TÉCNICO LISBOA

Tenant Aware Big Data Scheduling with Software Defined Networking

I.Tasneem Akhthar

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga

Examination Committee

Chairperson:	Prof. João Emílio Segurado Pavão Martins
Supervisor:	Prof. Luís Manuel Antunes Veiga
Member of the Committee:	Prof. Fernando Henrique Corte-Real Mira da Silva

October 2016

Acknowledgments

First and foremost, I would like to extend my sincere gratitude to my research guide, Professor Luís Veiga, for introducing me to this exciting field of Software Defined networking with Big Data and for his dedicated help, advice, inspiration, encouragement and continuous support, throughout my thesis.

I would also like to thank Pradeeban Kathirvelu who helped me to tackle and understand about the topic better. Then I would like to thank the open-source community in general and the floodlight and mininet mailing lists in particular. Their prompt answers and guidance helped me lot to navigate in these giant projects.

And finally, I would like to thank my parents, who formed part of my vision and taught me good things that really matter in life. Their infallible love and support has always been my strength. Their patience and sacrifice will remain my inspiration throughout my life.I am also very much grateful to all my family members for their constant inspiration and encouragement.

Abstract

The increase of data in the Internet world has created a need to process and acquire information from them using Big Data Analytics which in turn use data center for computing and storing purposes. The Big data analytics in data center needs a good network configuration to avoid delay or error in the network. But the traditional network could not avoid the error or dynamically create a network architecture. This gave rise to Software-Defined Networking which configures, deploys and manages the network infrastructures and is most widely used in the data center network.

Both these technology SDN and Big Data has benefited the data center network tremendously. SDN follows logically centralized approach, with which network allocation and scheduling can be performed with increased efficiency and reliability.

Due to the emergence of cloud services it has becoming easy to access the computational resource. The client can rent large size of the computational resources in a very reasonable price, but unfortunately there is performance degradation in the network level as many clients would be using the computational instance through same communication channel and SLA (Service-level agreement) could not guarantee the performance in the network layer. This bring about the problem in the multi-tenant data center environment.

Having these two technologies in mind we propose *MR-FLOOD* which is a conjugation of the Hadoop MapReduce framework and Floodlight controller. We brought these technologies together to give the tenants a fair bandwidth share and less latency using bandwidth or latency based job allocation strategy.

Our assessments show that the above mentioned properties are achieved, being carried out in two common data center network topologies: Tree and Fat-tree.

Keywords: Software-Defined Networking (SDN); Big-data processing; Job scheduling; multi-tenancy; cloud computing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Massive increase in the usage of social networking, data intensive technologies, internet, cloud computing, has shown the growth of data in an unexpected extent. This huge data collected can be processed to extract helpful information, which is known as ”Big data Analytics” and it is done by the data-intensive analytic framework such as Hadoop [39], Spark [42], Dryad [20] and many others.

From many years MapReduce is used as the big data framework which does data processing across large-scale computing clusters. Due to its simplicity and scalability it is used by big data analytics to parallelize the job and process the data. Among many features, One of the main features of MapReduce is its ability to make use of data locality and minimize network transfers. But according to the research [17] there is performance degradation in MapReduce framework as there occurs bottleneck in the network due to job completion time in shuffle phase. For example, the recent study of MR traces from Facebook acknowlegde that the shuffle phase uses more than 50% of the job completion time [43]. This problem can be solved by optimizing the network used for communication, to reduce the response times.

MapReduce normally runs in large data centers (DCs) where network must be utilized in a proper way to increase the productivity. Many researchers have concentrated in optimizing the network based on the scheduling algorithms to improve the data locality and data transfer in the network. But very few work has been carried out in order to dynamically adapt the network behavior to MapReduce application's needs. So there must be some kind of controller which is already aware of the application's traffic demand in the network. These observations helped to improve the performance of individual MR applications and the overall network utilization, by introducing the revolutionary idea which is known as Software-Defined networking (SDN). SDN controls the network through software which in turn makes the network programmable. It has been introduced as a replacement for conventional networking methods so that it can meet today's market requirements.

Software-Defined Networking is a technique which allows a network controller to manage the network scheduling, in order to make full use of the resources and improve the performance of the software

running on the network. In short, SDN decouples the network control (control plane) and forwarding functions (data plane) which enables the control over the network and abstracts the underlying network infrastructure for applications and network services.

In Figure 1.1 we can see the abstraction provided by SDN.

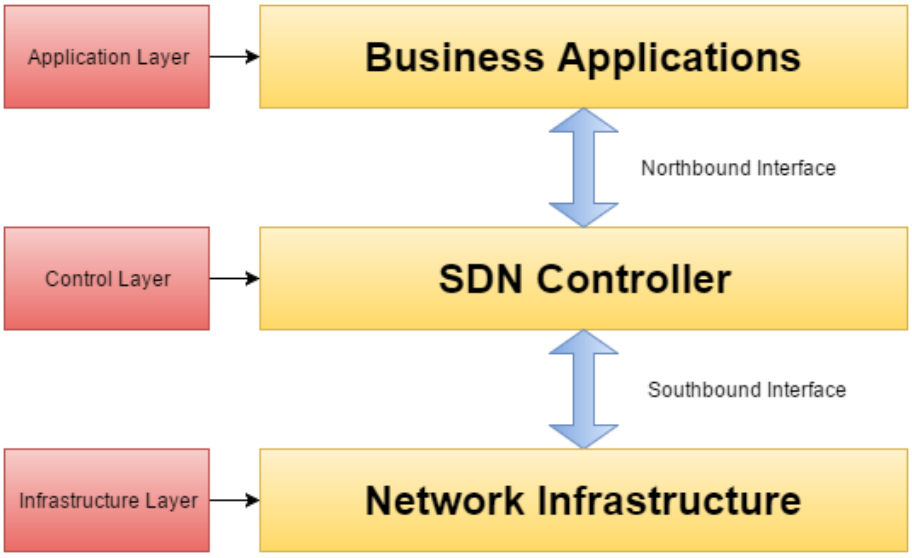

Figure 1.1: Fundamental abstractions provided by SDN.

Software-Defined Networking allows large scale networks to be coordinated and managed effectively from a logically centralized controller. Large scale cloud and networking systems are built leveraging the programmability and reconfigurability offered by SDN. For Big Data applications, software-Defined networks provide for the ability to program the network at runtime, in a manner such that, data movement is optimized for faster execution.

While existing MapReduce frameworks such as Hadoop are exploited for real time evaluation of QoS-driven service composition, SDN and OpenFlow controllers should be leveraged to enable a QoS-driven self-configuration of the MapReduce platforms. This research proposes a bandwidth and latency aware MapReduce service composition and self-configuration for big data Analytics with SDN, by extending and leveraging Hadoop MapReduce framework and floodlight SDN controller. Our project is also responsible for allowing the tenant to make use of the big data application(Hadoop) and still keep the data isolated from each tenant in the data center.

There are other bog data frameworks which is recently developed and they do give better performance but we chose MapReduce frameowrk as it was used in MRemu tool [30] to which we are upgrading in this project, we also improve the performance of MapReduce jobs through runtime communication using Software-Defined network, which is used to have a global view of the state of the data center network. This way we are able to allocate the MapReduce jobs in the Data center by ensuring the bandwidth and bandwidth. It is evaluated by trace-driven emulation-based experiments, using popular

benchmarks and real-world applications that are representative of significant MapReduce uses.

1.1 Shortcomings of Current Solutions

A great number of works identified the problems we have outlined above, and currently Big Data utilizing SDN is a hot research topic with a lot of attention. However, as later described in the document, those solutions focus either on providing latency or bandwidth guarantees to each tenant or on maximizing network utilization and bandwidth sharing among tenants. This is not enough because these works, either increase the satisfaction of tenants (providing them with guarantees), or improve the network management of data centers.

1.2 Objectives

The main objective of this master project is to improve the state-of-the-art, by leveraging SDN on Big Data and boost up the performance measures. Our objectives are characterized as properties of the system we want to build. Thus, we develop a network management solution on data center network that:

- Is scalable to Data Center environments;
- Provide bandwidth guarantees for the MapReduce job;
- Provide latency guarantees for the MapReduce job;

1.3 Document Roadmap

The rest of this work is organized as follows: In chapter 2 we present the related work, describing our enabling technologies as well as the research works that are similar to ours. Chapter 3 describes the architecture, algorithms and data structures of our solution, focusing on the main components and their interactions. In chapter 4 we present the implementation we have done by following the design depicted in chapter 3. Then, in chapter 5 we evaluate our implementation, taking into account the objectives defined in the previous section. Lastly, in chapter 6 we make some concluding remarks about this work and point out some possible directions for future work.

Chapter 2

Related Work

In this section we present the industry and research work considered relevant for the development of this thesis. In section 2.1, we start by describing our key enabling technology Open Networking Foundation. In the next section 2.2, we present the taxonomy of the big data related with this thesis. In section 2.3, we describe the integration of two eminent technology. Finally, in section 2.4 we summarize analysis of all the studied systems in order to wrap-up the related work.

2.1 Open Networking Foundation (ONF)

The present and the future Internet technologies are emerging where majority of the devices are connected to the Internet due to which new challenges are commanding to build a network which gives maximum of the bandwidth usage, dynamic management without any delay or error in the network. The traditional approaches based on manual configuration of network devices are cumbersome and cannot fully utilize the capability of physical network infrastructure. So for the betterment of the networking infrastructure, ONF which is an organization implements and promotes Software-Defined Networking using the open standards, which would simplify the network design. This open standard provided by ONF is the OpenFlow protocol which will be discussed in the later section.

Software-Defined Networking (SDN) has been introduced as one of the most promising solutions for future Internet. It is an emerging networking paradigm that has the biggest break through in the networking infrastructure where it finally separates the network's control logic (the control plane) from the underlying routers and switches that forward the traffic (the data plane) [21]. Now with the separation of the control and data planes, network switches are just a forwarding devices and the control logic is implemented in a logically centralized controller. SDN allows to program the network application [40]. All these features such as efficient configuration, better performance, and higher flexibility to accommodate innovative network designs is provided by SDN through ONF.

SDN was mainly used in research, global WAN, campuses and clouds. The major difference in

SDN and the ordinary network is that it is programmed at the network level through open programming interface rather than through vendor specific switch level command line interface. With all these features SDN supports the data center network.

2.1.1 Software-Defined Networking Protocol:

ONF has proposed a reference model for SDN. This model consists of three layers, i.e. an infrastructure layer, a control layer, and an application layer, piled up over each other.

LAYER 1:*Infrastructure Layer*

The infrastructure layer consists of switching devices such as switches, routers, etc in the data plane. Switching devices collects the network status and stores them in the local device temporarily and then sends the network status to the controller to realize the information such as network topology, traffic statistics, and network usages. In Infrastructure layer the switches processes the packets based on the rules which the controller has defined.

LAYER 2:*Control Layer*

The control layer links the application layer and the infrastructure layer with its two interfaces namely *South bound interface and North bound interface*. To interact with the infrastructure layer (i.e.,south-bound interface such as OpenFlow,NETCONF etc) which is underneath to the control layer, it specifies functions for controllers to access functions provided by switching devices. The functions may include reporting network status and importing packet forwarding rules. And to interact with the application layer (i.e.,the north-bound interface such as REST, REST-CONF, Java APIs) which is above the control layer, it provides service access points in numerous forms, for example, an Application Programming Interface (API).

SDN applications can access network status information recorded from switching devices through this API, make system tuning decisions based on this information, and carry out these decisions by setting packet forwarding rules to switching devices using this API. Since multiple controllers will exist for a large administrative network domain, an “eastwest” communication interface among the controllers will also be needed for the controllers to share network information and coordinate their decision-making processes.

LAYER 3:*Application Layer*

The application layer consist of SDN applications designed to fulfill user requirements. Control layer offering a programmable platform with which the SDN applications are able to access and control switching devices at the infrastructure layer. Dynamic Access Control, Seamless Mobility and Migration, Server Load Balancing, Network Monitoring, Attack Detection, Network Virtualization are some of the examples of SDN application.

Now we explain about the most commonly used Southbound interface which is known as Openflow [25]. OpenFlow is an interface between the control and data planes of the SDN technology. Through this interface the information is exchanged between planes.

2.1.1.1 OpenFlow:

OpenFlow is protocol which is presented by McKeown et al. with an objective to enable easy network experiments in a campus network [25] and is currently used in SDN. Earlier the experiments which were using OpenFlow were mainly aimed at creating a separate software controllable network focusing on controlling how to forward the packets. Later, the researchers identified that the implementation of a separate software controllable network using OpenFlow is actually practical to enable the "Software-Defined Networking". Openflow spilts the control plane with the data forwarding plane and exchanges information.

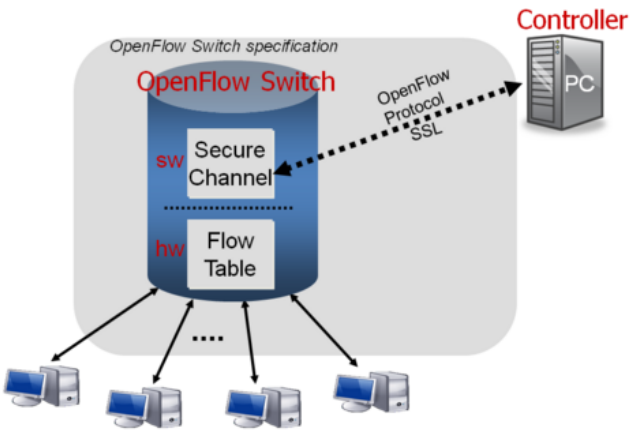

Figure 2.1: OpenFlow Architecture [25].

The OpenFlow protocol offers flow table manipulation services for a controller to insert, delete, modify, and lookup the flow table entries through a secure TCP channel remotely. OpenFlow may be seen as just a protocol specification used in switching devices and controllers interfacing. Its idea of creating a separate network solely for network control manifests the key concept of SDN and lays foundation for network programmability and logically centralized control. Development of SDN and OpenFlow has same concepts and design approaches which go hand in hand with each other. On one hand, by developing the concepts in SDN are based on the design of OpenFlow. On the other hand, as the concept of SDN becomes clearer and more mature, then it would influences the future development of OpenFlow. We can say that, OpenFlow defines initial concept of SDN and SDN defines future development of OpenFlow.

In the OpenFlow architecture, illustrated in 2.1, OpenFlow makes use of the modern Ethernet switches and routers containing the flow tables for essential networking functions, such as routing, subnetting, firewall protection, and statistical analysis of data streams. Flow tables consist of flow entries, each of which determines how packets belonging to a flow will be processed and forwarded. In an OpenFlow

switch, each entry in the flow table has three parts,

i)"header" used to match incoming packets
ii)"action" or a set of instructions, to define what to do for the matched packets
iii)"counters", used to collect statistics for the particular flow, such as number of received packets, number of bytes and duration of the flow;

When the packet arrives at an OpenFlow switch, packet header fields are extracted and matched with the header portion of the flow table entries. If there exists matching entry then the switch applies the appropriate actions, associated with the matched flow entry. If there is no match in flow table look-up,then the action will depend on the instructions defined by the table-miss flow entry. Every flow table must contain a table miss entry so that the switch can handle table miss.

2.1.1.2 OpenFlow versions:

Below we will be describing the OpenFlow version and their features evolution in time to time.

OpenFlow version 1.0:

OpenFlow 1.0 was released in the year 2009 with one flow table and three components having Header field, Action field and counter field. In this version there is a enqueue option through which the packet can be queued to some specific port. In this version the header field can only be checked upto 12 fields, if suppose the header field has more than 12 elements then it cannot match and this is a complication caused by this version. Moreover there is only one flow table, so only one operation can be performed at a time. This becomes major problem in this version and later to avoid this another version of OpenFlow was evolved which is described below.

OpenFlow version 1.1:

OpenFlow 1.1 came into existence in the year 2011 and it fixed the previous version issues by introducing multiple flow tables and group table. But having multiple flow table was not easy to maintain and the vendors of the switch was feeling difficult to match up with the multiple flow table. To avoid this ONF came up with Negotiable DataPlane Model(NDM) with which the switch vendors were able to provide different type of table patterns, matching the need of the switch operator. The other major change in version 1.1 was the evolution of Group table, where the actions set can be applied to the group of flows. It consists of group entries which can be divided into four types. Such as All, Select, Indirect, Fast failover.

OpenFlow version 1.2:

OpenFlow 1.2 was released very soon after the version 1.1. It came into existence in December 2011. In the previous version there was fixed match field but in this version the match field was made flexible with the introduction of OXM (OpenFlow Extensible Match). Apart from this, OpenFlow version 1.2 also supported IPV6 and ICMPv6. The other disadvantage the previous version had, only one controller supported but in this version a new option was introduced to avoid single point of failure and it

supported multiple controller acting as Master or Slave role.

OpenFlow version 1.3:

OpenFlow version 1.3 was released in April 2012. The header fields supported the extension of VLAN and IPv6. For the betterment of video streaming and IP telephony, QoS was introduced in version 1.3. Apart from this the other feature introduced in this version was Meter table. It was used for the purpose of monitoring the traffic rate, of the flow. It even had the statistics about the duration of the per meter-flow.

OpenFlow version 1.4:

OpenFlow version 1.4 was released in August 2013. It had advanced features such as Synchronized tables, which would synchronize the table in two directions, such as unidirectional and bidirectional. It has extensions towards the wired protocol. It even introduced a feature called Bundles, it is similar to the atomic operation where all modifications in the group are committed or none of them are committed. This version also has vacancy events, role state events, group and meter change notification. The other change this version had was the change in the TCp port to 6653.

OpenFlow version 1.5:

OpenFlow version 1.5 was released in January 2015. This is the latest version while writing this thesis. From previous version it enhanced the scheduled bundle to include execution time. The other feature in this version includes Egress table, extensible flow entry statistics, TCP flag matching, packet type aware pipeline.

2.1.2 OpenFlow Network Simulators/Emulators

Below there are few network simulators and emulators used by OpenFlow. These emulator are used to emulate the data center network.

Mininet 1.0:

It is a network emulator which is flexible, deployable, interactive scalable, realistic and shareable. It was developed by Lantz et. al.[22] to create a prototyping workflow on large networks with less resources such as a laptop, which has constrained amount of resources. These workflow is supported by the lighweight OS level virtualization. The clients can create the network and test it with large topologies, if it works then it can be deployed for the usage. Mininet runs remarkably on the laptop which runs on Linux distribution. with the help of Command Line Interface and Application Program Interface one can create, interact and share the network in Mininet.

To create a network in Mininet the following components are used: *Links, Hosts, Switches, Controllers*. Link is made by a virtual Ethernet Pair also called as veth pair, which creates a logical connection between the virtual interface. Now the packets can be sent to each other. Hosts are the containers for

the network state which is also known as Network Namespace. Switches are as same as the hardware switch which would deliver the packet. It has user space and Kernel space switches. The last component for creating the network is Controller which can be situated anywhere in the network.

Mininet 2.0 (Hi-Fi):

Mininet 2.0 [18] was developed by Nikhil Handigol et al., In version 1 the Mininet could not perform better when the load was increased. So the author came up with Mininet Hi-Fi which would give better performance at even High loads by providing resource isolation, resource provisoining and monitoring mechanism. Mininet Hifi is a container based emulator, which creates realistic networking experiments. To perform the resource isolation Mininet Hi-Fi uses Control Groups (cgroups) in which all the processes in a group is dealt in a similar manner. To provision the resource for the network Mininet Hi-Fi allows the client to allocate the CPU by splitting it among the containers, then to choose the needed link speed and topologies. The last improvement on Mininet Hi-Fi was to monitor the performance which is done by measuring the inter-dequeue time of the packet which would let us know if there is any delay.

MaxiNet:

MaxiNet [37] is also a network emulator developed by Philip Wette et al., but the roots were still from Mininet with some changes like stretching the emulation of large number of nodes on some constrained machine. The main idea behind MaxiNet is to use the emulation on the data center network. MaxiNet contains several Mininets which works as a slave. These instances of the Mininet has to be accessed from a centralized API which acts as a Master. All the communication which happens between the MaxiNet and Mininet is through the Remote Procedure Call (RPC). MaxiNet does not work well in the heterogeneous clusters where the load has to be distributed evenly on all the slave. Due to this there might be bottleneck occurring in the machines with less resources.

EstiNet:

EstiNet [36] can be worked as the network emulator and the network simulator. As it is expansion of all the above techniques, the cost to build also sums up, where the real devices and real operating system are used due to which the results are very accurate. Wang et al., developed the EstiNet network simulator by using kernel re-entering methodology where the tunnel networks plays the main role by allowing the exchange of the packets between two applications and sending them to the simulation engine. By default, the EstiNet can be used as network emulator. As described earlier EstiNet gives appropriate results when it is compared with the theoretical value but due to high cost and increase in the execution time makes it unpopular among others.

We use Mininet 2.0 as it was used by the MRemu tool [30] which will be described in the next chapter.

2.2 Big Data Platforms

Big Data is a trend setter where the data is too huge. The large amount of data should be handled efficiently to acquire the appropriate knowledge from it. This appropriate knowledge is acquired by processing the data which means the data also has to be analysed apart from just storing it. Below we have mentioned some of the concepts of Big Data which would enlighten our work. First we would discuss about the processing methods and then the about storage in big data.

2.2.1 Batch-Data Processing

When we try to process massive amount of data it becomes very difficult to process it. To decrease the overhead on the resources which process these huge data at a time, we try to make small chunks of the data to process. This is much easier when compared to huge amount of data. This technique is knows as Batch-Data processing. Some Batch-Data processing techniques relevant to our work are mentioned below:

2.2.1.1 MapReduce

Map-Reduce

MapReduce is a type of Batch-Data processing was introduced by Dean et al. [12] 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 [5] and we chose it for our project due to its ubiquity and popularity. It broadly consists of 3 phases, namely the map, shuffle and reduce phases. as illustrated in Figure 2.2.

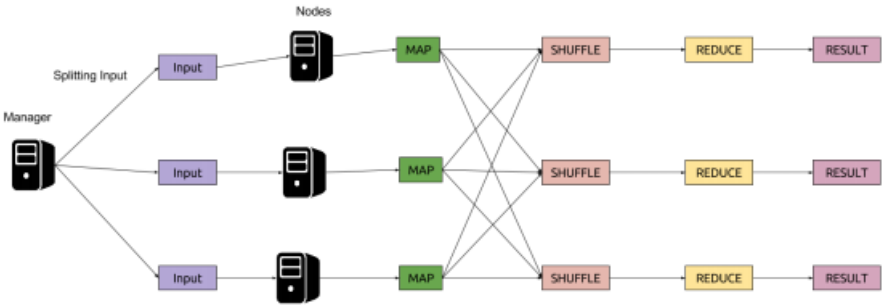

Figure 2.2: High level overview of the MapReduce Framework.

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 [12]. 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 [15] and HDFS [7].

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 [12]. 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 [12].
- On notification about these locations, the reduce worker reads the buffered data from the persistent memory store of the map workers via RPC calls [12]. 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.

2.2.1.2 Hadoop

Hadoop

Hadoop is Apache project used in various organization and companies as it is based on open source. Hadoop [38] is the framework which stores and processes large amount of data present in the clusters. These two functions (ie. storing and processing) are performed by Hadoop components namely Hadoop Distributed File System for storing purpose and MapReduce for processing purpose. In Hadoop there

exists a Master slave architecture in which the NameNode and JobTracker are the master node while DataNode and TaskTracker belongs to the slave nodes.

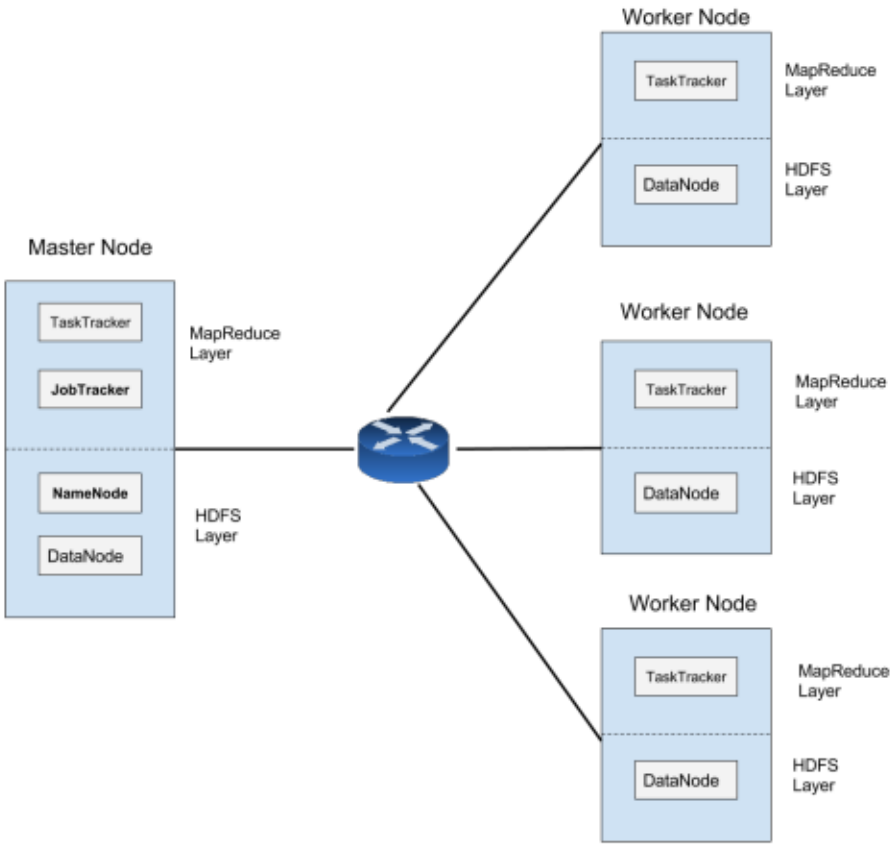

Figure 2.3: Overview of the Hadoop architecture.

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 [39]

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 [39]. It keeps track of the progress of every job, rescheduling it to other TaskTracker nodes in case of failure [39].

TaskTracker - TaskTrackers or worker nodes, run the MR jobs assigned to them by the JobTracker node

and report the progress back to the Jobtracker [39] node.

Most of the data transfer loads in a Hadoop cluster are attributed to the shuffing phase, when intermediate data from the mappers is shuffed over to the reducer nodes. Chowdhury et al. [9] analysed Hadoop traces from Facebook's Hadoop cluster and found that on an average, 33% of the runtime is consumed by the shuffe phase. Additionally, in 26% of the tasks with reduce jobs, more than 50% of the runtime is spent in the shuffe phase, while it accounts for upwards of 70% of the runtime in 16% of the jobs [9], confirming results reported in literature [4] which state that network is a bottleneck in MapReduce.

2.2.1.3 Spark

Spark [42] is also one of the batch data processing. It is sub project of Hadoop and it is advanced when compared to Hadoop and MapReduce. Spark is comparatively much faster as it works on 'in-memory cluster computing'. Apart from doing batch processing, it can also processess different type of jobs where stream processing is also supported. Apache Spark's supports many languages such as Java, R, Scala, Python.

Apache Spark has a incredible feature called Resilient Distributed Dataset (RDD) which makes it faster because it does not spend much time in reading/writing into HDFS, rather it does processing inside the memory and store the memory as object and shares it with other. Due to this there is reduce in the usage of time and it becomes faster when compared to Hadoop and MapReduce as they do the data sharing using the network and HDFS.

2.2.2 Stream-Processing

Stream-Data processing also processes high volume of data but the data in stream processing are continuous(i. e. real time). Like batch processing here the data must be processed in small time slots.

2.2.2.1 S4 (Simple Scalable Streaming System)

S4 was developed by Leonardo Nuemeyer et. al. [28] in Yahoo labs. The author was influenced by MapReduce and created S4. It not only handles batch data but also processes real-time data. In Simple Scalable Streaming System,there are two components which helps in streaming the data : *Processing Node(PN)* and *Processing Element (PE)*, where the Processing Node has Processing Element Container(PEC) in which there are number of Processing elements which is used for computation purpose and the Processing Nodes (PN) are used for listening to the events and as a dispatcher. Events are the stream data which need to be processed are routed to Processing Node using hash function and with

the help of the key. The communication Layer in the S4 used to handle the mapping done between the physical and logical node.

2.2.2.2 Storm

Storm is proposed by Ankit et. al. [34] which is another kind of the stream data processing used in Twitter. Storm is scalable,resilient,efficient and easy to administer. Storm was mainly designed for Twitter and real time data. Storm is based on master node and worker node, where the master node is called as Nimbus. Here the Nimbus does the same work as the Job Tracker in Hadoop framework. The user has to submit the topology to the Nimbus. And the worker nodes does the actual work which is allotted by the master node as in the MapReduce where the Task Tracker does all the work. Zookeeper is used as a coordination between the master node and the worker node and keeps track on them. In Storm there is a supervisor who receives assignments from the master node Nimbus and accordingly it attempts to distribute the task in a transparent manner to the worker node. Supervisor is supposed to check for the health of the worker node and if the worker node has failed then supervisor respawns it. The limitation of the Storm is that is does not provide automatic load balancing.

2.3 SDN for Big Data

SDN for Big Data is a big game changer that accentuates the big data analytics ability and the centralized network management. Big data application which are used to extract value and insights efficiently from a very large volumes of data. It provides the ability to orchestrate and dynamically create large *Hadoop* clusters on demand through *Centralized* controller. MapReduce is a big data framework which has been adopted to analyse the massive data. Normally these framework depend on manual configuration to acquire the network proximity information to improve the performance of the data placement and task scheduling. But manual configuration is cumbersome and sometimes infeasible in large scale deployment. Here is where the SDN comes into picture, it abstract the proximity information of the network [41] and the developer does not have to manually configure it. SDN and Big Data are the trends which revolutionize all aspects of modern life by leveraging to merge both of them to increase their value. SDN can make use of large amount of operational data to optimize the network behaviour, while Big Data can benefit from the flexible network resource [32].

Junaid Qadir et al., [32] has described about SDN and BigData as well where he has mentioned that these both trending technologies as the intersection point in the datacenter world. He describes that OpenFlow features provides different network characteristics to numerous categories of traffic in a Hadoop cluster such as it provides high priority to the shuffle traffic in Hadoop, it also provides high priority to the Hadoop traffic when there are other traffics in the network.

Abhijeet Desai et al. [13] has tried to solve the current problem of large datacenters or internet having enormous amount of data to process and control as well as the network complexity as it has been very difficult to measure the network characteristics. In earlier times as the Network administrators used to check the mentioned characteristics of the network manually, such as the Qos, bandwidth capacity. The author used SDN for abstraction and Hadoop for processing and proposed a new architecture named as "Advanced Control Distributed Processing Architecture (ACDPA)", which is used for resolving the flow characteristics and setting the priority for the flows based on QOS.

As SDN and BigData complements one another we show you how the BigData application uses SDN to give better results.

2.3.1 BigData Application Using OpenFlow

There are many number of techniques which uses OpenFlow to improve the transfer of data to the nodes with better bandwidth utilization. Some of them are discussed below:

Zhao li et. al. [23] proposed OFScheduler which is a network optimizer based on OpenFlow for improving MapReduce operations in a heterogeneous cluster. OFscheduler is aimed to increase the utilization of bandwidth and balance the workload of the links in the network. The author uses tag to identify different types of traffic and then OpenFlow tailors and transmits the flow dynamically.

FlowComb is a network management framework [11], which is also based on OpenFlow. It benefits Hadoop with increased in bandwidth utilization and decreased data execution times. The centralized decision engine present in FlowComb, collects data from software installed on application servers and decides, how to schedule forthcoming flows globally without congesting the network.

Qin et al., [33] have proposed a framework that employs SDN in Hadoop to decrease the time taken by data to reach the distributed data nodes from the mappers. To avoid the delay the author has developed a technique which eventually measures the current available bandwidth on the links through OpenFlow protocol and schedules tasks in a way which decreases the execution time. He named the proposed technique as 'bandwidth aware task scheduler' (BASS) which utilized SDN and it manages the available bandwidth of the links, and then allots into the time slot(TS), later BASS decides to allocate the task remotely or locally with respect to the completion time. The main plus point in BASS is that it does task scheduling, even when the OpenFlow controller has scanty amount of bandwidth to consider.

Sandhaya Narayan et al., [27] uses OpenFlow to control the network resources in Hadoop. The most

traffic generating phase in Hadoop is Shuffle phase where the intermediate data is moved from Mappers to the Reducers. Due to the heavy traffic in the Shuffle phase, which in turn makes the bandwidth link unavailable between Mappers and Reducers, causes delay in the Reducer. Consequently lowering the performance of the Hadoop cluster. So the author has used OpenFlow technology which is used in SDN to furnish better link bandwidth for shuffle traffic which simultaneously decreases the Hadoop job's execution time.

Wang et al., [35] have proposed a application-aware networking setup based on SDN controller with optical switching, in which the network configuration is made easier by using the integrated control plane in job placement. The job is scheduled in two ways *i)Bin packaging placement ii)Batch processing*. Due to application awareness of the network there is improvement in the big data performance by allocating and scheduling bandwidth, as well as the completion time of the job is minimized drastically. The author used Hadoop as an example of the big data application framework. The challenge which the authors faced during configuration of the network for Hadoop task was to handle the aggregated traffic in the mappers and reducers, which was resolved by ranking the racks based on the traffic demands and then the racks needed to be added in the tree topology in descending order of the demand.

Freguson et al., [14] has developed an API named PANE which is implemented on OpenFlow Controller that gives read and write authority from the network administrator to end user. With the help of PANE,the end users are allowed to work in the network , which means that the end user and their application can manage the network according to their need. It allows applications to contact the controller to request for resources or set up rules for future traffic. PANE solves two issues (i)It allows multiple end user (principal) to control the network resources (ii) It successfully solves the conflict between the end users requests, meanwhile it also maintains the level of fairness and security.

End Users (principal) in PANE which gets read and write authority from the network administrator can issue three type of message such as Queries,Hints, Request [14]. These three message are used by principal when the user wants to request for bandwidth or access control resources(Request Message),when user wants to learn about the traffic between host or available bandwidth (Query), whereas the last message notifies the end user about the current or future traffic characteristics(Hint).

Participatory Network restricts on the limits of the end user authority with the help of *shares*. Share mentions which end user can send which kind of message in the flowgroup, where the flowgroup refers to the messages in some of the network flow. A share has three segements: *principals, privileges and flowgroup* [14]. The authors demonstrate the feasibility of their proposed approach with four distributed applications such as Ekiga, SSHGuard, Zookeeper and Hadoop.

Chen Liang [24] made use of PANE API to estimate the gain in real distributed system such as Hadoop and zookeeper, where PANE played a major role of reserving the bandwidth for the applications. When there was other traffic in the same network as of Hadoop traffic, Chen Liang tried to to minimize the reservation for Hadoop. Before the actual traffic happens, suppose if the reservation is set up then it would be idle for some time and avoid other applications from using the bandwidth. So the author tried minimizing the ideal time of the reservation by analysing the place where the flows are created, and the PANE API makes the reservation before the traffic occurs. There was an issue popping up in the small flow request, which explains that when the reservations are made just before the traffic is about to happen, then there are chances of the traffic flow has already finished. Basically it ends up wasting time and space on creating an idle reservation. To address this issue, he came up with an idea to avoid reserving for small flows as it is possible to get already finished in less amount of time.

Xiaolin et al., proposed Palantir [41], which is a SDN service specific for distributed frameworks. It was build to abstract proximity information from the network for the distributed frameworks such as MapReduce, Dryad. Earlier the network administrator has to manually configure the network for the distributed framework but Palantir offered great help to the network administrator by defining and capturing the proximity information. To implement Locality Aware Task Scheduling and Rack Aware replica placement we need to observe the network proximity information, where Palantir allowed the distributed framework to express their definition of proximity. With the help of this definition the Palantir automatically abstracted the proximity graph from physical network topology. Palantir is leveraged on FloodLight controller [1] where it uses the existing device and topology manager of FloodLight to abstract the network information. Palantir had some concerns on scalability of the system so they tried to reduce the computational cost by caching the abstracted proximity domain for each registered framework in memory. When the framework corresponding to the Palantir was shutdown or unregistered then the cached results are deleted from the memory.

After the classification and description of the related work throughout this section, we now present a Table 2.1 which further shows a summary of various projects which leverages SDN to improve the performance of BigData applications.

2.4 Summary and Final Remarks

We have come to the end of the chapter, here we present the crux of this chapter. After going through section about SDN and BigData, we saw how we can use both technology to get better results in the data center network. We presented the some of the work already done by other researcher. The criteria we have present in Table 2.1 is what we have been discussing throughout this chapter. We are comparing the results whether the projects already researched utilizes the bandwidth and latency. But many of the projects like OFScheduler [23], FlowComb [11], BASS [33], PANE [14] just either utilizes the bandwidth or latency. But we intended to utilize both bandwidth and latency for the job scheduling.

[1] http://www. projectfloodlight. org/floodlight/

Project Name	Integration	Latency Utilization	Bandwidth Utilization	Result
OFScheduler [23]	OpenFlow & Hadoop	No	Yes	Incereased bandwidth utilization and improved performance of MapReduce by 24-63% in multipath heterogeneous clusters.
FlowComb [11]	OpenFlow & Hadoop	No	Yes	Improved resources utilization and low data processing than(reduction in time by 35% to port sort 10 GB randomly generated data) of Hadoop.
Programming Network at run time for Big Data [35]	SDN,Optical circuit & Big Data	Yes	Yes	Jointly optimize Big Data application performance and network utilization
Hadoop acceleration in an OpenFlow based cluster [27]	OpenFlow & Hadoop	Yes	No	Shuffle phase time reduction and improved application (Hadoop) performance with user having ability to control the network
Participating Networking [14]	OpenFlow & Big Data applications like Hadoop	No	Yes	improved network management with the help of an API for distributed applications (like Hadoop) to control SDN
BASS [33]	OpenFLow & Hadoop	No	Yes	Efficient bandwidth allocation with the help of an OpenFlow and optimized execution time due to bandwidth aware scheduling.

Table 2.1: Summary of various SDN projects that have been proposed to improve Big Data application performance [32]

There is a project [35] which tried to utilize both but it was not utilizing the CPU during its job scheduling and it was more keen towards flow scheduling.

Chapter 3

MR-Flood

We now describe our solution *MR-Flood*. We start this chapter by providing a use case for our system, in the upcoming paragraph. Then, in section 3.1 we provide a detailed architecture explanation, mainly on the interactions between the component of our thesis solution. In section 3.2 we go in-depth to the components presented in the previous section, and describe the internal working of each one. Then, in section 3.3, we summarize this chapter.

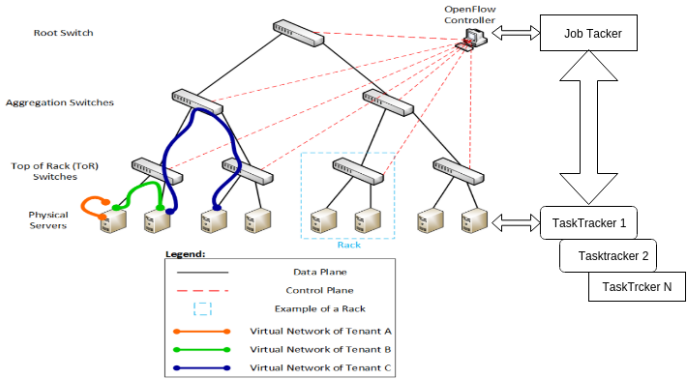

Figure 3.1: High level architecture of the solution with a simple data center tree topology.

In Figure 3.1 we show the high level architecture of our solution, which is inspired by ViTena [8] in this use case we are also using a simple tree topology with depth equal to 3 and fanout equal to 2. Most of our work will be similar to ViTena [8] but with the changes due to the inclusion of MapReduce job scheduling in the VMs which is allocated as described in ViTena [8]. Here we are using the traditional topology to have a better understanding of our architecture. The OpenFlow controller is the main component of our solution, as it is responsible for running the ”Latency and bandwidth aware job scheduling” algorithm in order to map the requests on the physical network, and then dynamically program the switches to deploy the requested task. We also introduce MapReduce emulator (rightmost) which is another component of this solution and it consist of Job Tracker and Task Tracker. It emulates as MapReduce framework.

Now we explain about the Openflow switch where all the switches acts as a packet forwarder as they do not have any intelligence to forward the packet, it listens only to the controller and acts according to the rule described by the controller. The controller is logically connected with all the switches in the network, which is depicted by a dashed red lines that becomes the control plane. They are only logically separated but not physically separated from each other.

Our algorithm ”Latency and bandwidth aware job scheduling” tries to schedule the batch of task on the smallest available subset of the physical network (on the same physical server, then on the same rack, and so on). Our aim to maximize the proximity of Batch of task in the VMs belonging to the same tenant, which results in minimizing the number of hops between those VMs, in turn it reduces the latency and having the task placed in proximity i.e to be in the same server or in different rack, this saves the bandwidth usage in the upper links of the tree, which helps in saving the bandwidth in the upper links as the network is scarcer in data center [6]. By this we can allocate more number of task without making the network links to become bottleneck.

Now we explain you the Figure 3.1, where we show you an example of three virtual network placements according to algorithm presented in ViTena [8]. The virtual network of tenant A represents the best case possible where all the batch of task can be mapped on the same physical server. In this, there is no usage of the network due to which bandwidth is saved for future requests of the task allocation. Now in other situation where the batch of task is split into two cases based on latency and bandwidth oriented request such that the given batch of task cannot be mapped into the same server. For instance let us assume that the task is latency oriented and the request cannot be mapped into a single server then we try to allocate the request in other servers belonging to the same rack as shown in the virtual network of tenant B this drastically reduces the latency. In the second case we assume that the request is bandwidth oriented then our algorithm looks for the server which are scattered apart to allocate the request (i.e. not in the same rack) as shown in the virtual network of tenant C to ensure that bandwidth and latency are not compromised.And the nearest free server are kept for the future task which are latency oriented.

Our algorithm ensures that the network, can provide the bandwidth and latency guarantees requested by each Batch of task Apart from this we also use the centralized information in the controller to provide two properties not seen in the systems we have surveyed, those are the fair bandwidth sharing among tasks and reduction of latency for the Batch of Task requests. Fair bandwidth sharing is achieved by instructing every switch used by a virtual network to follow the QoS disciplines by creating a new queue for that task. Secondly the reduction of latency is achieved by ensuring latency and bandwidth for batch of task algorithm, by placing the task using best fit strategy. To know the heuristics we need to abstract the current network from the OpenFlow controller.

3.1 Architecture of the Proposed Solution

Now we describe in greater detail about the architecture shown in the previous section. Figure 3.2 is the similar to the previous architecture but here it shows in greater detail about the component of our project. Thus, in Figure 3.2, it contains the software that will be used for the exectuion of our alogorithm.

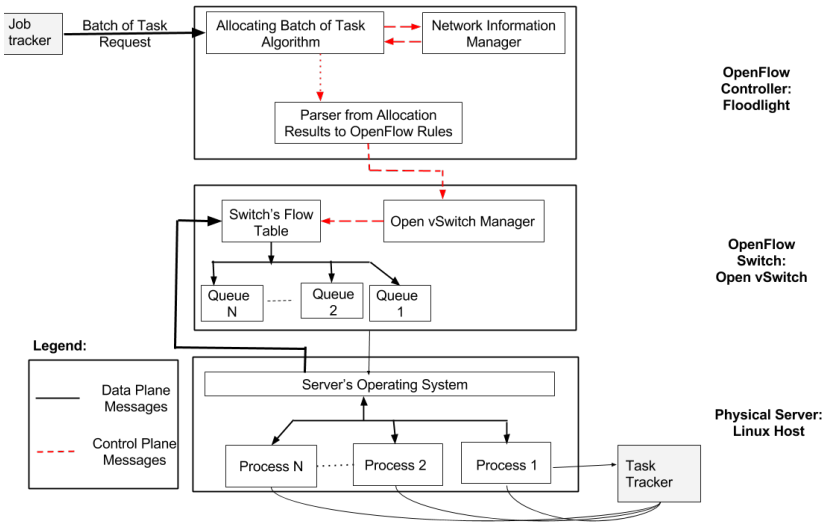

Figure 3.2: Software architecture of the components present in the solution.

Let us justify our choices regarding the components shown in Figure 3.2. For the OpenFlow controller, we will use Floodlight for two reasons: first because it is based in Java, which as stated earlier gives the highest performance; and second because it is build using Apache Ant,which is very easy and flexible in use. Regarding the other components, Switch and Physical Server, the choices are Open vSwitch and Linux Host (respectively) because this solution will be implemented within the Mininet emulator, that uses those components. We will use Mininet because it is open-source and also because it is used by the MRemu tool [30]. Our solution can be executed in the real data center network with or without any changes. The only exception is the Linux Host, where there would be a hypervisor running and it would manage the VMs inside the host machine. We make an assumption to simplify our solution that all physical server have the same CPU frequency so the tenant asks for the percentage of CPU instead of the CPU frequency.

Now we depict the flow of information in the system which uses our algorithm. First of all, the BigData application's task expresses its demands in a Batch of task request (e.g. an JSON file). This consists in defining two things: the number of VMs required (and also the percentage of CPU of each one) as well as the bandwidth required between the VMs that will be connected (expressed in MBit/s). The batch of task request is fed into the ensuring latency and bandwidth for batch of task algorithm (detailed in the next section), that is running in the OpenFlow controller. Upon receiving the request, the algorithm contacts the network information manager to get the current state of the network. Based on this state,

the algorithm determines (if the request is accepted) where this batch of task will be allocated. One important thing we should know is, all the requests are processed by the controller, this updated view of the network involves zero control messages over the network, since the controller just has to update this information when it processes a new task allocation request.

The controller then translates the result of the algorithm (i.e. the affected switches and hosts) to OpenFlow rule(s) to reprogram the switch(es). Upon receiving this message, the Open vSwitch manager takes two actions: creates a new queue for the batch of task, with the minimum bandwidth present in the received message; and installs a new rule in the switch's flow table to forward packets from that batch of task to the newly created queue. This means that there will be one queue for each batch of task passing on that switch. In this way, a queue (Batch Of Task) can use more bandwidth than its minimum when the other queues are not using it.

The bandwidth share between queues is made fairly according to the minimum bandwidth a queue has. Thus, the resource usage is maximized, as the tenants share unused bandwidth in the fair manner, and they also get their minimum bandwidth guarantee when the network is saturated. Suppose if the network is saturated, packets may be dropped at the switch. This ensures performance isolation at the network level. As the Open vSwitch is based on the Linux kernel, we will use the traffic control (tc) utility HTB (Hierarchy Token Bucket) to make the configuration of the queues.

We will represent and implement Batch of task as processes. To simulate tenants workloads, each process will be running a traffic generator. As depicted in Figure 3.2, upon the necessary configurations, each process (i.e. Batch of Task) can communicate with other processes on the same virtual network, using either the operating system (in case the processes are on the same server), or contacting its adjacent switch which will use the flow table to check to which queue it should forward this solicitation (in case the processes are on different servers). To make this distinction, each physical server needs to keep a list of the processes (and the corresponding virtual network of each one) it owns.

3.2 System Components

After the discussion about the detail architecture of our solution described in the last section, we now focus on each component of our solution. As we can see in Figure 3.2, our system has three main components: MRemu; the OpenFlow controller; the OpenFlow switch; and the physical server. In the following subsections, the functioning of each component is defined in greater detail.

3.2.1 MapReduce Emulation - MRemu

Using the real hardware for running data centre experiments is not an easy option for many researchers, since such resources are not readily available. We went through 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. [30]. 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 [30] such as Mininet, as described in 2.1.2 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 [9].

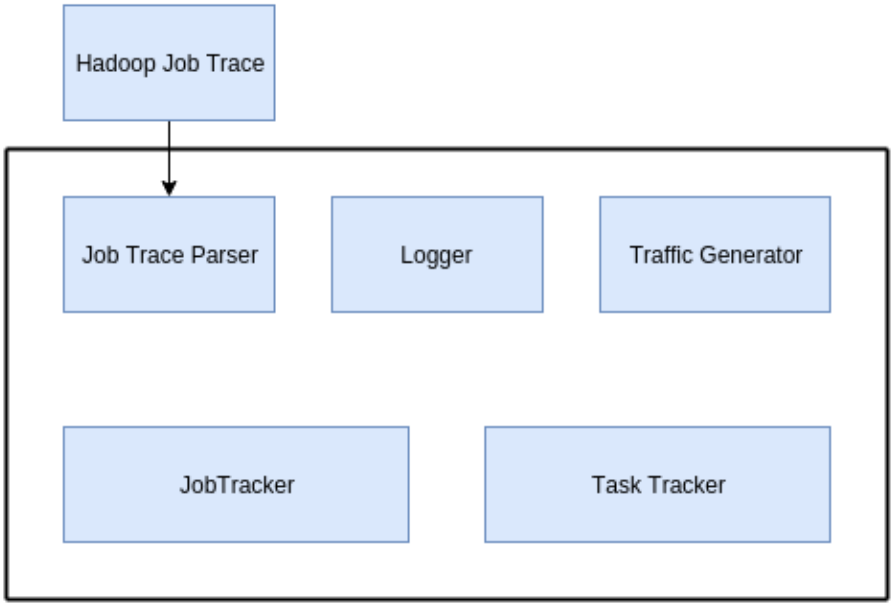

Figure 3.3: Block Diagram of MRemu Hadoop Emulator. [30].

There are numerous studies which uses synthetic traffic patterns, generated by following certain probabilistic distributions [10], which fails to capture the true network workloads. 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. The traces produced by Neves et al., are available online [29], have been employed by us in this project with many modifications in them so that it can offer the solution we needed.

Figure 3.3 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 2.1.2, for Hadoop traffic scheduling experimentation. MRemu supports production SDN controllers such as POX/NOX [16] and OpenDayLight [26], making it ideal for evaluating novel approaches to route Hadoop traffic in order to accelerate it. But Floodlight was not supported so we had to make changes to make it work. Neves et al. [30] 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 [19] 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. However, Job Completion times were found to be near accurate, owing to which, we chose to use MRemu for evaluating our bandwidth and latency aware scheduling approach as described in Section 4.

MRemu has certain limitations, such as MRemu supports emulation of only one job at a time, having no support of concurrent job execution. But we overcame this limitation to match the requirement of our implementation. The other limitation MRemu had was it could 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. This also was solved by us by emulating virtual network in the floodlight controller.

After many modifications MRemu met our requirements for Hadoop emulator in the data center network, hence we made a choice to use it in our thesis.

3.2.2 OpenFlow Controller - Floodlight

Here comes the OpenFlow controller ”Floodlight”, which has played the main role in this thesis to fulfill the implementation of our solution. The Floodlight controller is the component where the main intelligence, of how to ensure the bandwidth and latency aware job scheduling exists. Meanwhile we present the explanation about our bandwidth and latency ensuring algorithm. We also emphasize about the data structure used in our solution with respect to the algorithm depicted below. Finally we explain the decision of how the job is allocated on the nodes by guaranteeing the parameters discussed above.

3.2.2.1 Ensuring Latency & Bandwidth For Batch Task Algorithm

Now we present to you our pseudo-code for the latency and bandwidth aware job scheduling which had some modification to the ViTena [8] where the algorithm will be running inside the FloodLight OpenFlow controller. Already we have discussed about our algorithm but now in this section we discuss it in detail for more better understanding.

Algorithm 1 Ensuring Latency & Bandwidth For Batch Task Algorithm

Input: BOT(Batch Of Task)
Output: `True` if request is accepted,`False` otherwise.

```
1:  totalBOtaskLoad ← getBOtaskFromBOT(BOT)
2:  highestCPUAvailable ← getMostFreeCPU()
3:  if (totalBOtaskLoad < highestCPUAvailable) then
4:      appropriateList ← findAppropriateList(totalBOtaskLoad)
                          ▷ Maintains N sorted lists that segments server with free CPU% [0-(100%N)∗i],0<i<N
5:      for all (server in appropriateList) do
6:          serverCPUAvailable ← getCPUAvailable(server)
7:          if (totalBOtaskLoad < serverCPUAvailable) then
8:              allocTask(BOT,server)
9:              return True
10:         end if
11:     end for
12: else
13:     server ← getMostFreeServer()
14:     firstServer ← server
15:     while (True) do
16:         [preAllocatedBOTs,remainingBOTs]←splitRequest(BOT,highestCPUAvailable)
17:         BOTsAlreadyAllocated ← (preAllocatedBOTs,server)
18:         preAlloc(preAllocatedBOTs, server)
19:         server ← getNextServer(server)
20:         if (server.isFirst()) then
21:             cancelAllPreAllocs()          ▷ Cancelling all pre allocations
22:             return False
23:         end if
24:         CPUAvailable ← getCPUAvailable(server)
25:         [preAllocatedBOTs,remainingBOTs]←splitRequest(BOT,CPUAvailable)
26:         if (remainingBOTs.is Empty()) then
27:             allocAllPreAllocs()
28:             return True
29:         end if
30:     end while
31: end if
32: return False
```

As stated earlier, the Ensuring Latency & Bandwidth for Batch of Task has to guarantee that each Batch of task gets (at least) the requested bandwidth and latency.To explain the rationale behind the algorithm, we will divide it in two cases: when the batch of task request fits in on physical server and when it needs to be spread across multiple servers(algorithm 2). This is the first check made, comparing the total CPU load requested with the highest CPU available at the moment (algorithm 1:line 3).

If the request fits in one server, we want to find the server with the least free CPU that fits the request (i.e. best-fit). After finding that server, we allocate this request on it (which includes updating the network state with this new request). To find best-fit server, we first find in which list we will search (algorithm1:line 4). We will maintain 10 lists: the first keeps the servers with 0 to 10 % of CPU free, the

Algorithm 2 Splitting The Request Based On Bandwidth &Latency

```
1:  procedure splitRequest(BOT)
2:      BWBasedBOT←sortBOTbyBWDemands(BOT)
3:      latencyBasedBOT← sortBOTbylatencyDemands (BOT)
4:      if (BOT is BWOriented) then
5:          cpuAvailableInServers ← listOfFreeScatteredServer()
6:          preAllocatedBOT←preAllocate(BOT,CPUAvailbaleInServers)
7:          BWDemands←SumOfDemand(BWOriented,BOTsAlreadyAllocated,
                PreAllocatedBOTs,servers)
8:          linksResidualBW←calcResidualBW(BOTsAlreadyAllocated, server)
9:          if (BWDemands > linksResidualBW) then
10:             clearLastPreAllocatedBOT()
11:             return NULL
12:         else
13:             preAlloc(preAllocatedBOTs, servers, BWDemands)
14:             BOTsAlreadyAllocate ← BOTsAlreadyAllocated + (preAllocatedBOTs, servers)
15:             return
16:         end if
17:         return
18:     else                                                    ▷ BOT is latencyOriented
19:         cpuAvailableInServer ← listOfNearestFreeServers()
20:         preAllocatedBOT←preAllocate(BOT,CPUAvailbaleInServer)
21:         latencyDemands ← calcSumOfDemands(latencyBasedBOT, BOTsAlreadyAllocated,
                preAllocatedBOTs,server)
22:         linksResidualLatency ← calcResidualLatency(BOTsAlreadyAllocated, server)
23:         if (latencyDemands > linksResidualLatency ) then
24:             clearLastPreAllocatedBOT()
25:             return NULL
26:         else
27:             preAlloc(preAllocatedBOTs, server, latencyDemands)
28:             BOTsAlreadyAllocated← BOTsAlreadyAllocated + (preAllocatedBOTs, server)
29:             return True
30:         end if
31:         return
32:     end if
33:     return True
34: end procedure
```

second the servers with 10 to 20 % of CPU free, and so on. This is just to reduce the search space for the appropriate physical server, which in a large data center environment can reduce the run time of the algorithm considerably.

If the request does not fit in one server, we call splitRequest(..) procedure (algorithm 1:line 16)which is defined in Algorithm 2**(Splitting The Request Based On Latency & Bandwidth)**,where we sort the request by decreasing bandwidth and the latency (algortihm 2:line 2 & 3),then we check for the Bandwidth based batch of task (algortihm 2:line 4-6),now we find the list of free scattered server to allocate as much BOT as possible in the free server on the entire data center. In this way, the most consuming Bandwidth demands are on the scattered physical server, which significantly does not affect the bandwidth demands,moreover it ensures that the nearest free server cpu is available for the latency based BOT.

In the next few lines (algortihm 2:line 7-11), we see if the server we are checking has connection(s) with sufficient bandwidth (as defined in the request) to the other server(s) already pre-allocated in previous iteration(s). If it does not have enough bandwidth, we try on the next server by clearing the last pre allocated BOT i.e no commit (lines 9-12). If it does, we pre-allocate the BOT mapped onto this server.

Similarly when the BOT is latency based task (Algortihm2:line 3) then it goes to (Algortihm2:line 19) to find the list of nearest free server to allocate the task so that fetching or storing the latency based task does not increase the latency. In the next lines (Algortihm2:line 20-22), we see if the server we are checking has connection(s) with other server which would provide reduce latency (as defined in the request) already pre-allocated in previous iteration(s). If it does not reduce the latency we try on the next server by clearing the last pre allocated BOT i.e no commit (Algortihm2:lines 23-25). If it does, we pre-allocate the BOT mapped onto this server.

Now as the important decision are made based on bandwidth and latency we jump back to (Algorithm 1:line 16)where the splitrequest() returned True if preAllocated BOT was allocated.Finally, we check if the set ”remaining BOT” to be allocated is empty (Algortihm1:lines 26-29): if it is we allocate all the pre-allocations (i.e. commit) and return True; if it is not, we move on to the next server to allocate the remaining ones.

After pre-allocating (since the request can be rejected) what is possible on the most free server, we try to allocate the rest on adjacent servers.In line 19, the getNextServer(*server*) function returns the servers on the same rack of *server*, then the servers on an adjacent rack, and so on. Next we check if we already tried on every server of the data center (line 19), and reject the request if so (cancelling all the pre-allocations).

As we can see by the description of the algorithm, we will have the batch of task which are stored based on bandwidth and latency in the servers, since the algorithm either allocates the task on the basis of their respective needs (if the request does not fit one server), or finds the best-fit server (if the request fits in one server).

3.2.3 OpenFlow Switch - Open vSwitch

As described earlier in section 3.1, we are using Open vSwitch (OVS) as one of our system component to include virtualization in the networking architecture. OVS is virtualized switch which is adaptive with the Linux kernel. The main component of OVS are ovsdb-server, ovs-vswitchd, OVS- Kernel module.

- **ovsdb-server** - It is the database that has the configuration of the switches. It uses OVSDB protocol to communicate with the ovs-switchd.
- **ovs-switchd** - It is the core of the OVS and it communicates with the OpenFlow for switching purposes.
- **OVS-Kernel module** - It is situated in the Kernel space and it communicates with ovs-switchd using netlink.

As it is deployed in Linux kernel, it uses traffic control (tc) for queuing mechanism. Tc is the one which decides whether to accept the packets at a particular from the ingress queue and it also decides which packet should be transmitted with a particular rate at the egress queue.

The queuing disciplines (qdiscs) in Linux has many different types such as FIFO (First In First Out), PFIFO (Priority FIFO), SFQ (Stochastic Fairness Queuing), TBF (Token Busket Filter), HTB (Hierarchical Token Bucket). PFIFO is the default qdiscs and it operates with priority. But in our sollution we want to limit the total bandwidth to a particular rate and this is only done by TBF and HTB.

TBF is good example for shaping the traffic as it tamsmit the packets only if there are sufficient tokens available. Otherwise it waits until the bucket is filled with tokens and sends the packet when the bucket has enough tokens. TBF has a child class and it is known as HTB. It is best known for controlling the traffic and limiting the bandwidth rate as specified.

In our solution We use HTB as our qdiscs, because it is the only queuing discipline which limits the bandwidth of a particular user. HTB also does borrowing mechanism where the child borrows the tokens from their parent to bring the rate to expected rate and start the transmission of the packets. We have designed a HTB tree with one root so that all the leaf are at same level. And they can share the

bandwidth with each other queues. We do not use ceil rate as we are configuring from the OpenFlow controller, so we use only Assured rate. This makes the client to use the full bandwidth when the other clients are not using the network link.

3.2.4 Physical Server - Linux Process

The physical server which we are using for our experiment is Linux server. And as we have used Mininet as our emulator it creates a data center network, using physical server as a Linux process. In the real data center, all the server would be running an hypervisor software, which manages the physical resources to the Virtual Machines that were allocated on that server.

In our experiment we emulate the server as a Linux process, and then the VMs are emulated as a child process to that particular server (parent process). We use & operator to create a child process with their parent and it runs in the background. All this child process is run with different port.

We have depict the physical sever and the VMs using mininet but we cannot be sure if this would guaranteee better performance in the real world scenario as well.

3.3 Summary and Final Remarks

In this chapter we have presented our solution. We began by showing a common use case for our system, then described its architecture in detail, and we finally described each system component and how they function internally.

We have developed an algorithm that does job scheduling of batch of task(incrementally) using the virtual network, it also provides bandwidth guarantees in a work-conserving system. This algorithm is also can be scalable to the Data Center networks, as depicted in section 3.1 .

We intend to prove the claims made throughout this chapter in the Evaluation section, showing that we achieve the desired properties through emulation runs.

Chapter 4

Implementation

The previous chapter described our proposed latency and bandwidth aware job scheduling algorithm which *ensured bandwidth and latency guarantees* for the tasks allocated. In this chapter we bring to you the implementation of our algorithm. From the previous chapter, we know that, we used Floodlight as our OpenFlow controller and MRemu [30] as our MapReduce emulator. Thus, in section 4.1 we describe the modification we made in the MapReduce emulator (MRemu) to make it work with our floodlight controller. Then in section 4.2 we describe the main part of our implementation, which consists of the modifications and extensions that we have made to the open-source Floodlight controller. And atlast in section 4.3 we present Mininet topology, with this we create a network of links and switches to connect to the controller and test it for our use case. In the end, we conclude this chapter by emphasizing some important aspects and some remarks about our implementation.

4.1 MapReduce emulator - MRemu

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 [30], as discussed in detail in 3.2.1. 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 work flow of the Hadoop emulation with the changes we made.

To use MRemu with our Floodlight controller we have to mention this in our StartUp script that the remote SDN remote controller will be running on the IP address 127.0.0.1:6633, which is used for ensuring the bandwidth and latency guarantees for the jobs provided by the tenants.

In the figure 4.1 we present you the block diagram of Mremu with the modification in the modules to support our implementation. The modified modules are in yellow color. Now, we will discuss about the changes we made in the module.

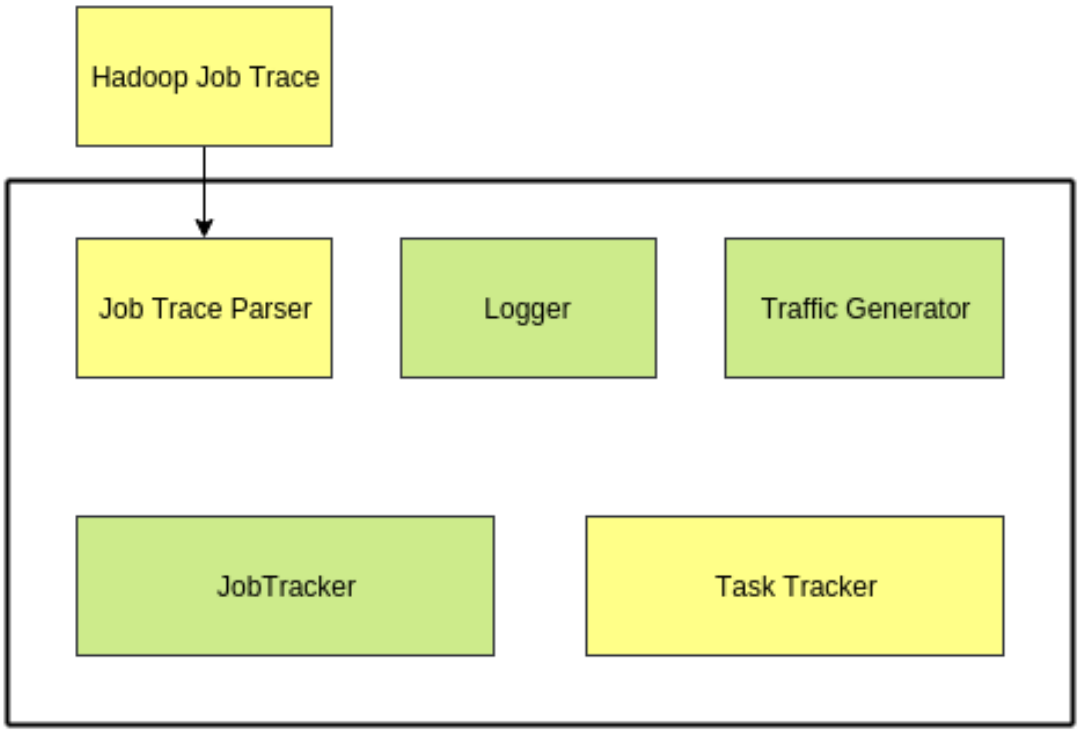

Figure 4.1: Block Diagram of MRemu architecture with modifications.

4.1.1 Concurrent Job Execution

The first and foremost change we made in MRemu was to train it to accept ***concurrent job execution*** as it had the limitation of supporting one job at a time. This would be not suitable for us, as we need many number of task to be executed. So, we fixed this limitation by using Python's Multiprocessing library, to have concurrent job execution. We needed concurrent job execution in our implementation as we have to check our algorithm until the links get saturated, to which we have to keep supplying the task until our algorithm returns us *false*, which means that there is no more server to accommodate the job execution.

4.1.2 Hadoop Job Traces

The Hadoop job trace was not sufficient and suitable for our implementation as it supported only 16 nodes as explained in 3.2.1. The main disadvantage was that, it already predefined in which host the task tracker should run *("host": "172.27.102.6")*, for the task submitted by the job tracker. This would not be suitable for us. As we need to dynamically assign task to host which guarantees the bandwidth/latency.

This made us to modify the Hadoop Job Trace. We created a JSON script which would give us the batch of task. These task can be MAP or REDUCE task which depends on the number of maps/ number

of reduce task mentioned in the same JSON file. While creating the batch of task we eliminated the host member, which predefined the host to perform the task in that particular host.

Apart from this we also added two members in the ”tasks” in the figure 4.2, they are *CPU* and *bandwidth*. This is the major change which we have made to make our algorithm work. Here the CPU and bandwidth have their own value described, through which our ”Bandwidth and latency aware task scheduling algorithm ” will parse the data to provide the perfect match as mentioned.


```
{
    "config": {
        "jobTracker": "172.27.102.31"
    },
    "finishTime": 1385953779.558,
    "name": "job_201312012205_0002",
    "numMaps": "384",
    "numReduces": "48",
    "startTime": 1385953633.608,
    "submitTime": "1385953633.578000",
    "tasks": [{

            "finishTime": 1385953773.031,
            "cpu": "3.4",
            "name": "attempt_201312012205_0002_r_000015_0",
            "bandwidth":100,
            "processingTime": "15.560000",
            "shuffleFinished": "1385953757.134000",
            "sortFinished": "1385953757.471000",
            "sortingTime": "0.337000",
            "type": "REDUCE",
            "startTime": 1385953642.713,
            "waitFinished": "1385953648.982471",
            "waitingTime": "6.269470"
    }],
    "transfers " : [{
            " dstAddress " : " 172.27.102.16 " ,
            " dstPort " : 59861,
            " duration " : 2.140714136,
            " finishTime " : 1388441782.972 ,
            " mapper " : " attempt_201312301708_0016_m _000014_0 " ,
            " reducer " : " attempt_2013 12301708_0016_r_000003_0 " ,
            " size " : 62584054,
            " srcAddress " : " 172.27.102.1 ",
            " srcPort " : 50060 ,
            " startTime " : 1388441780.831286
    }]

}
```


Figure 4.2: Hadoop job traces with modification.

4.1.3 Job Trace Parser

Job trace Parser is a module in the MRemu, which loads the Hadoop job trace and parses the information present in it. As we made changes in Hadoop job trace, we need to modify the readtask() method so that it reads the CPU and bandwidth values. Then the information is used from job trace parser to emulate hadoop jobs. Then in turn the Mremu emulator run the JobTracker and the TaskTracker.

4.1.4 JobTracker

Our algorithm gets the batch of task from the JobTracker to run the emulation. As we mentioned before, the host is not mentioned in the batch of task in prior, so the JobTracker queries the OpenFlow controller (Floodlight) with the batch of task to assign a host to that particular task. The query is made through **REST API** of floodlight mpdule And our algorithm starts running when it receives the batch of task.

4.2 OpenFlow controller - Floodlight

The OpenFlow controller, Floodlight is the main component of our solution, so majority of implementation is made in Floodlight controller. Floodlight controller is written in the Java[1] programming language. The modular architecture of the Floodlight controller is shown in Figure 4.3. The original architecture[2] is composed by the modules in blue. The other modules are which are in pink is used rigourously and the yellow colour module is built from scratch.

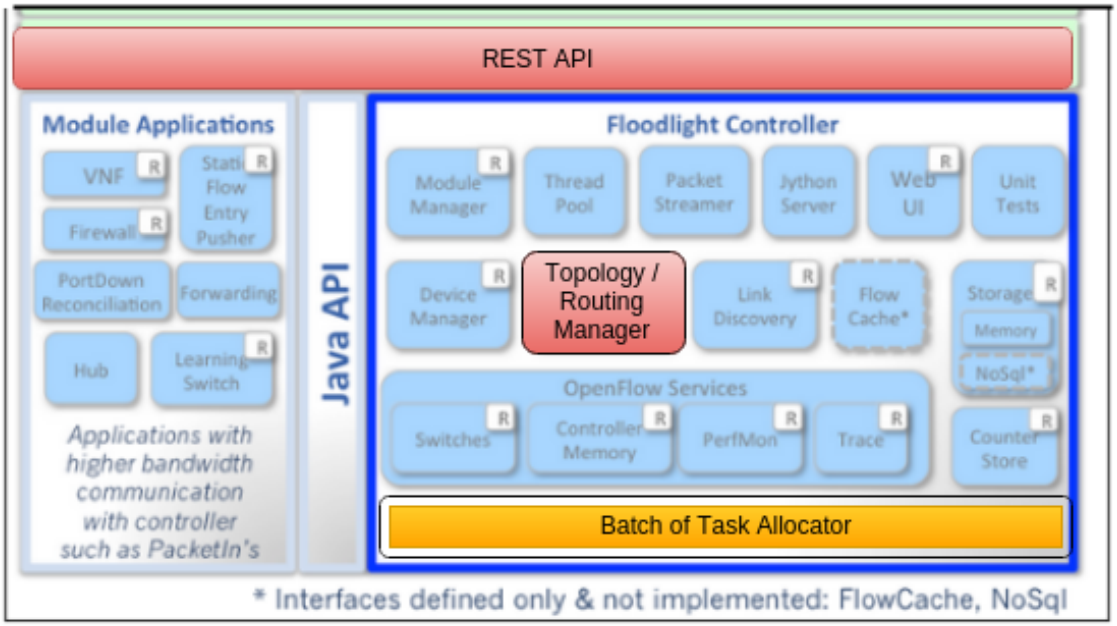

Figure 4.3: Modular software architecture of the Floodlight controller, and our extensions to it.

[1] https://www.java.com/en/ (Last Access: 15-10-2016)
[2] https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Architecture (Last Access: 10-10-2016)

The Floodlight controller is based on an event driven architecture, which means that modules must subscribe to the messages they want to receive, for example, in order for a module to receive an OpenFlow PacketIn or whatever message, the module must subscribe for this type of messages. When the controller receives an OpenFlow message from a switch it will dispatch that message to all modules that have subscribed for that specific message type.

When controller receives multiple messages from one or more switches, these messages are enqueued so that while dispatching the controller only supports one message at a time. This means that the controller's performance is dependent on the time it takes to process a message in each individual module, as the processing time of a single message is equal to the sum of all the processing time done in each individual module.

This step has to be followed in order to add a new module to the Floodlight controller, one must create a module of his choice and name it. Then code base of the controller a new Java package. In order for a module to start up it must be added to a configuration file that defines which modules are launched when the controller is started. This is done by adding the module's path to the `floodlightdefault.properties` file. When the controller starts, it will start the different modules, and set references between them according to the inter-module dependencies. These dependencies are defined by implementing the appropriate methods (according to which dependencies one wants to set) of the `IFloodlightProviderService` interface.

This figure 4.3 shows Floodlight's modular architecture. In this Figure, we can see modules that are displayed with three different colors. The modules in blue are the original Floodlight's modules that we simply use. The module in pink (REST API, Topology/ routing manager) is also one of the original modules of the Floodlight controller, but this is used as per our implementation, since we needed a different behavior than what was originally implemented in the controller. Finally, we have the yellow module, which are the modules that we developed from scratch and weren't part of the original Floodlight architecture.

4.2.1 Batch of Task Allocator Module

This is the main module of our solution. This is the module where we actually fulfill our goal to allocate the batch of task in the host. Our module keeps waiting for the task to be submitted through our REST API. When the task it submitted it processes them serially unless all the batch of task are over. Once you receive `False` from the module, it means that no more request can be handled as all the server are busy. While on its infinite loop, this module is waiting for Json parsed files.

In Fig 4.2 we can see a REDUCE task is requesting for the host to get the task done. Here in the *"tasks"* array we have included CPU, bandwidth values which is the main requisite for getting a host to

fulfill the task allotted. This means that the task need **"CPU:3.4%"** where 3.4% is already present and the bandwidth the task wants is described as **"bandwidth:100"**. Now let us see how the request is accepted. We have two cases:

- **When the request fits in one server,** i.e. the CPU requested can be accommodated in the same physical server, the processing of the request is fairly simple. In this case, this module does not need to use any of the modules described earlier in this chapter.
- **When the request does not fit in one server,** i.e. when this request has to be divided among two or more physical servers, this module will start splitting and enquiring the task if it is bandwidth based or latency based task. This Latency based task and bandwidth based task is the global data structure, which is set by us before the program starts. For example we set **Bandwidth-Operation-mode = 1** and then start the controller. This would treat all the incoming batch of task as the bandwidth based task.

  Now when the task are treated as bandwidth based task it chooses the host which are far apart. This is done with the help of routing manager. The routing manager calculates all the paths using YEN algorithm and the method used to get the scattered host is **getpathslow(link utilization)**. This method would return all the computed path between all the host but in reverse order by placing the longest cost path.

  Suppose if the **Bandwidth-Operation-mode = 0**, then the incoming task are treated as the latency based task. Here the same routing manager is used to calculate the path. Here also we use the same algorithm and same method but we follow a different parameter. **getPathSlow(hop count)**, the parameter hop count would give the number of hops, with which we can decide the less hop paths. Once all the task are successfully placed in the host and processed the algorithm will return *True*.

4.3 Network Topologies - Mininet

In the previous section we presented to you our implementation of the main module. Now we need to discuss the network topology on which the network was emulated for the SDN controller. We use Mininet to emulate the the data center network. We make use of the already developed Mininet script [8] for tree topology and fat tree topology.

We have used two `python` scripts [8] that use Mininet's API and create two different network topologies: the first creates a Tree topology, as the one shown in Figure 3.1; whereas the second creates a Fat-Tree topology, as the one described by Al-Fares et al. [3]. The first script receives two parameters: depth and fanout. The depth specifies how many levels the tree will have (counting from the root), and

the fanout details how many children each node in the Tree structure has. The Fat-Tree script receives only one parameter, named k, which defines how many ports each switch has. Then, in a Fat-Tree network, the number of connections to a node's father is the same that are going to that node's children (e.g. with $k = 4$, an Top of Rack switch would connect to two hosts and two aggregation switches).

4.4 Summary and Final Remarks

Throughout this chapter we discussed *MR-FLOOD*'s implementation. We described in detail the how we implemented the Open Flow controller, Floodlight, as well as the work we have done with MRemu.

While describing the modules we have developed, we also state which part of the architecture defines the module being implement, and, when appropriate, we show how we translated the generic solution presented in chapter 3 to the actual implementation.

Chapter 5

Evaluation

In this chapter we show you the evaluation we have made to our implementation described in the last chapter. In section 5.1 we describe about our environment in which we have made the evaluation and in section 5.2 we show the goal which we have evaluated for our thesis.

5.1 Evaluation Environment

5.1.1 Test Bed

Our experiments were run in a machine with the following hardware specifications:

- **CPU:** Intel® Core i5-2410M processor @2.3GHz
- **RAM:** 8 GB DDR3 @ 1333 MHz
- **HDD:** 500 GB Serial ATA @ 7200 rpm

The relevant software installed in this machine was:

- **OS:** Ubuntu 12.04 LTS (Linux Kernel 3.13.0)
- **Network Emulator:** Mininet - version 2.2.1
- **OpenFlow Switch:** Open vSwitch - version 1.4.6
- **OpenFlow Controller:** Floodlight - version 1.2

5.1.2 Benchmark Traces

The traces which we are using to generate traffic in our experiment were obtained from the MRemu github repository with modification explained in previous section. Neves et al. [30] used the traces which was obtained from the HiBench benchmark suite. [19]

- WordCount - This is the most common application used during the MapReduce evalution. It has a text input data where each words are counted when they occur. The traces had 50 GB of input data.

- TeraSort - It is an application where huge data is sorted to check the time it takes to sort using the MapReduce framework. With the help of this application the MapReduce framework is evaluated. we use the traces of 31 GB of data from HiBench benchmark suite.

- Sort - We also use the Sorting application where it sorts the input data which is in text. This input data is generated from a Random TextWriter. This application is widely used to evaluate the performance factor. In this application 32 GB of data was sorted to obtain Job traces.

- PageRank - This application is implemented using the page-rank [19] algorithm, where it calculates the ranks if the web page by counting the number of reference links in the web page. So this application is also used in our evaluation strategy. Here we use 500k pages to process the page rank which was configured by Neves et al. This 500k pages of data has 1 GB of input data where Neves et al. used Pegasus project [2] to derive the job traces. Hence, this is also used by us to know the efficiency of the MapReduce framework during job scheduling.

- Nutch - Apache Nutch [1] is a web crawler, used for page indexing purposes, this is a;so used in MapReduce framework and Neves et al [30] used 5M of pages to do the indexing and it was approximately 8 GB of data.

- Bayesian Classification - This is last application which we are using for our evaluation strategy. It is based on the Naive Bayesian [19] algorithm. It is basically used in Data mining, Machine learning, Knowledge discovery and it is part of Apache Mahout [31]. For this application Neves [30] configured 100K pages to be used as a job trace.

MaxiNet [37] is best to emulate the data center networks as it gives the realistic feel of the data center networks and it was our choice at first. But then, it was not supported by Mremu [30] framework. so we have to use Mininet to create our data center network topologies.

We use two different types of mininet topologies developed by [8] to evaluate our experiment. These two topologies were used to emulate the data center.

- Tree topology - It has 125 servers, 31 switches and 155 links. This consists a Tree topology with a depth equal to 3 and a fanout equal to 5.

- Fat-tree topology - It has 128 servers, 160 switches and 384 links in the network. Here the k factor is set to 32, which means each switch has 32 ports.

5.2 Goal - Job Completion time

In this goal we want to evaluate the job completion time using tree topology. The job traces are serially processed based on the latency/bandwidth parameter. We evaluate to check how efficient our experiment is when compared with the Zhuoyao Zhang et al., [44] where he has not used SDN to abstract the underlying information of the data center network.

5.2.1 Tree Topology

In Figure 5.1, we can see the results of obtained with a Tree topology:

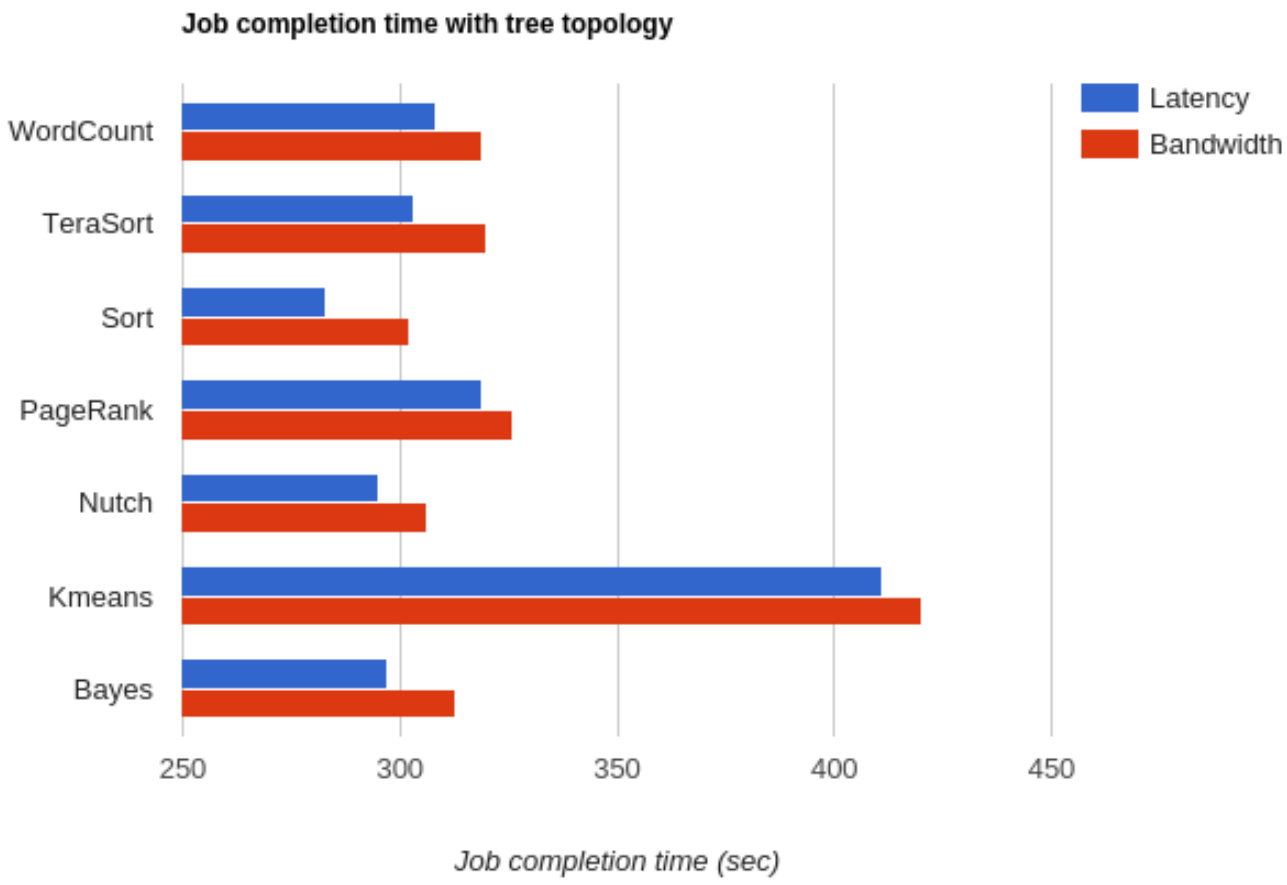

Figure 5.1: Job completion time using a Tree topology.

It has a row of blue bar which depicts the Job completion time when the latency was set as parameter and the red bar in the chart depicts the Job completion time when bandwidth was set as parameter for our job scheduling algorithm.

Our results for the job completion time are better when compared with [44]. Our job scheduling algorithm processed the request of all jobs was quickly and achieved 47.3% less time when compared to Zhuoyao Zhang et al., [44] work.

By analyzing Figure 5.1, we can see that bandwidth oriented task has higher job completion time.

This is because the bandwidth task are scattered into the servers. And the latency based task are closely kept in the server due to which the time taken to complete the job is lower when compared to the bandwidth based job.

5.2.2 Fat-tree Topology

The results for Job completion time using a Fat-tree topology, are presented in Figure 5.2:

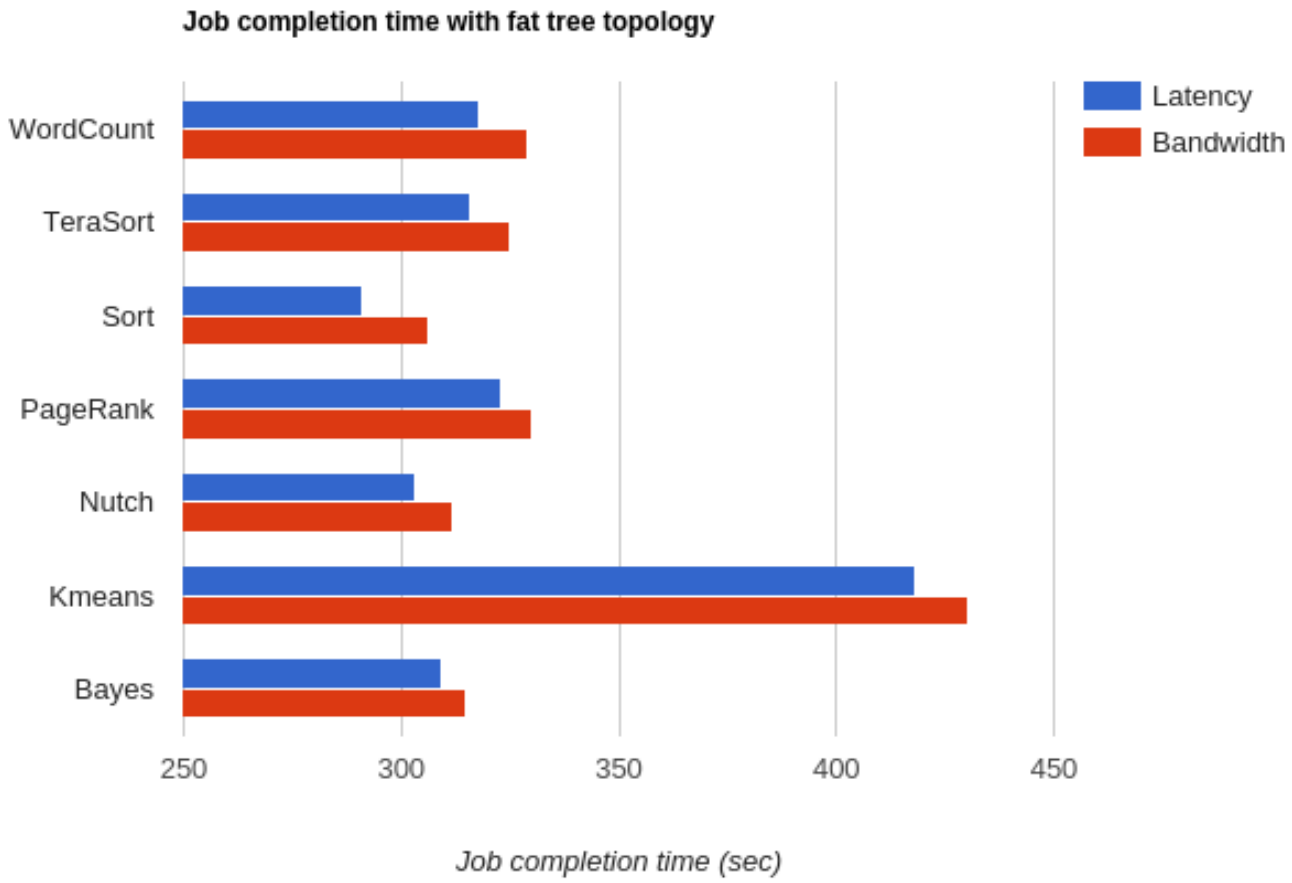

Figure 5.2: Job completion time using fat tree topology.

In this topology we have more links and switches as the k factor is set to 32 and due to this, the fat tree topology would have more path than the Tree topology.

As described in the previous topology the blue bar depicts the job completion time of the latency oriented task and the red bar in the chart depicts the job completion time of bandwidth oriented task.

Looking at Figure 5.2, we can see that the completion time for all the task using this topology is higher than the one using a Tree topology. And when this result was compared with Zhuoyao Zhang et al., [44] work, it showed a variation of 48.1% which seems to be better. Fat tree took more time to complete the job completion time when compared to tree topology as it had more number of paths between two nodes.

5.3 Summary and Final Remarks

In this chapter we have evaluated the goal of our solution, by performing job completion time experiments, on all our job traces. We also introduced our evaluation methodology, where we described about our evaluation environment, which had the test bed of our solution and the evaluation setup. Then we discussed about the job completion times using two different topologies.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation we have addressed the topic of Job scheduling, dynamic network management and configuration in Data Center networks. As we have outlined in chapter 1, Big Data analytics has issues in Data Center networks, due to the performance degradation in the network and also due to the sharing of the network between the tenants.

We have solved this problems by using the abstraction of underlying networks. With the help of abstraction we gathered the network parameters such as bandwidth, latency and CPU to provide better resource to the tenant without degrading the performance. All this was possible by using the Floodlight SDN controller.

We solved this by designing OpenFlow controller that is able to allocate the batch of task in a data center emulated system, achieving the bandwidth/ latency requested during the allocation of tasks. We have designed our task allocation algorithm taking into account the environment where it will run in, a Data Center, which means that it has to scale well.

We have implemented our controller on top of the Floodlight open-source project with the Hadoop Mapreduce emulator. We began studying both the tools so that we can implement our logic into the floodlight controller with which our job scheduling would fulfill its demand of allocating the batch of task. But for this we need to develop a new module inside the Floodlight controller. Apart from that we also make modification inside the MapReduce emulator to be able to provide tenant aware job scheduling.

In MapReduce emulator we have overcome the limitation Neves et al, had. We have enhanced the tool by introducing ”Concurrent job execution” and we also made many changes so that we can make our implementation possible.

After the comprehensive evaluation of our implementation. we use the developed scripts [8] to create two Data Center network topologies: Tree and Fat-tree. After the evaluation conducted, we got the results, which were as we intended in the beginning of this dissertation. From the evaluation, we can see that our system has: low execution time when it was allowed to schedule the job based on bandwidth or latency parameters.

6.2 Future Work

As we have come to the end of this dissertation we would like to share the future work, which could be done on this project.

- **Run bandwidth and latency task together** - In this thesis we have not combined the bandwidth and latency based job scheduling. But for the improvement this would be the best choice. If this is done then we can have better solution and which can be very useful in the data center network as it can be simultaneously executed.

- **Make the controller and emulator fault tolerant** - Mremu emulator and the floodlight controller is not fault tolerant so we can try to work around them to be fault tolerant, both to failures in links and in nodes (switches and servers). By making these technology fault tolerant, and applying this in the real data center network would show way to new research.

- **Parallelize the processing of task allocation** - Our algorithm supports only serialized task allocation. If the OpenFlow controller could parallelize the task requests simultaneously, then there would be great change in the performance and we can process many request at the same time. This also can be taken as a challenge and can be developed into a new research.

- **Create new network topologies** - Our job scheduling algorithm need to be tested with different types of the topologies to know in which topology the job scheduling runs well and can bring better solution to our algorithm.

Bibliography

[1] Apache nutch. URL `http://nutch.apache.org/`.

[2] Pegasus: A peta-scale graph mining system implementation and observations. URL `http://www.cs.cmu.edu/~pegasus/`.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[5] L. A. Barroso, J. Clidaras, and U. 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.

[6] K. Bilal, S. U. Khan, J. Kolodziej, L. Zhang, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang, and D. Chen. A comparative study of data center network architectures. In *ECMS*, pages 526–532, 2012.

[7] D. Borthakur. Hdfs architecture guide. hadoop apache project, 2008.

[8] D. Caixinha, P. Kathiravelu, and L. Veiga. Vitena: An sdn-based virtua; network embedding algorithm for multi-tenant data centers., In 15th IEEE International Symposium on Network Computing and Applications(NCA 2016).

[9] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. *ACM SIGCOMM Computer Communication Review*, 41(4):254–265, 2011.

[11] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu. Transparent and flexible network management for big data processing in the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*, 2013.

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

[13] A. Desai and K. Nagegowda. Advanced control distributed processing architecture (acdpa) using sdn and hadoop for identifying the flow characteristics and setting the quality of service (qos) in the network. In *Advance Computing Conference (IACC), 2015 IEEE International*, pages 784–788. IEEE, 2015.

[14] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An api for application control of sdns. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 327–338. ACM, 2013.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[16] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[17] M. Hammoud, M. S. Rehman, and M. F. Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 49–58. IEEE, 2012.

[18] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. 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.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[21] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[22] B. Lantz, B. Heller, and N. 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.

[23] Z. Li, Y. Shen, B. Yao, and M. Guo. Ofscheduler: a dynamic network optimizer for mapreduce in heterogeneous cluster. *International Journal of Parallel Programming*, 43(3):472–488, 2015.

[24] C. Liang. Software defined network support for real distributed systems.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[26] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014.

[27] S. Narayan, S. Bailey, and A. Daga. Hadoop acceleration in an openflow-based cluster. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, pages 535–538. IEEE Computer Society, 2012. ISBN 978-1-4673-6218-4. doi: 10.1109/SC.Companion.2012.76. URL `http://dx.doi.org/10.1109/SC.Companion.2012.76`.

[28] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[29] M. V. Neves. Mremu: An emulation-based framework for datacenter network experimentation using realistic mapreduce traffic. URL `https://github.com/mvneves/mremu`.

[30] M. V. Neves, C. A. De Rose, and K. 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.

[31] S. Owen, R. Anil, T. Dunning, and E. Friedman. Mahout in action. 2012.

[32] J. Qadir, N. Ahad, E. Mushtaq, and M. Bilal. Sdns, clouds, and big data: New opportunities. In *Frontiers of Information Technology (FIT), 2014 12th International Conference on*, pages 28–33. IEEE, 2014.

[33] P. Qin, B. Dai, B. Huang, and G. Xu. Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. *arXiv preprint arXiv:1403.2800*, 2014.

[34] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[35] G. Wang, T. Ng, and A. 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.

[36] S.-Y. Wang, C.-L. Chou, and C.-M. Yang. Estinet openflow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, 2013.

[37] P. Wette, M. Dräxler, and A. Schwabe. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[38] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[39] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[40] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie. A survey on software-defined networking. *Communications Surveys & Tutorials, IEEE*, 17(1):27–51, 2014.

[41] Z. Yu, M. Li, X. Yang, and X. Li. Palantir: Reseizing network proximity in large-scale distributed computing frameworks using sdn. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 440–447. IEEE, 2014.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[43] D. Zeng, L. Gu, and S. Guo. *Cloud Networking for Big Data*. Springer, 2015.

[44] Z. Zhang, L. Cherkasova, and B. T. Loo. Optimizing cost and performance trade-offs for mapreduce job processing in the cloud. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.