage the entire SDN network from a single command line while a simulation is running. The *Mininet* python API can be used to define custom network topologies, node types and experiments [38].

*Mininet* uses Open vSwitch [2] for emulating OpenFlow switches in the network, and works well with all SDN controllers such as POX/NOX [28], OpenDayLight [24], *et cetera.* A *Mininet* host is essentially a shell process which resides in it's own namespace [38],

having virtual Ethernet interfaces, making it very lightweight. An SDN controller can work with *Mininet* as long as the switches running in mininet have IP connectivity to the controller [38].

*Mininet* scales well to over 1000 nodes owing to sharing of the file system, process ID space, kernel and other resources, providing a bandwidth of 1-3 Gbps through a single switch [38], making it ideal for running emulations of a data centre network on a single laptop. However, *Mininet* has certain limitations, such as it suffers from a lack of performance fidelity when the emulation has high loads, additionally, software based forwarding cannot surpass the speeds obtained by TCAM accelerated vendor switches, live host migration is not supported and *Mininet* does not support distributed emulation since it can only run on a single machine. Nonetheless, *Mininet* offers a viable alternative to running experiments in real hardware and provides a scalable and interactive environment with an extensible API to conduct experiments, which is why we chose it for implementing our experiment as described in Chapter 5.

### 2.3.2 Hadoop Emulation

Use of real hardware robust enough for running data centre experiments is not a valid option for many researchers since such resources are not readily available. We faced a similar problem, therefore we chose to run the experiments of this project on a Hadoop emulator-based test bed.

The emulator that met our requirements was MRemu, devised by Neves *et al.* [47]. MRemu is a framework that works as a Hadoop emulator by reproducing traffic patterns of a Hadoop job trace with the same latencies as in the original Hadoop job trace. Hence, using MRemu, Hadoop jobs are emulated in hosts running on Linux containers [47] such as *Mininet*, as described in 2.3.1 to emulate data centre nodes, running on a single physical host, which have low IO and CPU resources to run a real Hadoop job.

Studies have demonstrated that MapReduce applications are sensitive to performance

of the network [21, 18], nonetheless, numerous studies use synthetic traffic patterns, generated following certain probabilistic distributions [9, 26, 20], which fails to capture the true network workloads [47]. Taking this into consideration, Neves *et al.* developed a tool for extracting traces from Hadoop job execution logs with enriched network information to generate comprehensive Hadoop job traces to be used with MRemu. These traces [46] produced by Neves *et al.*, made available online, have been employed by us in this project.

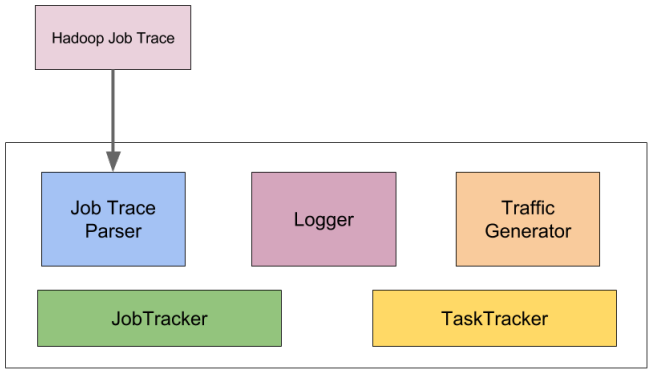


Figure 2.4: Block Diagram of MRemu Hadoop Emulator.

Figure 2.4 shows a high level overview of the Hadoop emulator in MRemu. Hadoop job traces are fed to the Job Trace parser, which uses information from the trace, such as wait times and task durations, to mimic the latencies in a real Hadoop job execution. The traffic generator generates *iperf* flows, mimicking network transfers of the real Hadoop job corresponding to the trace, while the logger logs Hadoop events to local disk of the emulated nodes.

MRemu makes it possible to test different network routing control logic using SDN for Hadoop traffic on the constraints of a single physical system, making it ideal to be used in conjugation with *Mininet*, as described in 2.3.1, for Hadoop traffic scheduling experimentation. MRemu supports production SDN controllers such as POX/NOX [28] and OpenDayLight [24], making it ideal for evaluating novel approaches to route Hadoop traffic in order to accelerate it.

Neves *et al.* [47] tested MRemu, by first obtaining Hadoop traces from a real cluster of 16 identical nodes, running Hadoop 1.1.2, while the emulation was performed on a single

node with 16 GB of RAM and 16 x86 64 cores, running on *Mininet* 2.1.0. The original Hadoop jobs ran applications forming part of the HiBench Benchmark Suite [34] such as Bayes, Sort, Nutch and PageRank. On performing a comparison between job completion times in real hardware and the MRemu emulation setup, Neves *et al.* observed that the job completion times were comparable. Furthermore, Neves *et al.* evaluated individual flow completion times as well and found the transfer durations in the emulation to be slightly different than the real transfer durations, owing to inaccuracies in Hadoop logs (used to extract traces) due to high-level latencies. Nonetheless, Job Completion times were found to be near accurate, owing to which, we chose to use MRemu for evaluating our proactive approach as described in Section 5.

However, MRemu has certain limitations, such as

- MRemu supports emulation of only one job at a time, having no support of concurrent job execution.
- MRemu does not model failure handling techniques of Hadoop, making it a requirement of only running successful job traces for emulation [47].
- MRemu does not support distributed emulation, hence it can only support a limited number of Linux container nodes that can be supported by a single physical server and cannot scale to hundreds of nodes.

Nonetheless, MRemu adequately met our requirements for a Hadoop emulator that can be run over *Mininet*, therefore, we chose to use it for the current project in conjugation with *Mininet*.

# Chapter 3

# Background

In this chapter, we describe the research which led to the current project. In Section 3.1, we provide an overview of the MapReduce framework and subsequently the Hadoop MapReduce implementation [31].

Furthermore, in Section 3.2 we describe the current state-of-the art in data centre architecture design and focus on *fat-tree* [9] data centre topology in 3.2.1, which is used in our experiment as the network topology because of its ability to alleviate inter-node communication bandwidth bottlenecks in large-scale clusters.

Finally, in Section 3.3 we describe the current state-of-the-art *flow scheduling* mechanisms for in data centre networks, namely Equal Cost Multi-Path Routing (ECMP [33]) and Global First-Fit [10], implementations of which are used in our experiment to compare against proactive measures of flow scheduling.

## 3.1 MapReduce

MapReduce was introduced by Dean *et al.* [22] as a programming abstraction, which enables distributed cluster computing with commodity servers and is highly fault tolerant and reliable. MapReduce is one of the most frequently used framework for big data processing [11] and we chose it for our project due to its ubiquity and popularity. It

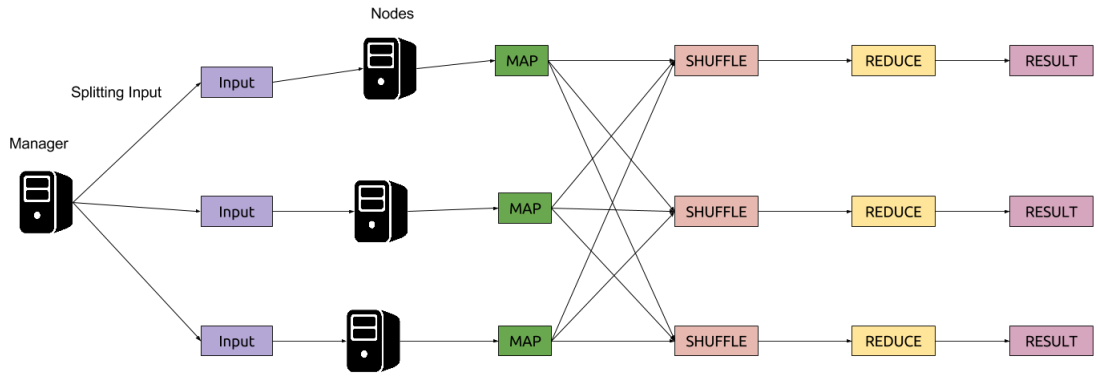


Figure 3.1: High level overview of the MapReduce Framework.

broadly consists of 3 phases, namely the *map*, *shuffle* and *reduce* phases as illustrated in Figure 3.1.

MapReduce is a computational model based on *key-value* pairs. The *Map* function produces a set of intermediate *key-value* pairs for a given input [22]. Subsequently, the intermediate data is grouped according to its keys in the *shuffle* phase. Finally, in the *reduce* phase, all values for a given key are combined to produce the final result. The data used for MapReduce jobs is usually stored in a distributed file system (DFS), which takes care of fault tolerance by replicating the data across the cluster. Examples of such data stores include GFS [25] and HDFS [13].

### 3.1.1 High Level Execution Overview

The input data set of a MapReduce job is first partitioned into **M** pieces, which are processed in parallel by different machines [22]. The Reduce function is invoked by partitioning the intermediate key set of the data into a set of **R** pieces, which is distributed across the entire cluster for computation.

MapReduce computation proceeds in the following steps

- The input file is first split into **M** pieces while MapReduce starts up on the entire

cluster of machines.

- One of the machines in the cluster is designated as the *master* node, while all the other nodes in the network are *worker* nodes which are assigned jobs by the *master* node. The master assigns a map task or a reduce task to each one of the idle worker nodes in the network from a total of **M** map tasks and **R** reduce tasks.

- The corresponding partitioned input is read by the worker which is assigned a map task, it parses the input and emits key-value pairs and passes each of those to the map function. Intermediate key-value pairs emitted by the map function are buffered in memory.

- The buffered pairs are written into persistent memory store of the worker nodes on a periodic basis and are partitioned into **R** pairs. Memory locations of these key-value pairs are sent to the master node so that they can be forwarded to the reduce worker nodes [22].

- On notification about these locations, the reduce worker reads the buffered data from the persistent memory store of the map workers via RPC calls [22]. Subsequently, it sorts the intermediate keys of the data by grouping the same keys together, this is also called the shuffling phase.

- The sorted intermediate data is iterated upon by the reduce worker nodes and for each unique key encountered, it passes the corresponding key-value pairs to the reduce function.

After all the above steps culminate, the output consists of **R** output files. These files may or may not be merged into a single output file, depending on the MapReduce implementation.

### 3.1.2 Hadoop Overview

Hadoop is one of the most frequently used open source MapReduce implementations. We choose Hadoop as the MapReduce implementation of choice for this project due to it being open source and its widespread usage for Big Data analysis. It broadly consists of *two* main components

- Hadoop MapReduce - Open-source implementation of the MapReduce computation model.
- HDFS (Hadoop Distributed File System) - A resilient, fault tolerant and distributed file system that provides high throughput access to application data and is designed to be used with commodity hardware [13].

Similar to the Google MapReduce model [22] outlined in Section 3.1, HDFS too has a master-slave architecture [13]. As illustrated in Figure 3.2 a Hadoop cluster consists of the following

- HDFS layer consists of two types of nodes responsible for managing the Distributed file system
  - **NameNode** - The entire Hadoop cluster contains one *NameNode* which serves as the master server, managing the file system access of worker nodes and the file system name space. It also instructs the *DataNodes* (slaves) in the cluster to create, delete and replicate data blocks.
  - **DataNode** - There is a usually one *DataNode* on each node in the cluster, which is responsible for managing the persistent storage attached to that node in the cluster. File system read and write requests are also processed by the *DataNodes*.
- MapReduce layer consists of two types of nodes that control the execution of MR jobs [58]

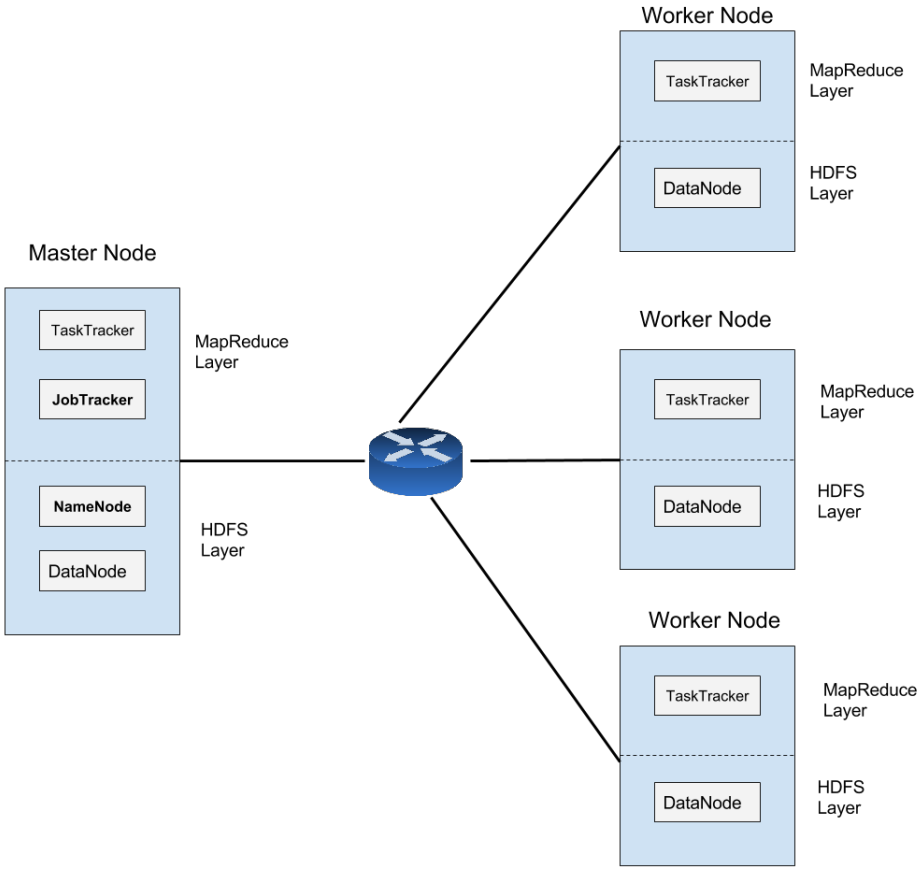


Figure 3.2: Basic Hadoop Architecture with the master node housing the JobTracker and the NameNode functionalities of Hadoop, while the worker (slave) nodes housing the TaskTracker and DataNode functionalities of Hadoop.

- **JobTracker** - Similar to the *NameNode*, there is one *JobTracker* Node in a Hadoop cluster, housed in the master node, which is responsible for scheduling all the jobs of the system to be run on the *TaskTracker* (worker) nodes [58]. It keeps track of the progress of every job, rescheduling it to other *TaskTracker* nodes in case of failure [58].
- **TaskTracker** - *TaskTrackers* or worker nodes, run the MR jobs assigned to them by the *JobTracker* node and report the progress back to the *Jobtracker* [58] node.

Most of the data transfer loads in a Hadoop cluster are attributed to the shuffling phase, when intermediate data from the mappers is shuffled over to the reducer nodes, as

outlined in subsection 3.1.1.

Chowdhury *et al.* [18] analysed Hadoop traces from Facebook's Hadoop cluster and found that on an average, 33% of the runtime is consumed by the shuffle phase. Additionally, in 26% of the tasks with reduce jobs, more than 50% of the runtime is spent in the shuffle phase, while it accounts for upwards of 70% of the runtime in 16% of the jobs [18], confirming results reported in literature [10, 26, 30] which state that network is a bottleneck in MapReduce. Therefore, with this project, we aim to make the network *application-aware* to alleviate loss in performance of MapReduce due to network bottlenecks.

## 3.2 State-of-the-art in Data Centre Architectures

Distributed big data computational frameworks such as MapReduce [22], Dryad [35] and Hadoop [31] leverage enormous clusters made up of commodity CPUs for their compute prowess. Various recent studies [12, 15, 23, 25, 26] have determined that networks are bottlenecks in data centres running distributed big data application frameworks. In order to scale the number of hosts in a cluster running these distributed big data application frameworks, multiple paths are required between source and destination hosts [9, 26, 27, 30, 29], which has heavily influenced data centre network design.

Recent data centre architectures advocate horizontal scaling of hosts instead of vertical to overcome limited port densities in commercial switches in order to support the communication patterns of big data applications [9, 26, 27]; therefore such architectures take advantage of the availability of a large number of parallel paths between any two source and destination switches, and are known as *multi-rooted tree* topologies [10]. Multi-rooted tree topologies consist of two or three level trees of switches [9, 10] with higher speed links, while the aggregate bandwidth decreases while moving higher up in the multi-rooted tree topology [32].

At the leaf of a multi-rooted tree topology, there are a number of GigE ports (48-288)

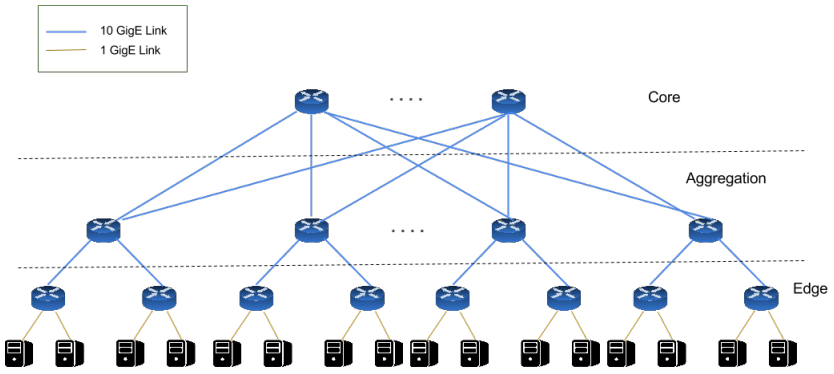


Figure 3.3: Common Multi-Rooted Data Centre Architecture with 10 GigE and 1 GigE links.

with 10 GigE uplinks to one or more switches in the *aggregation layer* [9] as illustrated in Figure 3.3. Moving up in the tree hierarchy, switches have 10 GigE ports (32-128), capable of switching significant amounts of traffic between the edges [9].

Al-Fares *et al.* [9] advocate the use of a *fat-tree topology* [39] for data centre network design and showed that full aggregate bisection bandwidth can be achieved using commodity Ethernet switches of data centre clusters with tens of thousands of hosts. Fat-tree topology aims to alleviate shortcomings in current data centre network design such as

- over-subscription of links higher up in the topology, where over-subscription is defined by Al-Fares *et al.* [9] to be the ratio of worst-case aggregation bandwidth which can be achieved by the end hosts to the total bisection bandwidth available in the network; where typically, data centre designs are oversubscribed by a ratio of 2.5:1 (400 Mbps) to 8:1 (125 Mbps) with 1 Gb/s commodity Ethernet switches [32] and,

- the cost of building a network with an over-subscription ratio of 1:1 is quite substantial, with each of the 48-port GigE switch at the edge costing around $7000, while the 128-port switches at the aggregation and core layer cost around $700,000 [9],

by providing backward compatibility to hosts running Ethernet and IP, scaling economically using commodity Ethernet switches for data centre design and providing a

scalable interconnection bandwidth enabling any host to communicate with any other host in the network at its full local network interface bandwidth [9].

### 3.2.1 Fat-Tree Topology

*Fat-tree* topology [39] is a special instance of a *Clos* topology, which was introduced in the 1950s to deliver high levels of bandwidth in telephone networks by interconnecting smaller commodity switches [19]. It is organised into $k$-pods, where each pod contains of two layers of $k/2$ switches [9]. Each of the $k/2$ hosts in the lower layer are connected directly to each of the $k$ port switches in the lower layer of the pod. The remaining $k/2$ ports of each of the switches in the lower layer are connected to $k/2$ ports of the total $k$ ports up the hierarchy in the aggregation layer.

The core switches are $k/2^2$ in number, with each of the core switch having one port connected to each of the $k$ pods . Consecutive ports in the aggregation layer of each pod switch are connected to core switches on $k/2$ strides in such a manner that the ith port of any core switch is connected to pod i. A fat-tree topology built with k-port switches supports $k^3/4$ hosts.

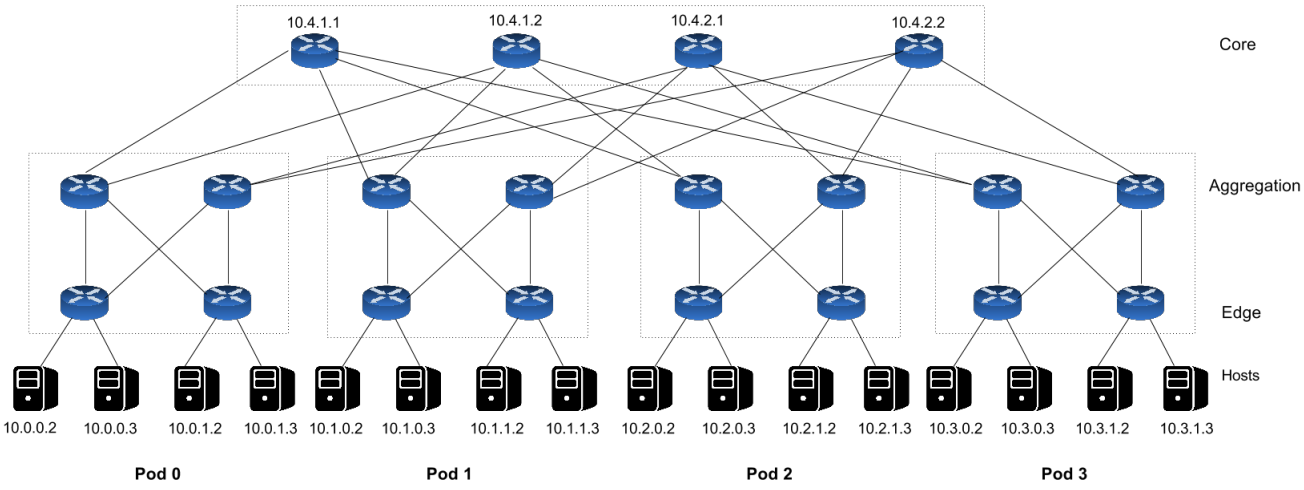


Figure 3.4: Fat-tree topology with 4 pods having 16 hosts in total (k=4).

In this project, we use a fat-tree topology with $k = 4$ for our experiments as described

in Chapter 4. Figure 3.4 illustrates a $k$-ary fat-tree topology with $k = 4$, which is the topology we employ for our experiments. It consists of $k^3/4$ i.e. 16 host machines with 4 pods. Hosts connected to the same lower level switch form a subnet; hence, all traffic between two hosts in the same subnet is switched while all the other traffic is routed.

In order to achieve maximum bisection bandwidth in a fat-tree network, outgoing traffic from a pod needs to be spread evenly amongst the core switches. There is a need for the core switches to be able to recognize and provide special behaviour to the traffic classes that need even spreading since

- Routing protocols such as OSPF2 [44] take hop-count as a metric of shortest path, causing switches to concentrate traffic going to a subnet to a single port even though there are $(k/2)^2$ paths in the network with the same cost, thereby under utilizing the path diversity in the network and causing severe congestion at these points [9].

- Extensions to OSPF2 such as OSPF-ECMP [54] cannot be used on commodity Ethernet switches and require an overwhelmingly large number of prefixes.

To alleviate these shortcomings, Al-Fares *et al.*, devised an addressing scheme that allocates all IP addresses in the network within the private 10.0.0.0/8 block.

Addressing in a *fat-tree* topology follows the following pattern

- Switches in the pod are allocated IP addresses in the form 10.*pod.switch*.1, where *pod* belongs in the range [0, $k$-1], while the *switch* denotes the position of the switch in the pod

- Core switches are allocated addresses in the form 10.*k.j.i*, where $j$ and $i$ denote the co-ordinates of the core switch in the $(k/2)^2$ grid

- Hosts have addresses of the form 10.*pod.switch.I D*, where $I\ D$ denotes the host's position in its subnet

This addressing scheme helps in building a two-level routing scheme and scales to 4.2M hosts. The two-level routing scheme enables even spread of traffic across the network [9]. Two-level lookup is implemented in hardware using Ternary Contet-Accesible Memory (TCAM).

Due to all these optimizations, Al-Fares *et al.* [9] found that in a fat-tree network of 16 hosts, the two-level switches achieve approx. 75% of the aggregate ideal bisection bandwidth. For their benchmark suite, Al-Fares *et al.* leveraged dynamic flow allocation strategies available in certain routers and found that their flow classifiers performed significantly better than traditional tree topologies which achieve only 28% of the ideal bandwidth, with the worst-case aggregate bisection bandwidth achieved by the network to be 75% of the ideal. In light of the results achieved by Al. Fares *et al.*, we choose *fat-tree* topology as the topology of choice for running our simulations described in Chapter 4.

## 3.3 Flow Scheduling in Data Centre Networks

To take advantage of topologies such as *fat-tree*, described in 3.2.1, which have multiple paths between the same host-destination pair. In this section, we describe current state-of-the art protocols such as Equal-Cost Multi-Path (ECMP) [33] and Hedera [10] for Multi-Path flow scheduling. These two flow scheduling techniques along with our *proactive* approach are employed in the design of our experiment as described in Chapter 4 and subsequently evaluated against each other in Chapter 6.

### 3.3.1 Equal Cost Multi-Path Routing

Switches that support ECMP are configured with many possible paths for the same host-destination pair. When a packet arrives at a switch, with possible multiple paths to its destination, selected fields of the packet are hashed modulo the total no of paths and the packet is forwarded along the path that corresponds to the result. This results in splitting of the load, maintaining the arrival of packets for the same flow. ECMP is supported by

most enterprise switches.

However, ECMP has a lot of limitations, such as it does not take flow bandwidth into account when it makes flow allocation decisions, which can cause bottlenecks at certain links even when the communication pattern is simplistic; this further results in collisions because the static mapping of flows to paths does not take into consideration the current network utilization which overwhelms switch buffers degrading overall performance [10]. Moreover, the growth in routing table entries is multiplicative as the number of paths grow causing increase in lookup latency and cost.

Nonetheless, ECMP implementations support 8-16 multiple paths currently, and it works for topologies such as *fat-tree*, therefore we evaluate it against other approaches, namely Global First-Fit and a proactive approach, for routing of Hadoop traffic as described in Chapter 4 for our current project.

### 3.3.2 Global First-Fit Flow Scheduling

To alleviate the shortcomings of ECMP described in the previous sub-section, Al-Fares *et al.* [10] devised a dynamic flow scheduler called *Hedera*. Hedera is essentially an extension to ECMP; it works similarly to ECMP as described in 3.3.1 for small flows, however when flows exceed a certain threshold rate, *Hedera* dynamically allocates an appropriate path to the flow after doing a demand estimation and installs the path in the appropriate switches. The flow lives only till a certain timeout after which, the flow entries are removed from the path.

The demand estimator converges to the natural flow demand by performing repeated iterations of decreasing the flow capacities at the receivers and increasing it at the sources until the capacities converge. Al-Fares *et al.* devised two algorithms for dynamic flow placement, namely Global First-Fit and Simulated Annealing. In this project, we focus only on Global First-Fit and evaluate it against other approaches of routing Hadoop traffic in a *fat-tree* topology, since it is much simpler to implement than Simulated Annealing.

When a flow exceeds the threshold rate, Global First-Fit initiates a linear search of all the possible paths that can accommodate the flow and places the flow in the *first* such path that it encounters [10], making it a *greedy* algorithm. The flows are placed by creating a reservation of the bandwidth capacity along the path and subsequently installing the flow entries in the aggregation and edge switches. Global First-Fit achieves this by maintaining a list of the reserved capacities of all the links in the network and placing flows accordingly.

Al-Fares. *et al.* ran benchmark tests to evaluate the effectiveness of Global First-Fit against ECMP routing in a test bed of 16 hosts arranged in a *fat-tree* topology with OpenFlow [43] enabled switches. They also ran tests in a simulator for measuring the scalability of their routing algorithms. Al-Fares *et al.* [10] found that Global First-Fit and Simulated Annealing significantly outperform ECMP for various communication patterns achieving achieving 39% more of the total bisection bandwidth available in the network, as compared to ECMP.

# Chapter 4

# Design

In the previous chapter, we summarised the background for the current project, more specifically by providing a description of the technologies that we use in the design of our experiment, such as Hadoop [31], Global First-Fit flow scheduling [10] and fat-tree data centre topology [9]. In this chapter, the design approach and objectives of our experiment are described in Section 4.1. Subsequently, in 4.2, we describe our *proactive* flow scheduling algorithm. In Section 4.3, a high level overview of our experiment design is provided and finally, in Section 4.4, an analysis of our experiment design is provided.

## 4.1 Approach

We propose a *proactive* flow scheduling mechanism in order to determine if there are performance gains, in terms of Hadoop [31] Job completion times and total bisection bandwidth achieved by a data centre network.

The existing flow scheduling mechanisms described in 2.2 are overwhelmingly *reactive* in nature. Therefore, our approach stands as a contrast to these existing flow scheduling mechanisms, since

- We configure the network before any commencement of traffic on the basis of previous executions.

- There is less control overhead than *reactive* approaches on the network.

Finally, by measuring our approach against existing flow scheduling mechanisms, we want to investigate if the control overhead of *reactive* approaches lowers their performance.

## 4.2 Proactive Flow Scheduling Algorithm

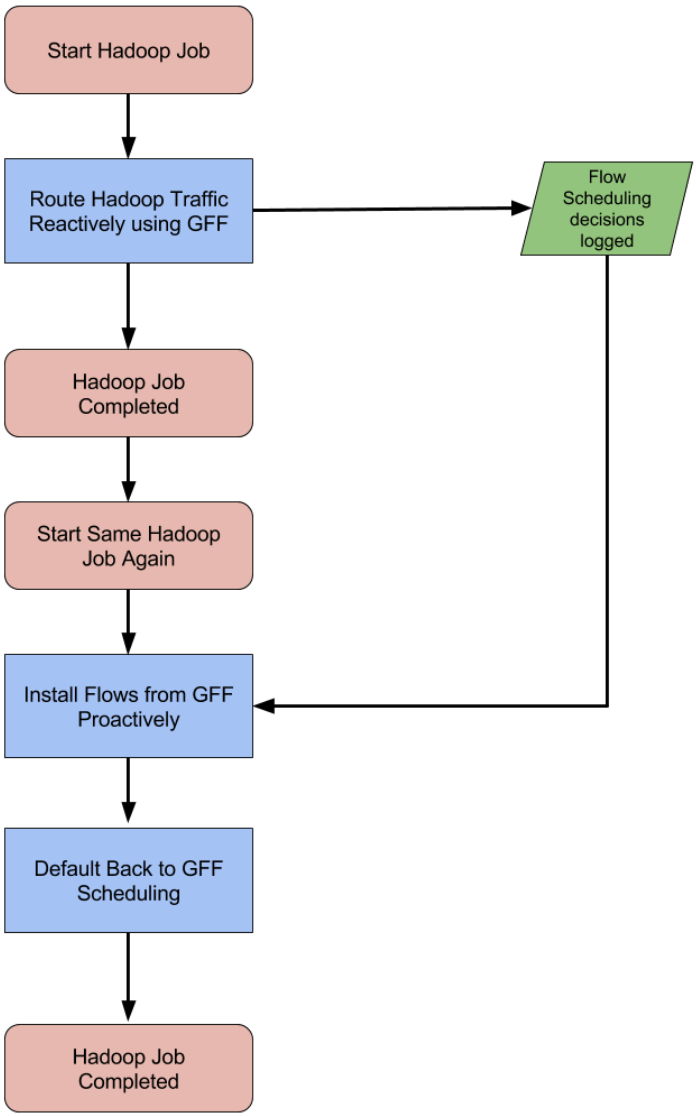


Figure 4.1: A flowchart depicting the *Proactive* Flow Scheduling Algorithm.

We device a proactive routing strategy as depicted in Figure 4.1 in the following steps

- We run a Hadoop job on a cluster of 16 hosts, arranged in a *fat-tree* [9] topology;

where all switches in the network are controlled by a centralized controller using the OpenFlow [43] protocol.

- Hadoop traffic is routed in the network using the Global First-Fit (GFF) [10] flow scheduling algorithm.

- The flow scheduling decisions made by the GFF algorithm are logged and persisted into the memory of the controller.

- Concurrently, the total bytes transmitted and received by each host in the cluster are logged into disk, which are used to calculate the total bisection bandwidth achieved by the hosts in the network.

- Subsequently, the same Hadoop Job is run again and the traffic of the network is handled by the *Proactive* controller.

- The *Proactive* controller reads the flow scheduling decisions for the same job made by the GFF algorithm earlier, and installs them statically in the network, prior to the start of Hadoop transfers.

- After installing flow entries statistically, it defaults to the GFF behaviour for routing flows in the network.

- While the Hadoop job is running, each host in the network logs the total number of bytes transmitted and received by its network interface in order to calculate the average bisection bandwidth achieved for *proactive* routing.

- The steps above are repeated iteratively for different Hadoop jobs. When all the iterations are completed, we obtain Hadoop Job completion times and total bisection bandwidth utilization for our *proactive* and GFF (*reactive*) routing strategies.

## 4.3 Architecture Overview

The High-level architectural design of our experiment is illustrated in Figure 4.2, and it broadly consists of three parts: an SDN controller, a fat-tree data centre topology and a Hadoop Emulator. A brief overview of each part is provided in this section.

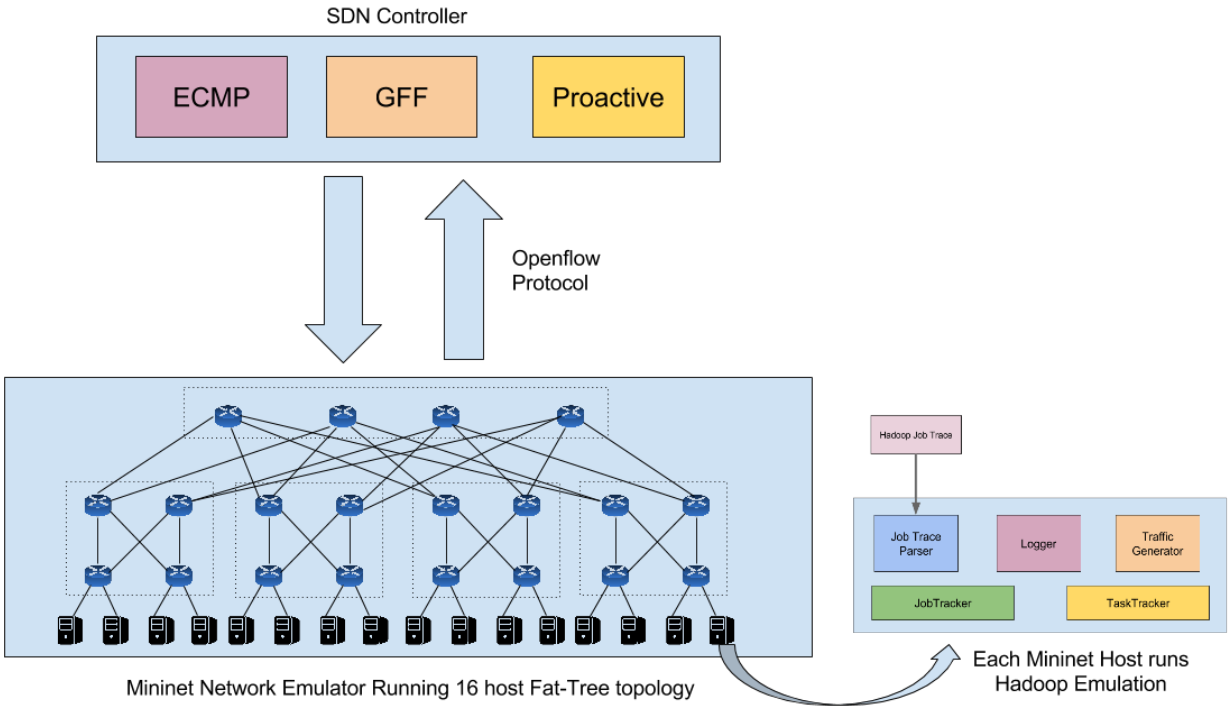


Figure 4.2: High Level Overview of the experiment Design.

### 4.3.1 SDN Controller

The SDN controller has a three-fold functionality as illustrated in Figure 4.2. It routes Hadoop application traffic with three different routing algorithms, namely, ECMP [33] which is the *de-facto* standard of routing traffic in a multi-rooted tree architecture [9], Global First-Fit [10], which is a *reactive* flow scheduling algorithm that builds upon ECMP, and finally, our *proactive* flow scheduling algorithm which leverages the *reactive* flow decisions made by GFF for a certain Hadoop job, installing them statically before communication commences in the network for the same Hadoop job.

At a time, it schedules flows using one of the three algorithms, which is specified when the SDN controller is launched. It controls the forwarding behaviour of all the switches in the fat-tree topology using the OpenFlow [43] protocol as illustrated in Figure 4.2.

### 4.3.2 Fat-tree topology

As mentioned in 2.3.1, we employ *Mininet* [38] for emulating a data centre network, since we do not have access to a 16 host cluster. Therefore, we use the *Mininet* network emulator for emulating a $k$-ary fat-tree topology (described in 3.2.1), where $k = 4$ as illustrated in Figure 4.2. All the switches in the emulation are connected to an external SDN controller via the OpenFlow protocol, while each of the 16 hosts in the *Mininet* emulator run a Hadoop emulation.

### 4.3.3 Hadoop Emulation

As illustrated in Figure 4.2, each host in the *Mininet* emulator is running a Hadoop emulation. Since, the entire fat-tree topology is run on a single machine, it is not feasible to run a full version of Hadoop [31] on the emulated hosts. Therefore, we use a lightweight alternative, by running a Hadoop emulator MRemu, on each host. As described in 2.3.2, MRemu emulates Hadoop by producing iperf flows which are based on real Hadoop Job traces. Thus, we are able to simulate the running of Hadoop jobs in a 16 host fat-tree topology, where the routing decisions of the switches are controlled by an external SDN controller.

## 4.4 Analysis

Our aim in proposing the design of this *Proactive* flow scheduling algorithm was to investigate if there are benefits in terms of better application performance and high bisection bandwidth utilization by *proactive* configuration of a data centre network.

In order to fulfil our aim, we introduced the design of our experiment in this chapter whereby we use *reactive* configurations logged from the GFF algorithm, which are fed into the *proactive* controller to be installed into the network as static flows. Automatic generation of proactive configuration for the network is beyond the scope of this project.

We find that the design set forth meets the requirements of this project, and is general enough to be applied to any Big data processing application traffic. It is a first step towards evaluating a *proactive* measure of network configuration for big data application processing, owing to which it is straightforward to implement. The next chapter describes a proof-of-concept implementation of this design in python.

# Chapter 5

# Implementation

The previous chapter described our proposed *Proactive* flow scheduling algorithm and the design of our experiment. We leverage pre-existing *reactive* flow scheduling decisions in the design of our *Proactive* scheduler. In this chapter, we describe the implementation of our experiment design that has been employed to evaluate the effectiveness of our proposed *Proactive* flow scheduler. In Section 5.1, an overall description of our design implementation is provided, outlining the working of different major components. In Section 5.2, the implementations of ECMP, GFF and our *Proactive* flow scheduling SDN controllers are described. Moreover, Section 5.3 describes the structure of Hadoop job traces and functional working of the Hadoop emulation. In Section 5.4, our implementation for measuring throughput from the hosts in the Hadoop emulation is described, and finally, Section 5.5 summarises our design implementation.

## 5.1 Implementation description

We have implemented our *Proactive* flow scheduling algorithm using the *dart* branch [7] of the POX [4] SDN controller. POX provides an extensible API written in Python, which can be used to program its controlling behaviour. As described in 2.3, we use the *Mininet* Network Emulator [38] for emulating a 16 host fat-tree data centre topology, where each

host is running a Hadoop emulation using *MRemu* [47]. The forwarding behaviour of the switches in the fat-tree network topology is controlled by the POX controller using OpenFlow [43]. We run Hadoop jobs and measure Hadoop job completion times along with the total network bandwidth utilization for three flow scheduling algorithms, namely, ECMP [33], Global First-Fit [10] and our *Proactive* flow scheduling.

Figure 5.1 illustrates the working of *LaunchExperiment*, which is the main Python script that initiates the experiment. It builds a *Mininet Network topology object*, by using a script from *Ripl-POX* [8] to initialize the switch, host and link configurations for a fat-tree network topology. *Ripl-POX* is an extension to the POX controller which provides ECMP hash based routing, as described in 3.3.1, and Mininet network configurations for data centre topologies such as *fat-tree*, described in 3.2.1.

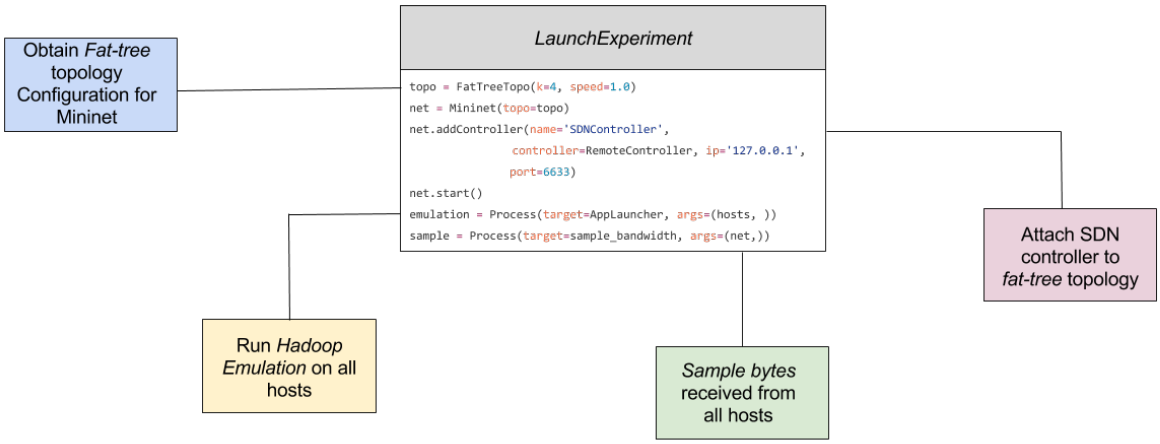


Figure 5.1: Execution of the main script that loads the different modules of the experiment.

Subsequently, a remote SDN controller is added to the Mininet network topology object by providing its IP address and port, as illustrated in the Figure 5.1, and the Mininet network emulation is launched. Finally, using Python's Multiprocessing library, two processes are launched simultaneously on each of the 16 hosts, where one of the process runs the Hadoop emulation as described in detail in Section 5.3, while the second process samples the bytes received by the host from its network interface, described in detail in Section 5.4.

*LaunchExperiment* script illustrated in the Figure 5.1 runs the experiment to measure *total bisection bandwidth* achieved and *Hadoop job completion times* when using different SDN remote controllers running on the IP address 127.0.0.1:6633, which are used for evaluating the effectiveness of our *Proactive* flow scheduling against *reactive* scheduling.

## 5.2 SDN Controller Implementation

In this section, we describe the Python-based implementation of our *Proactive* controller along with implementations of ECMP and GFF controllers using the Python API of the POX SDN controller.

### 5.2.1 Equal Cost Multi-Path Routing Implementation

As mentioned in 5.1, we used the ECMP routing implementation from *Ripl-POX*, which is an extension of the POX controller. ECMP routing is described in detail in 3.3.1. It is the *de-facto* industry standard routing mechanism for multi-path topologies [10] described in 3.2, and attempts to spread traffic evenly in a network with multiple equal cost paths by hashing selected fields of an incoming packet modulo the total number of paths available and forwarding the packet along the path that corresponds to the result.

### 5.2.2 Global First-Fit Flow Scheduling Implementation

Global First-Fit flow scheduling is an extension to ECMP routing for scheduling traffic in a topology with multiple equal cost paths between any two source and destination pairs of hosts, described in further detail in 3.3.2. It maintains link capacities of all paths in the network. When a flow needs to be scheduled, GFF performs a linear search of all possible paths that can accommodate the flow, and allocates the first path that it encounters, which meets the bandwidth demands of the flow.

The GFF controller class implementation [6] used in the experiment is illustrated in Figure 5.2. Once all the switches in the fat-tree topology are connected to the GFF

controller, the *_handle_ConnectionUp*(event) function stores the switches in a Python list with the switch Data Path IDs which uniquely identify OpenFlow switches, as keys.

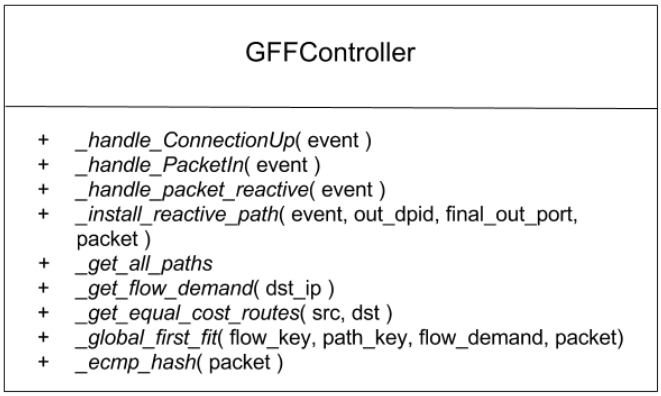


Figure 5.2: Global First-Fit Network Controller Class

Subsequently, the *_get_all_paths*() function calculates equal cost routes by calling the *_get_equal_cost_routes*(src, dst) function for all possible pairs of source and destination nodes in the network. When an OpenFlow switch gets a packet which has no matches in its forwarding table, the packet is sent to the SDN controller for processing [43]. This event is handled by the *_handle_packet_reactive*(event) function, which responds by installing a flow entry in the switch, matching the header fields of the packet *reactively*, using the GFF flow scheduling algorithm, and logs the header match fields along with the egress port into a JSON file, to be used later by the *Proactive* controller.

### 5.2.3 Proactive Controller

We implemented the *Proactive* Controller on the *hypothesis* that installing flows based on application traffic patterns in a proactive manner will reduce control overhead and high-level latencies caused due to installing of flows *reactively*. Towards this end, we log flow scheduling decisions made by the GFF controller in a JSON file as described in 5.2.2, for a particular Hadoop job. The same Hadoop job is run again and the logged flows from the previous execution are subsequently fed into the *Proactive* controller, which installs them as soon as all the switches in the network are connected to it. Finally, for every

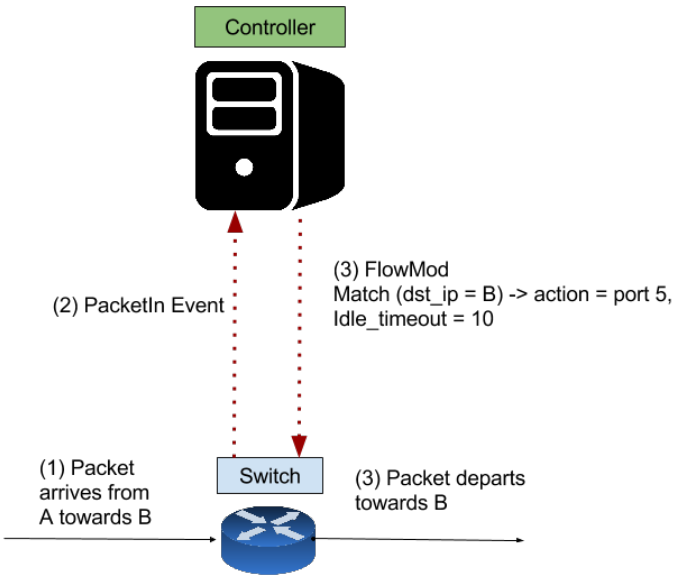


Figure 5.3: Handling of *packetIn* event on forwarding table miss by SDN controller using OpenFlow.

*packetIn* event *i.e.*, when a packet is sent to the controller by a switch since its header fields do not match any entry in the switch's forwarding table as illustrated in Figure 5.3, the controller procecsses the packet, and based on the control logic running in the controller, it creates *FlowMod* message, with an action specifying the egress port of all packets matching specific header fields. Subsequently, all packets matching the field as specified in the *FlowMod* message are routed to same egress port by the switch till the entry times out. Similarly, after all the *Proactive* entries time out, the proactive controller behaves like the GFF controller, by routing the packets *reactively* on *packetIn* events.

Figure 5.4 illustrates the implementation of the *Proactive* controller class using POX controller's Python API. Similar to the GFF controller class described in 5.2.2, the *_handle_ConnectionUp*(event) function stores all switches of the network connected to the controller corresponding to their data path IDs, in a Python list. Additionally, it calls the *_install_proactive_path*() function.

The *_install_proactive_path*() function reads all the flow scheduling decisions made by the GFF controller, and installs the same in the appropriate switches. One of the sample flow scheduling decision logged by the GFF controller is illustrated in Figure 5.5.

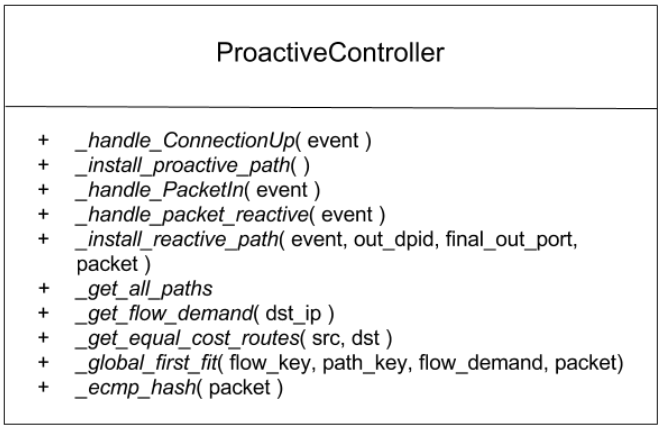


Figure 5.4: Proactive Network Controller Class


```
{
        "dl_type": 2048,
        "nw_dst": "10.0.0.2",
        "tp_dst": 6666,
        "dpid": 1,
        "tp_src": 48383,
        "dl_dst": "00:00:00:00:00:02",
        "dl_vlan": 65535,
        "nw_src": "10.0.0.3",
        "dl_src": "00:00:00:00:00:03",
        "out_port": 2
}
```


Figure 5.5: Example of a flow decision obtained from GFF flow scheduler which is installed by the *Proactive* network controller.

Notable fields in the log of a flow scheduling decision include *nw_src*, *nw_dst*, *dl_src*, *dl_dst*, *dpid* and *out_port* which are the source IP address, destination IP address, source Ethernet address, destination Ethernet address, switch data path ID and the output port respectively. All such logged flow decisions are installed by the *Proactive* controller in the switches along with an *IDLE_TIMEOUT*, so that switch routing tables are not overwhelmed. Once the flow entries expire, traffic is routed in a *reactive* manner, as described in 5.2.2.

## 5.3 Hadoop Emulation Implementation

Since it is not feasible to run a full version of Hadoop on emulated hosts, running on a single machine, due to memory and I/O constraints; therefore, we use a Hadoop emulator MRemu [47], as discussed in detail in 2.3.2. It uses traces of real Hadoop Jobs to emulate Hadoop traffic patterns in an emulation running on a single machine, without the need of a real Hadoop cluster. In this section, we describe the structure of Hadoop application traces used in the experiment, and briefly discuss the functional workflow of the Hadoop emulation.

### 5.3.1 Hadoop Job Traces

The Hadoop emulator generates traffic patterns in the network on the basis of Hadoop job traces, as discussed in detail in 2.3.2. The MapReduce applications used to obtain these traces are discussed at length in 6.2. Information available in the Hadoop job traces is stored in JSON format. Figure 5.6 illustrates one element each from the *transfers* and *tasks* JSON arrays in a Hadoop trace. The elements of the *transfer* array provide information about the network transfers and their durations, between two hosts.

This information is used to generate *iperf* flows between the source and destination hosts for the intended time duration, thereby simulating transfers of the real Hadoop job. Similarly, the *task* array provides information pertaining to the different tasks assigned to the hosts in the network and their durations, which are simulated by the corresponding hosts.

### 5.3.2 Functional Architecture

*LoadExperiment* script executes the Hadoop emulation on all hosts in the network by launching the *AppLauncher* class, as illustrated in Figure 5.1. Functional working of the Hadoop emulation on each host in the network is illustrated in Figure 5.7. The *AppLauncher* class executes Hadoop emulation on each host by obtaining information

```
{
        "transfers": [{
                "dstAddress": "172.27.102.16",
                "dstPort": 59861,
                "duration": 2.140714136,
                "finishTime": 1388441782.972,
                "mapper": "attempt_201312301708_0016_m_000014_0",
                "reducer": "attempt_201312301708_0016_r_000003_0",
                "size": 62584054,
                "srcAddress": "172.27.102.1",
                "srcPort": 50060,
                "startTime": 1388441780.831286
        }],
        "tasks": [{
                "finishTime": 1388441795.897,
                "host": "172.27.102.1",
                "name": "attempt_201312301708_0016_r_000002_0",
                "processingTime": 6.937,
                "shuffleFinished": 1388441788.953,
                "sortFinished": 1388441788.96,
                "sortingTime": 0.007,
                "startTime": 1388441772.748,
                "type": "REDUCE",
                "waitFinished": 1388441778.417361,
                "waitingTime": 5.669361
        }]
}
```


Figure 5.6: Example of a Hadoop job trace showing one element each from the transfers and tasks JSON arrays.

about the type of host, *i. e.* if the host is the JobTracker or one of the TaskTrackers from the *TraceParser* class. If the host's IP address corresponds to the JobTracker IP address in the Hadoop trace file, then it assumes the role of the JobTracker *i. e.* the master node in the network, while all other hosts in the network assume the role of TaskTrackers.

TaskTrackers simulate the working of a Hadoop job by creating *iperf* flows with the same durations as the actual transfer times, which are obtained from the *TraceParser* class. Realistic latencies caused while running Map/Reduce tasks in Hadoop are also emulated by the TaskTrackers. Similarly, information about the network traffic generated by the actual JobTracker is used to generate *iperf* flows by the emulated JobTracker. Debug information from the emulated JobTracker and TaskTrackers about network events

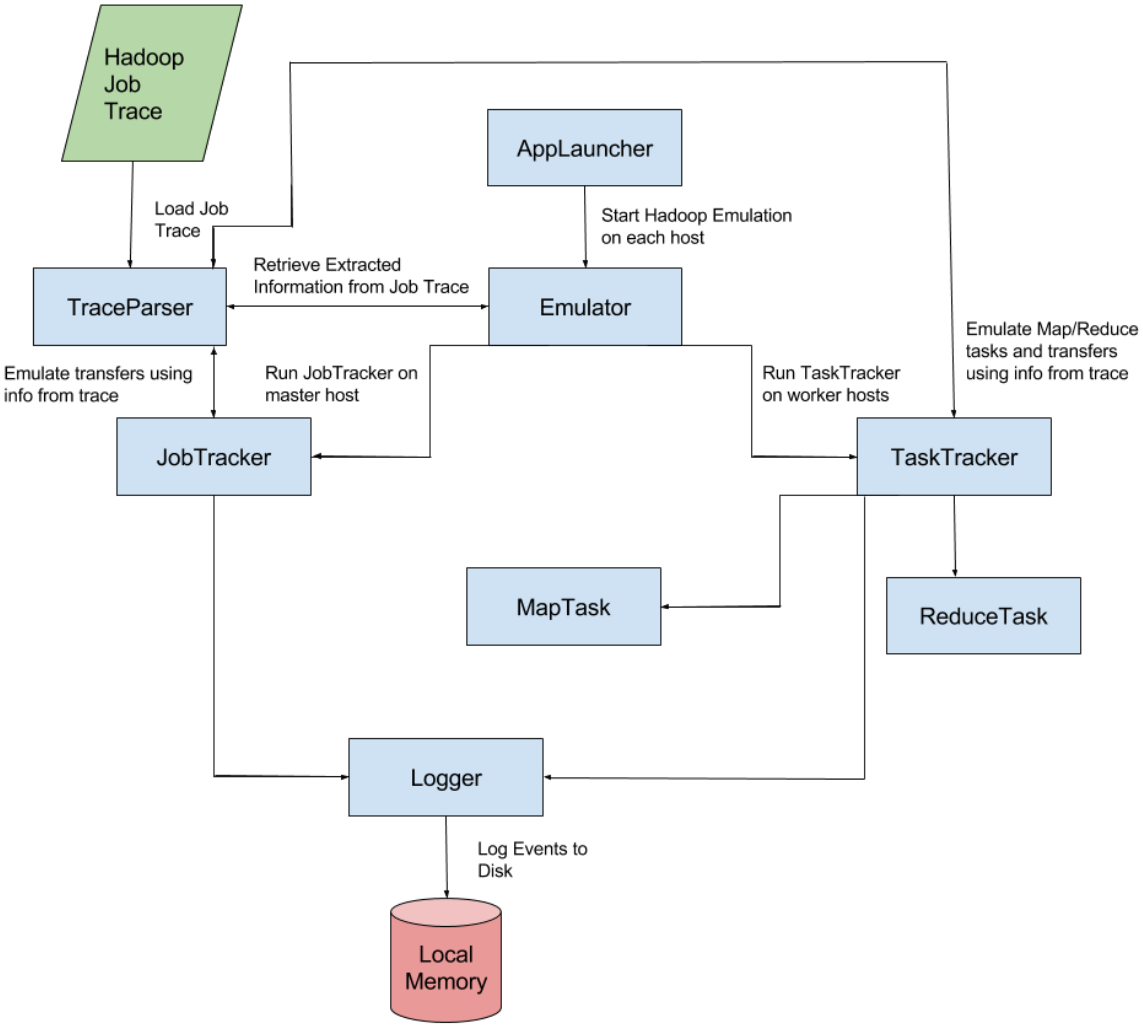


Figure 5.7: Functional Architecture of the Hadoop Emulation.

are logged in disk.

## 5.4 Throughput Measurement Methodology

In this section, the implementation for sampling throughput in the emulated network of our experiment is discussed. In order to measure the total bytes received by a host in the network, we leverage information obtained from */proc/net/dev* directory of each host in the network.

The */proc/* is a virtual Linux filesystem which is made available to user processes

by the Linux kernel in order to share internal information about the system [5]. The */proc/net/* directory contains a number of files providing some aspect of information on networking of the Linux system. The contents of the files in the */proc/net/* directory can be viewed by using the *cat* command. One such file in the */proc/net/* directory is the */proc/net/dev* file which provides information about the number of bytes transferred and received by the configured network interfaces of the system.

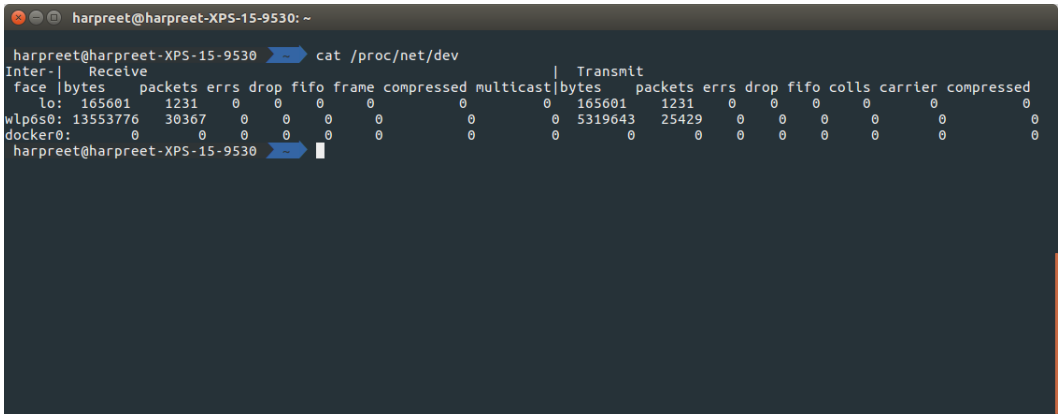


Figure 5.8: Sample output of *cat /proc/net/dev* command.

A sample output of the *cat /proc/net/dev* command is illustrated in Figure 5.8. At any instant, it gives the total number of bytes transmitted and received by the configured network interfaces. This information is leveraged to calculate the total bisection bandwidth achieved by the emulated Hadoop traffic in the network, when routed via *reactive* and *proactive* approaches.

Once the Hadoop emulation is executed on each host by the *LaunchExperiment* script, it launches the *sample_bandwidth(net)* function simultaneously, as illustrated in 5.1, which measures the bandwidth used by each host in the network from the */proc/net/dev* file.

Pseudocode for sampling bandwidth utilized by each host is illustrated in Figure 5.9. The *sample_bandwidth*(net) function creates an empty list of lists with the host objects as keys. Subsequently, while the Hadoop Emulation is running, it captures the total bytes received by the host interface by calling the *sample_rxbytes*(net, rxbytes) function, and storing the time duration between every two samples in a list. The *sample_rxbytes*(net,

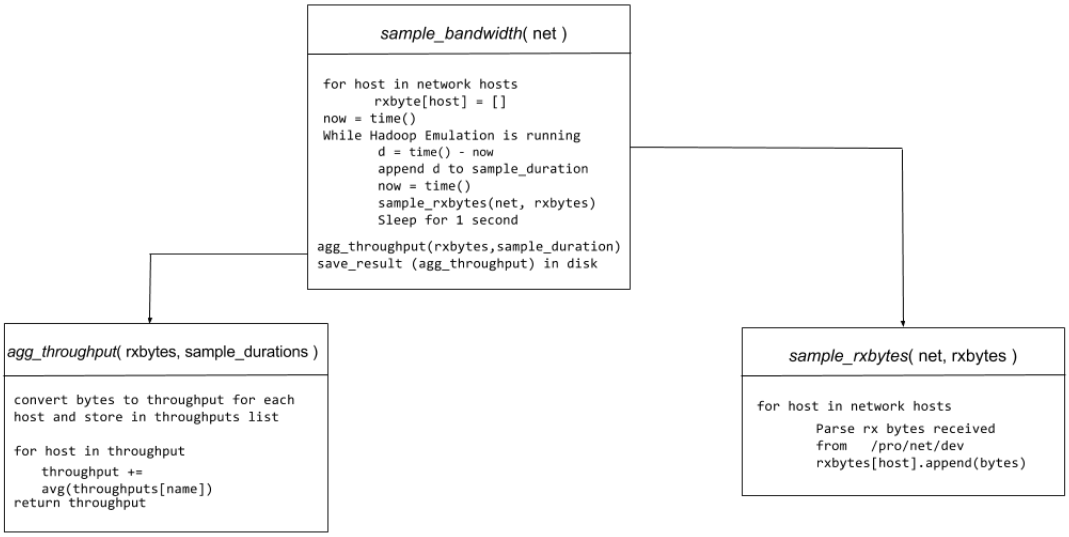


Figure 5.9: Pseudocode for sampling of network throughput from all hosts in the network.

rxbytes) function captures the number of bytes received by the network interface by reading the contents of the */proc/dev/net* file and stores it in a list corresponding to the host object.

When the Hadoop emulation terminates, the process of sampling bytes is stopped and the *sample_bandwidth*(net) function calculates the aggregate throughput achieved by calling the *agg_throughput*(rxbytes, sample_durations), where the list of bytes sampled and the the durations of their sampling are sent as parameters. The *agg_throughput*(rxbytes, sample_durations) function calculates the total aggregate throughput achieved by all the hosts in the network by converting the bytes to throughputs for each host and subsequently summing the average throughput achieved by each host in the network.

## 5.5 Summary

In summary, we implement the following components based on the design in Chapter 4

- An extension to the Global First-Fit flow scheduling controller which logs its flow scheduling decisions in a JSON file.
- A *Proactive* controller that leverages the flow scheduling decisions logged by the

Global First-Fit controller, by installing them *proactively* and subsequently defaulting back to the *reactive* behaviour of GFF controller.

- A mechanism to sample bytes received by each host in the network via the */proc/net/dev* directory, which is used to calculate the total throughput achieved by the hosts in the network.

The next chapter discusses the tests carried out to evaluate our implementation of the *Proactive* flow scheduling mechanism by comparing it against the total throughput achieved by *reactive* routing techniques and the effect of different flow scheduling mechanisms on Hadoop job completion times.

# Chapter 6

# Evaluation

The previous chapter detailed the implementation of our design proposed in Chapter 4, including the implementation of our *Proactive* flow scheduler. This chapter describes the benchmarking tests that were carried out to evaluate our *Proactive* flow scheduler against ECMP and Global First-Fit flow scheduling algorithms. Section 6.1 describes our experimental setup. Subsequently, in Section 6.2, we describe the nature of Hadoop job traces used for evaluation. Section 6.3 gives a brief overview of the methodology followed in obtaining experimental results. Sections 6.4 and 6.5 provide a comprehensive description regarding the evaluation of total average bisection bandwidth achieved by the different flow scheduling mechanisms and their effect on Hadoop job completion times. Finally, in Section 6.6, we summarize our findings and critically analyse the same.

## 6.1 Experimental Setup

The tests described in this chapter were all run on a Dell Server with 8 x 3.40GHz cores of Intel Core i7-4770 processor, 16 GB of RAM, and 475.4 GB of hard disk space running Ubuntu 14.04 LTS operating system. The Python implementation of our experiment used Python version 2.7. The Hadoop traces used in our evaluation were obtained by Neves *et al.* [47] on a real cluster of 16 identical servers, each having 12 x86 64 cores, a single

HDD and 128 GB of RAM, interconnected via 1 Gbps Ethernet links, running Hadoop 1.1.2 on top of Red Hat Enterprise Linux 6.2 operating system.

For running our network emulations, we used the *Mininet* network emulator from its cs244 branch version, since it is compatible with *Ripl-POX* controller that is used for ECMP routing, as discussed in 5.2. For the Hadoop emulation, we used the MRemu emulator [46], and used POX *dart* [7] SDN controller for flow scheduling of the emulated Hadoop traffic.

## 6.2 Benchmark Traces

The Hadoop job traces used to generate traffic in our experiment were obtained from the MRemu github repository [46]. The same traces were used by Neves *et al.* [47] to evaluate the performance of MRemu. The traces have been obtained from the following applications which are a subset of the HiBench benchmark suite [34]

- Sort - This application sorts text input data which is generated via a RandomTextWriter. Sort is used frequently to benchmark Hadoop performance. 32 GB of data was sorted to obtain Job traces.

- PageRank - It implements the page-rank [34] algorithm which calculates web page ranks by taking the number of reference links in the web page as a metric. Hence, PageRank serves as a large-scale indexing application. Neves *et al.* configured PageRank to process 500K pages, totalling approximately 1 GB of input size; and used the Pegasus Project [36] implementation of PageRank to obtain job traces.

- Nutch - It is part of Apache Nutch [50], which is a scalable and flexible web crawler that uses the MapReduce model for indexing pages in large-scale web search engines. Nutch was configured to index 5M pages, which totalled to approximately 8GB of input data.

- Bayesian Classification - Another canonical use of MapReduce is large scale machine learning. This application uses a classification algorithm for data mining and knowledge discovery called Naive Bayesian [34]. It forms a part of Apache Mahout, which is a popular machine learning library. To obtain Hadoop job traces, 100K pages were configured to be processed by Bayesian Classification.

However, the Hadoop job traces in the MRemu github repository [46] have not been classified according to the Hadoop applications that were run in the real cluster setup to obtain them. Therefore, in our evaluation of the effect of different flow scheduling mechanisms on total bisection bandwidth achieved and the respective Hadoop job completion times for different Hadoop job traces, we were not able to classify the different Hadoop traces on the basis of the HiBench applications that were used in order to obtain them. This is a limitation in our evaluation of the different routing mechanisms since different traces have achieved different levels of the network bisection bandwidth and we were not able to explore the causes for the same.

## 6.3 Evaluation Overview

We evaluated the effect of different routing mechanisms on the total bisection bandwidth achieved and Hadoop job completion times by running tests in the following manner

- Firstly, we launched the POX controller with ECMP routing.
- While the ECMP controller was running, we launched the Mininet network emulation of a 16 host fat-tree topology with each host running the Hadoop emulation.
- Subsequently, we stored the total bandwidth achieved and the total job completion times to disk, once the Hadoop emulation was over.
- We followed the same procedure listed above while running Global First Fit flow scheduling and our *Proactive* flow scheduling in the POX controller instead of ECMP

scheduling, and logged the total bandwidth achieved by the hosts in the emulation and their respective job completion times to disk.

The above procedure was repeated for different Hadoop job traces and the results obtained are discussed in the following sections.

## 6.4 Evaluation of Total Bisection Bandwidth Achieved

After running benchmark tests as described in the previous section, we plotted the total throughput achieved for ECMP, GFF and *Proactive* flow scheduling for different Hadoop job traces on the graph illustrated in Figure 6.1. Global First-Fit flow scheduling was observed to outperform ECMP routing in all instances, achieving **24.21**% more throughput on average than ECMP routing.

Al-Fares *et al.* [10] found Global First-Fit to outperform ECMP by achieving *39*% more of the total bisection bandwidth available in the network. The difference of *15*%, between our results and the ones obtained by Al-Fares *et al.* might be attributed to the fact that we run our experiment on a single server, causing the GFF flow scheduler to perform slower due to limited computational resources.

Our *Proactive* flow scheduling mechanism was observed to perform significantly better than ECMP, achieving **59.9**% more of the total bisection bandwidth available in the network than ECMP routing. In comparison to Global First-Fit routing, the *Proactive* flow scheduler was comparable for the first two of the Hadoop Job traces, *i. e.* **job1** and **job2** respectively, while for **job3** and **job4**, *Proactive* flow scheduling outperformed Global First-Fit scheduling, as illustrated in Figure 6.1. On an average, the *Proactive* flow scheduler achieved **11.6**% more of the total bisection bandwidth available in the network than Global First-Fit flow scheduling.

Since we are not aware of the exact nature of the Hadoop job traces used in the evaluation of the *Proactive* controller against ECMP routing and GFF flow scheduling, as described in 6.2, therefore, we cannot point out the reasons for *Proactive* flow

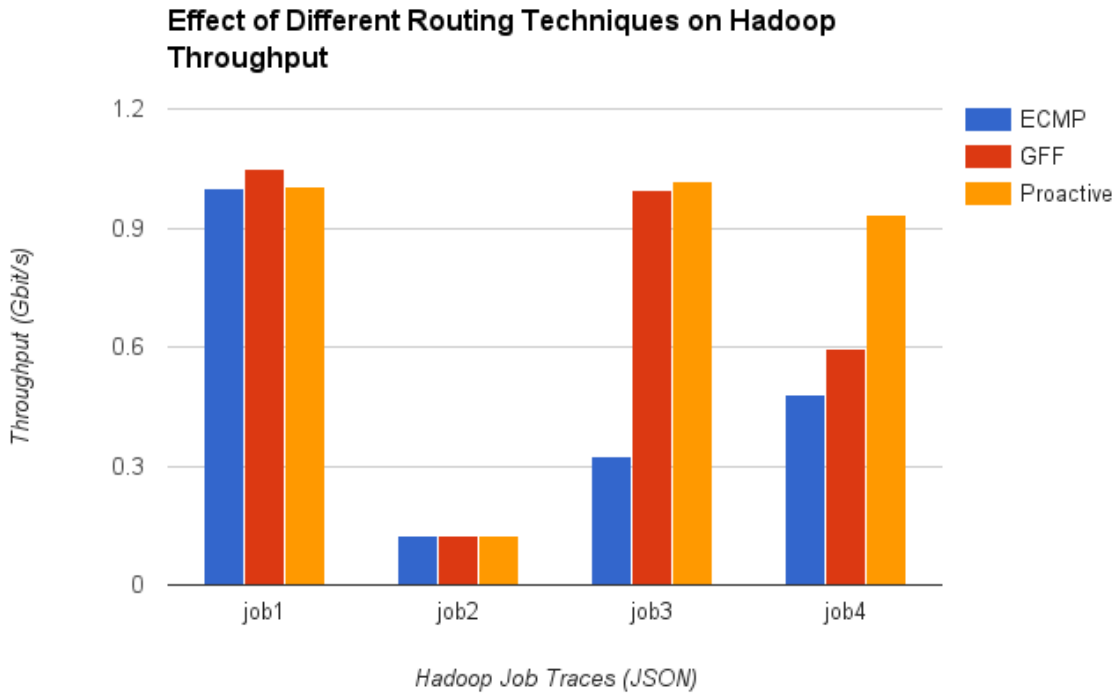


Figure 6.1: Total throughput achieved by hosts running Hadoop job emulation in an emulated network with 1 Gbps links, grouped by the throughput achieved when running different Hadoop jobs for ECMP routing, Global First-Fit flow scheduling and *Proactive* flow scheduling. On average, *Proactive* flow scheduling was found to achieve **59.9**% more bandwidth than ECMP routing and **11.6**% more bandwidth than GFF flow scheduling.

scheduling to outperform *Global First-Fit* scheduling for **job3** and **job4** Hadoop traces respectively, while performing comparably to *Global First-Fit* routing for **job1** and **job2** Hadoop traces, as illustrated in Figure 6.1. Nonetheless, the results obtained substantiate our claim that *Proactive* flow scheduling improves network performance by reducing the amount of control overhead in the network.

The throughput readings used in Figure 6.1 were taken by re-running the tests described in 6.3, and taking the *mode* of the throughput values obtained, so as to avoid *Random Errors* in the values since the experiment was compute intensive.

## 6.5 Evaluation of Hadoop Job Completion Times

As described in 6.3, we observed the Hadoop job completion times when traffic was routed following ECMP, Global First-Fit and *Proactive* scheduling mechanisms respectively, for different Hadoop Job traces and plotted the readings on the graph illustrated in Figure 6.2. We found the Hadoop job completion times to *correlate* with the average bisection bandwidth achieved by hosts in the emulated network, when traffic was routed following the three flow scheduling algorithms, as described in the previous section.

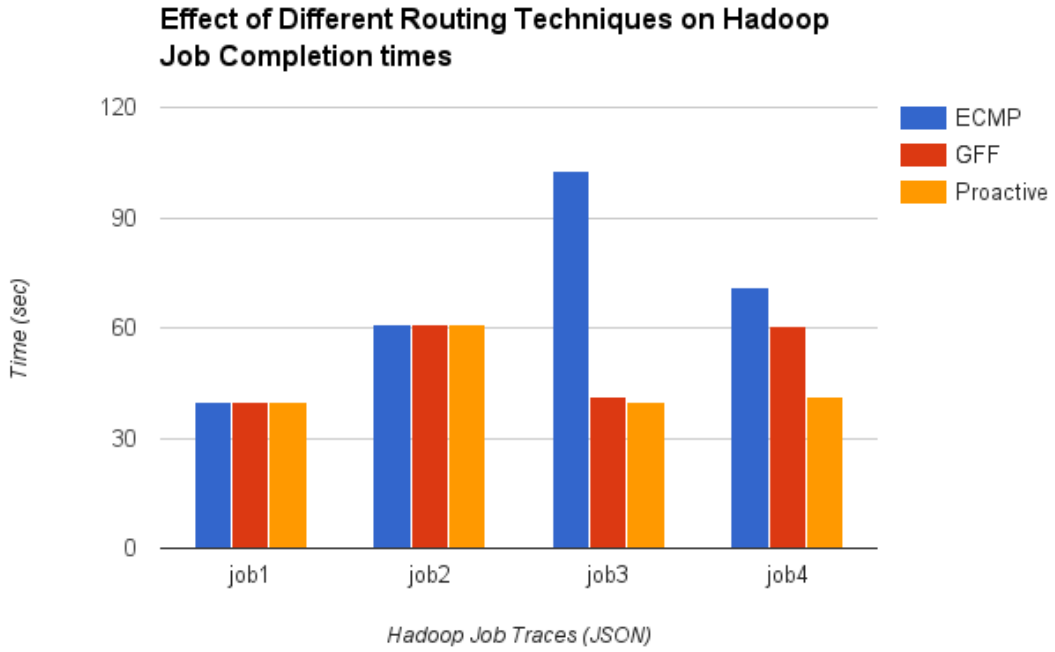


Figure 6.2: Time taken to complete Hadoop job emulation by hosts in an emulated network with 1 Gbps links, grouped by Hadoop job completion times when running different Hadoop jobs for ECMP routing, Global First-Fit flow scheduling and *Proactive* flow scheduling. *Proactive* Flow scheduler achieves lower Hadoop job completion times by **35.58**% in comparison to ECMP routing and **10.07**% in comparison to Global First-Fit flow scheduling.

*Hadoop job completion times* were found to have an **inverse correlation** with the *total bisection bandwidth* achieved, indicating that flow scheduling mechanisms that achieve a higher bisection bandwidth result in lowering of Hadoop job completion times. We calculated Pearson product-moment correlation coefficient for average throughput achieved

by different routing mechanisms as the dependent variable and the average time taken for Hadoop job completion by different routing mechanisms as the independent variable, and found the *correlation coefficient* to be **-0.9981**, which is indicative of total negative correlation between the two.

Moreover, Global First-Fit scheduling was found to lower Hadoop job completion times by **15.07**% in comparison to ECMP scheduling, while the *Proactive* Flow scheduler was found to lower Hadoop job completion times by **35.58**% in comparison to ECMP scheduling and **10.07**% in comparison to Global First-Fit flow scheduling.

## 6.6 Critical Analysis of Experiment Results

As mentioned in Section 6.2, the Hadoop job traces used were a subset of the HiBench benchmark suite of MapReduce applications, specifically, Sort, Nutch, PageRank and Bayesian Classification, produced by Neves *et al.* [47], made available on the MRemu github repository [46]. However, they were not classified into their application types, hence we were not able to account for the reasons behind the difference in bisection bandwidth achieved by different Hadoop job traces as illustrated in Figure 6.1; which is a limitation of our evaluation.

Nonetheless, we were able to evaluate the bandwidth achieved and Hadoop job completion times for ECMP, GFF and our *Proactive* flow scheduling mechanisms. Global First-Fit was found to achieve 24% more aggregate bandwidth than ECMP. Al-Fares *et al.* reported this difference to be 39%. We speculate the difference in our findings to be attributed to the fact that our experimental setup involved a single server emulating 16 hosts arranged in a fat-tree topology, resulting in a computation bottleneck as illustrated in Figure 6.3, which is a screenshot of the Ubuntu System Monitor utility, taken while the emulation was running, showing that at certain point of time, all 8 cores of our server used for the experiment were running at full processing utilization.

We evaluated our *Proactive* flow scheduling mechanism against ECMP and Global

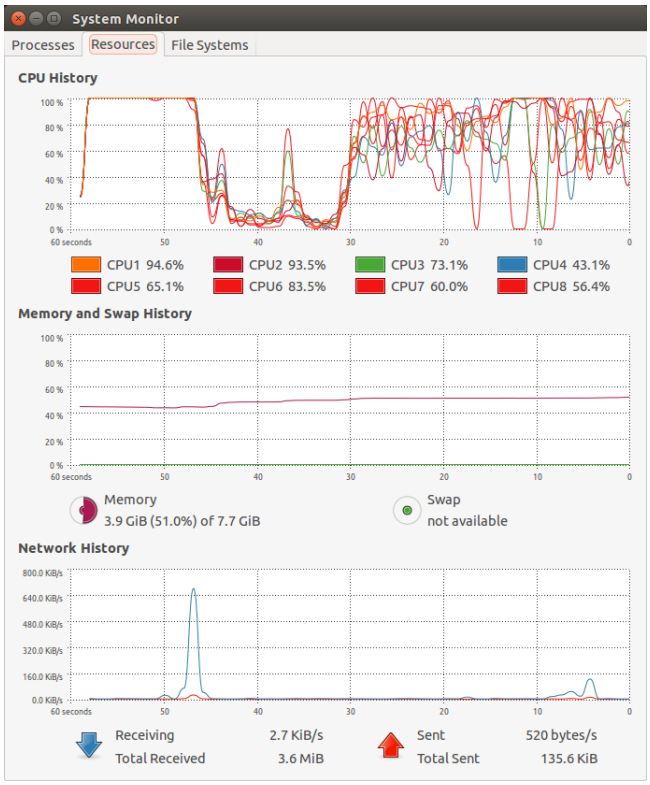


Figure 6.3: A screenshot of the CPU utilization while the experiment was running on the server.

First-Fit flow scheduling, and observed an average gain of **59.9**% and **11.6**% in total bisection bandwidth achieved against ECMP and Global First-Fit flow scheduling respectively. Moreover, *Proactive* flow scheduling achieved faster Hadoop job processing times by **35.58**% and **10.07**% in comparison to ECMP and Global First-Fit flow scheduling, highlighting the potential of *Proactive* Network configuration approaches to optimize performance of Big Data applications. Moreover, we found the total bisection bandwidth and Hadoop job completion times to be negatively correlated by Pearson's correlation coefficient of **-0.99**, which underlines the huge potential of *Reactive* and *Proactive* Network configuration to enhance the performance of Big Data processing applications in a data centre network.

# Chapter 7

# Conclusion

In the previous chapter, we summarized our evaluation of the *Proactive* flow scheduler in terms of its performance and compared it with ECMP routing and Global First-Fit flow scheduling. In this chapter, we will summarize the project and our results, and discuss possible future work.

## 7.1 Project Overview

Recent years have seen a steady growth in the amount of data generated by mobile and web applications, which is the fuel that drives Big Data Analytics. Mayer-Schönberger *et al.* [40] define "big data" as the ability of harnessing information in novel ways thereby producing valuable insights or goods of significant value. As the data volumes grow, big data processing frameworks such as Hadoop [31] require scaling out to thousands of commodity servers, subsequently resulting in an increase in the network traffic. Network performance has been found out to be of paramount importance for optimizing the processing times of Big Data applications, since research [10, 26, 30] has determined the network to be a performance bottleneck.

Recent studies [9, 26, 30, 29] propose horizontal scaling of hosts to thousands of commodity servers in multi-rooted tree topologies, such as a *fat-tree* topology that exploit

path diversity to overcome limited port densities in commodity switches, thereby scaling the network with the increasing number of servers required to process large volumes of data. Moreover, emergent technologies such as Software-Defined Networking enable programming of the network stack, by maintaining a global view of the network state which is enabled by the separation of the control plane from the data plane of forwarding devices [38].

Various research efforts [21, 56, 10, 45, 48, 57] have tried to make the network dynamically reconfigure according to application traffic demands, in order to avoid congestion and optimize big data processing in a data centre network. Dynamic network reconfiguration approaches are *reactive* in nature and try to optimize network performance by either being *application-aware*, where the big data application controller reports traffic demands to the network controller which subsequently reconfigures the network accordingly, or by adopting the approach of *traffic-awareness*, where a network monitor provides information to the network controller for *reactive* reconfiguration. However, such *reactive* measures of network configuration are bound to induce control traffic in the network and cause high level latencies in reconfiguring the network dynamically.

We propose a *Proactive* approach for the configuration of a data centre network, which installs flow rules in the network before the big data application starts. The flow rules are obtained from the previous execution of the same application when it is routed by the *Global-First Fit* [10] flow scheduling algorithm, which is a *traffic-aware* algorithm that builds upon *Equal Cost Multi-Path* (ECMP) routing. ECMP is the standard routing protocol in multi-rooted data centre networks with path *multiplicity*. Our *Proactive* network controller reverts to Global First-Fit flow scheduling once the *proactive* flows installed by it in the forwarding devices expire.

In order to evaluate our *Proactive* network controller against ECMP routing and Global First-Fit flow scheduling, we ran emulations of Hadoop jobs on a 16 host *fat-tree* topology, running on a single server, and measured the effect of the different flow scheduling mechanisms on total bisection bandwidth achieved by the hosts in the net-

work and Hadoop job completion times. An average gain of **59.9**% and **11.6**% in total bisection bandwidth achieved by the hosts in the network in comparison to ECMP routing and Global First-Fit flow scheduling was observed, when network traffic was routed by the *Proactive* controller. Moreover, the *Proactive* controller reduced Hadoop job completion times by **35.58**% and **10.07**% in comparison to ECMP routing and Global First-Fit flow scheduling.

Automatic generation of *Proactive* configurations for the network was beyond the scope of this project, therefore we used the network configurations generated by Global First-Fit flow scheduling. The results obtained by us indicate that *proactively* configuring the network results in an increase in the total network utilization which benefits the performance of big data processing applications. Given that current *application-aware* and *traffic-aware* approaches of dynamic network configuration are not optimized with *Proactive* configurations, there may be more to be gained in terms of average network bisection bandwidth utilization, if they are optimized before deployment.

## 7.2 Contribution

In summary, our project validates the effectiveness of *Proactive* configuration of data centre networks in optimizing big data application performance. Our evaluation results demonstrate that *reactive* approaches have a lot to gain if they are optimized *proactively* before deployment, since our *Proactive* flow scheduler is able to achieve higher bisection bandwidth utilization in a data centre network and lower Hadoop job completion times than current static ECMP routing and *reactive* approaches explored in research, such as Global First-Fit flow scheduling.

Furthermore, we established a *high negative correlation* between total bisection bandwidth achieved in the network and Hadoop job completion times, further highlighting the significance of network performance in accelerating big data applications.

## 7.3 Future Work

While our *Proactive* flow scheduling approach has increased the network performance in terms of the total bisection bandwidth achieved, it relies on flow scheduling decisions made by the *Global First-Fit* algorithm for the same Hadoop job. In order to fully explore the effect of *Proactive* configuration of the network, the network has to be configured proactively based on the application communication patterns. A future extension to our *Proactive* flow scheduler might be to independently install *Proactive* configurations in the network by generating them automatically, on the basis of application communication patterns.

We based our experiment on an emulation based testbed, since we did not have access to the hardware resources for deploying our implementation on a real cluster of 16 hosts. Moreover, for the same reason, we used the Hadoop job traces obtained by Neves *et al.* [47], which were not classified according to the MapReduce applications that were run to obtain the traces, thereby hindering our ability to account for the difference in total bisection bandwidth achieved for different Hadoop job traces as discussed in Chapter 6. Consequently, a future extension to our experiment would be to obtain Hadoop job traces from a real cluster and subsequently run the experiment on the same, in order to validate the findings from our emulation based testbed and account for the difference in the average bisection bandwidth achieved for different Hadoop applications.

## 7.4 Final Remarks

Big Data processing is being used to solve complex problems of society by levering enormous volumes of data available from ubiquitous computing devices. In order to ensure robust performance of big data applications, data centre networks have to be configured according to application communication patterns, so that traffic congestion in the network is avoided. By utilizing the emergent technology of Software-Defined Networking,

the network stack can be programmed to configure a data centre network in accordance with application communication patterns, enabling scalability of network performance as more commodity servers are added into the network in order to deal with the growing data volumes. We showed that optimizing the network *proactively* results in performance gains over current *reactive* approaches and hope that as research evolves out in this field, the full potential of *Proactive* network configuration for optimizing big data processing will be exploited.

# Appendix A

# Abbreviations

| Short Term | Expanded Term |
|---|---|
| SDN | Software Defined Networking |
| ECMP | Equal Cost Multi-Path |
| RPC | Remote Procedure Call |
| HDFS | Hadoop Distributed File System |
| MR | MapReduce |
| CPU | Central Processing Unit |
| ToR | Top of Rack |
| TCAM | Ternary Content Addressable Memory |
| GFF | Global First Fit |

# Appendix B

# Source Disk Contents

This document also includes a source disk containing:

- The complete source code and referenced libraries for the Python implementation described in Chapter 5, along with a readme file for running the experiment (in the folder **src**),
- A spreadsheet with the results obtained for evaluating the experiment in Chapter 6 (in the folder **results**).

# Bibliography


[1] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/. Accessed: 2016-08-22.

[2] Open vSwitch: An open virtual switch. http://openvswitch.org/. Accessed: 2016-08-22.

[3] Opnet modeler. http://www.riverbed.com/gb/products/steelcentral/opnet.html. Accessed: 2016-08-22.

[4] POX SDN Controller. http://www.noxrepo.org/. Accessed: 2016-08-26.

[5] Exploring the /proc/net/ Directory. http://www.onlamp.com/pub/a/linux/2000/11/16/LinuxAdmin.html, 2000. Accessed: 2016-08-27.

[6] Hedera flow scheduler implementation. https://reproducingnetworkresearch.wordpress.com/2015/05/31/cs244-15-hedera-flow-scheduling-draft/, 2015. Accessed: 2016-08-26.

[7] POX dart branch. https://github.com/noxrepo/pox, 2016. Accessed: 2016-08-26.

[8] ripl-POX: Ripcord-Lite for POX. https://github.com/brandonheller/riplpox, 2016. Accessed: 2016-08-26.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[11] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[12] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[13] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, page 39, 2008.

[14] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[16] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive packet sampling for accurate and scalable flow measurement. In *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, volume 3, pages 1448–1452. IEEE, 2004.

[17] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[18] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.

[19] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[20] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. *ACM SIGCOMM Computer Communication Review*, 41(4):254–265, 2011.

[21] Anupam Das, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Curtis Yu. Transparent and flexible network management for big data processing in the cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[24] Linux Foundation. OpenDayLight Project. `https://www.opendaylight.org/`, 2016. Accessed: 2016-08-21.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[26] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2:

a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[27] Albert Greenberg, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62. ACM, 2008.

[28] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[29] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.

[30] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 75–86. ACM, 2008.

[31] Apache Hadoop. `http://hadoop.apache.org/`. Accessed: 2016-07-05.

[32] Americas Headquarters. Cisco data center infrastructure 2.5 design guide. 2007.

[33] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[34] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[35] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[36] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[37] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[38] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[39] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[40] Viktor Mayer-Schönberger and Kenneth Cukier. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.

[41] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[42] Nick McKeown. How SDN will shape networking. *Open Networking Summit*, 2011.

[43] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[44] John Moy. Open shortest path first (ospf) version 2. *IETF: The Internet Engineering Taskforce RFC*, 2328, 1998.

[45] Sandhya Narayan, Stuart Bailey, and Anand Daga. Hadoop acceleration in an openflow-based cluster. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 535–538. IEEE, 2012.

[46] Marcelo Veiga Neves. Mremu: An emulation-based framework for datacenter network experimentation using realistic mapreduce traffic. `https://github.com/mvneves/mremu`, 2015. Accessed: 2016-08-21.

[47] Marcelo Veiga Neves, Cesar AF De Rose, and Kostas Katrinis. Mremu: An emulation-based framework for datacenter network experimentation using realistic mapreduce traffic. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 174–177. IEEE, 2015.

[48] Marcelo Veiga Neves, Cesar AF De Rose, Kostas Katrinis, and Hubertus Franke. Pythia: Faster big data in motion through predictive software-defined network optimization at runtime. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 82–90. IEEE, 2014.

[49] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50. ACM, 2009.

[50] Apache Nutch. `http://nutch.apache.org/`. Accessed: 2016-07-27.

[51] Yang Peng, Kai Chen, Guohui Wang, Wei Bai, Zhiqiang Ma, and Lin Gu. Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 19–27. IEEE, 2014.

[52] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil

Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, et al. Carving research slices out of your production networks with openflow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[53] Apache Spark. `http://spark.apache.org/`. Accessed: 2016-08-19.

[54] Dave Thaler and C Hopps. Multipath issues in unicast and multicast next-hop selection. Technical report, 2000.

[55] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, 2007.

[56] Guohui Wang, TS Ng, and Anees Shaikh. Programming your network at run-time for big data applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 103–108. ACM, 2012.

[57] Philip Wette and Holger Karl. Hybridte: Traffic engineering for very low-cost software-defined data-center networks. In *2015 Fourth European Workshop on Software Defined Networks*, pages 31–36. IEEE, 2015.

[58] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.