Delft University of Technology
Master's Thesis in Embedded Systems

A Dynamic Multipath File Transfer Engine for Software-Defined Networking

Yuchen Huang

TU Delft Delft University of Technology

A Dynamic Multipath File Transfer Engine for Software-Defined Networking

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Yuchen Huang
4511980
Y.Huang-14@student.tudelft.nl

16th November 2018

Author
Yuchen Huang (Y.Huang-14@student.tudelft.nl)
Title
A Dynamic Multipath File Transfer Engine for Software-Defined Networking
MSc presentation
16th November 2018

Graduation Committee

Fernando A. Kuipers	Delft University of Technology
Przemek Pawelczak	Delft University of Technology
Edgar van Boven	Delft University of Technology & KPN

Abstract

With the fast development of the internet, it is widely expected that data traffic will grow exponentially. To fulfill the internet users' demand of Quality of Experience (QoE), the way we deal with "the tsunami of data" becomes a big problem. How to transfer data, especially those very big files, as fast as possible can be a very important part of that problem. One promising solution could be using multiple paths instead of a single path transmission. Different conditions on different paths make it difficult to handle multipath transmission well. However, Software-Defined Networking (SDN), a centralized network model, provides new solutions to make multipath transmission possible. In a SDN network, the network status can be monitored. With such monitoring information, an optimal multipath solution can be found to transfer a file as fast as possible.

In this thesis, the optimal multipath transmission problem is studied. The objective is to minimize file transfer time between two end systems in a given SDN network. First, we model this as a linear programming problem on a time-expanded network. The time-expanded network used makes that the complexity of this solution is pseudo-polynomial. From this solution, we are able to derive a Fully Polynomial-Time Approximation Scheme (FPTAS) where we can accurately control the speed versus accuracy trade-off. Lastly, we have build an SDN-based proof-of-concept implementation through which we have evaluated our two algorithms.

Preface

First, I would like to thank Professor Fernando A. Kuipers. It is his professional advises and kindness that help me overcome all technical problems and finish this thesis. Then, I'd like to thank all the friends I've met in Delft. I will remember all the wonderful time and terrible food we enjoyed together. Finally, I'd like to thank my beloved girlfriend Lina He and my parents, I can't image how my life will be without their support and endless love.

Yuchen Huang

Delft, The Netherlands
16th November 2018

Contents

Preface v

1 Introduction 1

2 Software Defined Networking 3
2.1 SDN 3
2.1.1 SDN Architecture 3
2.2 OpenFlow 4
2.2.1 Flow Table 5
2.2.2 Group Table 6
2.2.3 OpenFlow Messages 6

3 Related work 9
3.1 Multipath routing 9
3.2 Traffic engineering in SDN 10
3.3 Dynamic control in SDN 10
3.4 Summary 11

4 Algorithm 13
4.1 Network model 13
4.2 Problem definition 14
4.3 Exact pseudo-polynomial algorithm 15
4.4 Fully polynomial-time approximation algorithm 18
4.4.1 Algorithm and proof 18

5 Proposal and Design 23
5.1 System design 23
5.2 Monitoring component 25
5.2.1 Link Delay 25
5.2.2 Link Load 26
5.3 Forwarding component 27
5.4 Enable multipath forwarding in Open vSwitch 27
5.4.1 Modify group bucket selection algorithm 27
5.4.2 Ensure file partition and bandwidth usage 29

5.5 Enable dynamic control in path selection 30
5.5.1 Process of dynamic path control 30

6 Evaluation 33
6.1 Algorithm test 33
6.1.1 Random graph 34
6.1.2 Square lattice graph 37
6.2 SDN network experiments 40
6.2.1 Test over different number of paths 40
6.2.2 Test with delay different delays 43
6.2.3 Test with dynamic path control 45

7 Conclusion & Future Work 49
7.1 Conclusion 49
7.2 Future Work 50

A Open vSwitch 2.7.0 Group Table Modification 55
A.1 ovs/ofproto/ofproto-dpif-xlate.c 55

List of Figures

2.1 The SDN architecture. [13] 4
2.2 Main components of an OpenFlow switch. [22] 5

4.1 Example input graph $G(V, E)$ and the corresponding optimal solution when $\sigma = 13$. 14
4.2 Example input graph $G(V, E)$ and corresponding auxiliary graph $G^A(V^A, E^A, \tau = 4)$. 16

5.1 The architecture of our SDN-based file transfer system 24
5.2 Application layer 24
5.3 Process of detecting link delay 25
5.4 Process of link load monitoring 26
5.5 Example of original and modified select group in Open vSwitch 28
5.6 Example of file partition using select group. 29
5.7 Dynamic path control process. 30

6.1 Running time results on the random graphs. 34
6.2 Relative errors in BFMS-A on the random graphs. 35
6.3 Relative errors in BFMS-A with different ϵ 36
6.4 Results of tests with 30 nodes and 1000000 units file when ϵ changes. 36
6.5 Running time results on the square lattice graphs. 38
6.6 Relative errors in BFMS-A on the square lattice graphs. . . . 39
6.7 Relative errors in BFMS-A with different ϵ. 39
6.8 Results of tests with 36 nodes and 1000000 units file when ϵ changes. 40
6.9 Example topology when 3 redundant paths are available. . . . 41
6.10 Transfer speed over different path numbers 43
6.11 Transfer speed over different path numbers when bandwidth is guaranteed 44
6.12 Topology of delay differential experiment 45
6.13 Transfer speed over different path delay 46
6.14 Scenario for dynamic path control experiment 47

6.15 Throughput with and without dynamic control. Additional paths are available after 50 and 100 seconds. 47

List of Tables

2.1 Main components of a flow entry. [22] 5
2.2 Main components of a group entry. [22] 6
2.3 OpenFlow Messages 7

Acronyms

ACO Ant Colony Optimization. 9

APIs Application Program Interfaces. 3

BFMS Big File Multipath Scheduling. 14, 15

DCN Data Center Networking. 9

FPTAS Fully Polynomial-Time Approximation Scheme. iii, 18, 49

IETF Internet Engineering Task Force. 9

MAF Maximum Auxiliary Flow. 17, 18

MPTCP MultiPath TCP. 9–11, 27

MTU Maximum Transmission Unit. 1

NBIs NorthBound Interfaces. 3, 4

ONF Open Network Foundation. 4

OVS Open vSwitch. 3, 27, 30, 55

QoE Quality of Experience. iii, 1, 11

QoS Quality of Service. 1, 11

RTT Round-Trip Time. 25

SDN Software-Defined Networking. iii, ix, 1–4, 9–11, 23–25, 27, 33, 40, 49, 50

SFTS Single File Transfer Scheduling. 15

TE Traffic Engineering. 1, 9, 10

Chapter 1

Introduction

With internet becoming more and more important in people's lives, enormous amounts of data are generated and transferred. Cisco Visual Networking Index forecasts that, by 2021, overall IP traffic will grow to 278 exabytes per month [3]. New technologies and new internet services make this trend more obvious. Users now require video services to provide 2K, 4K, and even 8K resolution. The data or files transferred in the internet are getting bigger and bigger. Inefficient bandwidth and long latency will have huge negative impact on users' QoE.

SDN has introduced a new way to provide Traffic Engineering (TE) and Quality of Service (QoS). Compared to traditional IP network, SDN has a centralized architecture in which the control plane and data plane are decoupled. This enables network managers to configure a network device to have various functions. Applications can also access the status of each port on each device and each link connected to it, which allows applications to make suitable decisions for better TE and QoS.

Multipath is a promising approach to the TE problem. It can offer better resource utilization, better reliability, and often even much better QoE. However, these advantages are not free, other problems are also introduced by multipath routing, such as variable path Maximum Transmission Unit (MTU), variable path latency, and even increasing difficulty of debugging on network management. But with SDN, new solutions can be found to reduce those disadvantages.

In this thesis, our objective is to reduce the transfer time of big size files in an SDN-enabled network. To realize our objective, we will address the following research problems:

1. How to take advantage of Software Defined Networking and achieve multipath file transfer in its environment?

2. How to find a multipath solution in reasonable time?

3. How to deal with the rapid change in network status?

Our main contributions are:

1. We propose and implement a big file transfer engine in an SDN-enabled network which can provide network monitoring, flow scheduling and multipath packet forwarding.
2. We propose a fully polynomial-time approximation algorithm to find a multipath file transfer solution.
3. We propose a dynamic scheduling scheme to reschedule flow according to the current network status.

The outline of this thesis is as follows. In chapter 2, we introduce Software Defined Networking and the OpenFlow protocol. They are the basic architecture and protocol for our solution. Before proposing our solution, related work is presented in chapter 3. In chapter 4, we explain our network model and the research problem. We also propose our fully polynomial-time approximation algorithm in this chapter. In chapter 5, we present the details of our proposal from the overall design of the whole system to the details of each component. Next, we perform many experiments in different scenarios to evaluate our solution in chapter 6. The results and evaluation are also in this chapter. Finally, the conclusions and future work are described in chapter 7.

Chapter 2

Software Defined Networking

2.1 SDN

Traditional networks are now facing several problems. First, the control of these networks is completely distributed to every single device. Current networks can easily contain more than thousands of network devices, so network managers can hardly control and manage the behavior of every device. Second, network control and forwarding are integrated into one device: any update or change on both parts cannot be easily achieved. Third, network managers cannot control the forwarding process directly, but over indirectly through configuring network protocols. Fourth, the inflexibility of network protocols becomes an obstacle to current fast-changing network applications.

SDN brings a new way of controlling networks. It introduces a network architecture where network control is decoupled from packet forwarding. The new architecture controls networks in a centralized way and network managers can apply their control policies directly via programmable Application Program Interfaces (APIs).

2.1.1 SDN Architecture

Figure 2.1 shows the basic components of an SDN architecture. It consists of three layers: infrastructure layer, control layer and application layer. The infrastructure layer, also known as data plane, comprises all network devices. A network device can be a physical switch, a virtual switch like Open vSwitch (OVS), or even another device like a router. All these network devices need to support an SDN southbound interface so that they can communicate with the control layer via it. OpenFlow is one of the most popular southbound interfaces, other choices can be OpFlex, POF and P4. The control layer, or the control plane, consists of one or more SDN controllers. The controllers manage the devices in the data plane in a centralized way and they are directly programmable via SDN NorthBound Interfaces (NBIs) specified by the controller. Controllers are the core elements of the

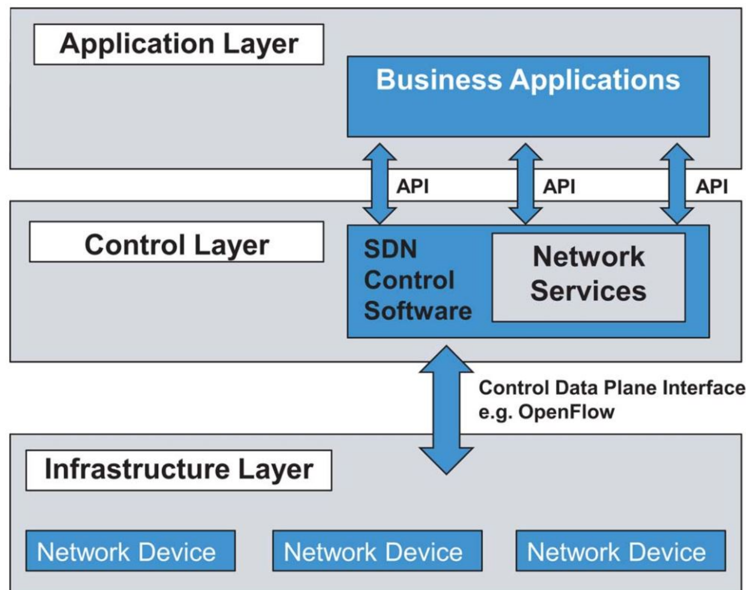


Figure 2.1: The SDN architecture. [13]

SDN architecture. There are many controllers available on the market, such as Ryu, Floodlight and Pox. Business applications exist in the application layer, and communicate their network requirements toward the controller via the NBIs. The applications can be load balancing, firewall, monitoring and other more complicated business services.

2.2 OpenFlow

OpenFlow is the most popular implementation of the SDN concept. It provides an open source Southbound API between controllers and switches. OpenFlow was originally proposed by Stanford University to help researchers to utilize the network hardware and design new protocols [13]. Now, OpenFlow is maintained by the Open Network Foundation (ONF). More and more commercial groups and companies are using OpenFlow to design their SDN platform and hardware [26].

The control plane is represented by controllers, while the data plane stays in OpenFlow switches. Figure 2.2 shows the main components of an OpenFlow switch. An OpenFlow switch consists of one or more OpenFlow channels, which can be used to communicate with controllers through the OpenFlow protocol, and one or more flow tables and one group table, which together perform the forwarding function [22].

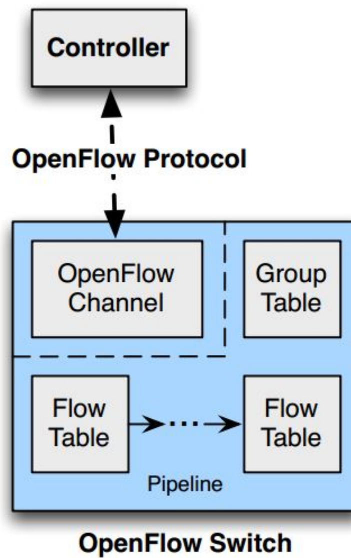


Figure 2.2: Main components of an OpenFlow switch. [22]

2.2.1 Flow Table

Flow tables are the most important feature in the OpenFlow switch. A flow table consists of flow entries. Table 2.1 shows the main components in a flow entry.

Match Fields	Priority	Counters	Instructions	Timeouts

Table 2.1: Main components of a flow entry. [22]

The match fields contain a subset of values that can be found in a packet header which can be used to match against packets. Additionally, one can also set ingress port value in the match fields to match packets coming from a specific port. The priority is used when a packet can be matched to multiple flow entries. The flow entry with highest priority is chosen to match the packet. When a packet is matched, the instructions in the flow entry show how the packet is going to be handled. Possible instructions can be Apply-Actions like output and drop, which perform immediately, and Goto-Table, which forward the packet to another flow table for pipeline processing. The timeouts contain information about the maximum living time or idle time of the corresponding flow entry. The counters show how many packets have been matched for the flow entry so far.

2.2.2 Group Table

Group table is another important feature of OpenFlow. Table 2.2 shows the main components of a group entry in a group table.

Group Identifier	Group Type	Counters	Action Buckets

Table 2.2: Main components of a group entry. [22]

Each group entry represents a group. A group is identified by the number in the group identifier. Flow tables can send packets to a specific group by using this number. Counters, similar to that in a flow entry, show the number of processed packets. The action buckets consist of an ordered list of action buckets. Each action bucket contains a set of actions and other associated parameters. In the group type, a group can be set to four different types:

- Indirect: Execute the only action bucket in the group. This is the simplest type of group, it only contains one action bucket. Indirect groups are designed for saving hardware resources. Multiple flow entries can point to the same indirect group to perform the actions in the action buckets.
- All: Execute all the action buckets in the group. A packet processed by an all group is copied and forwarded to all the action buckets in that group. Each bucket then applies actions to its own copy of the packet.
- Select: Execute one action bucket in the group. Multiple action buckets can be defined in a select group, but only one can be applied to one packet according to a selection algorithm specified by the switch.
- Fast failover: Execute the first live action bucket. This group is used for protection. When one route is broken, the switch can quickly shift to a pre-computed backup route.

2.2.3 OpenFlow Messages

The OpenFlow protocol defines OpenFlow Messages transmitted between an OpenFlow controller and an OpenFlow switch. These messages enable the controller to control the switch through the add, update, and delete actions on the flow entries in the flow tables. There are three types of messages, as shown in Table 2.3.

- *Controller-to-Switch Messages:* These messages are initiated by the controller and sent to the switch. They can modify the logical state of

the switch, for example, its configuration and flow/group entries. In some cases, these messages require a response from the switch. The packet-out message will be sent from the controller to the switch when the controller decides not to drop the packet, but to direct it to a switch port.

- *Asynchronous Messages:* These messages are sent without solicitation from the controller, for example, the port and flow status sent to the controller. They also include the Packet-in message, which can be used by the switch to send a packet to the controller when there is no flow-table match.
- *Symmetric Messages:* These messages are sent without solicitation from either the controller or the switch. Hello messages are exchanged between the controller and switch to establish the first connection. Echo request&reply messages are used by either the switch or controller to measure the latency or bandwidth of a controller-switch connection or just verify that the device is operating. The Experimenter message is used to build features in future versions of OpenFlow.

Table 2.3: OpenFlow Messages

Controller-to-Switch	
Handshake	Identify the switch and its capabilities.
Configuration	Set and query configuration parameters.
Modify-State	Modify flow/group entries, port behavior, and meters.
Multipart	Request statistics or state information from switch.
Packet-out	Send a packet to a specified port on the switch.
Barrier	Ensure message dependencies have been met or receive notifications for completed operations.
Role-Request	Change the controller role in multiple-controller situation
Asynchronous -Configuration	Set and query filter on asynchronous messages in multiple-controller situation.
Asynchronous	
Packet-in	Transfer packet to controller.
Flow-Removed	Inform the controller about the time out or deleted flow
Port-Status	Inform the controller of a change on a port.
Error	Notify controller of error or problem condition.
Symmetric	
Hello	Exchanged between the switch and controller upon connection start-up.
Echo	Request & Reply between the switch and the controller.
Experimenter	For additional functions.

Chapter 3

Related work

In this chapter, we present our review of the current research progress around traffic scheduling in traditional IP networks and SDN-enabled networks. We classify those related works into three major topics: (1) multipath routing based traffic scheduling approach; (2) TE in SDN-enabled networks and (3) Dynamic control in traffic scheduling.

3.1 Multipath routing

Since multipath routing has its natural advantages over single path routing in terms of throughput, fail-over protection and many other aspects, enormous research efforts have been spent in this direction. In [31], Ya *et al.* show a way to provide load balancing using multipath in Data Center Networking (DCN). They propose an improved Ant Colony Optimization (ACO) algorithm, which uses the idea of data flow segmentation to improve the network throughput and resource utilization. In [12], L. He proposed a survivability scheme for optical networks named Hybrid Single and Multiple Backup Protection (HSMBP). This scheme uses multiple paths to protect the primary path. It shows that HSMBP can effectively improve the network performance by reducing Bandwidth Blocking Probability (BBP).

Recently, MultiPath TCP (MPTCP) has become popular in the multipath routing research area. MPTCP is an extension to the regular TCP protocol. It was originally a Linux kernel project and now is standardized by the Internet Engineering Task Force (IETF) [10]. Multipath TCP provides the ability to split one transport connection into multiple "subflows" and simultaneously use multiple paths between peers. In [28], Pol *et al.* measured the end-to-end goodput with MPTCP in their intercontinental OpenFlow testbed. The results shown that they could indeed reach a much higher throughput with multiple paths compared to a single path. Much research has been published to take advantage of the multipath routing based on MPTCP. However, the usage of MPTCP is limited, since it works at

the transport layer. To set up an MPTCP connection, we need both host devices support it. Currently, not all devices support MPTCP, although many networks have redundant paths that can be used to set up multipath connections. As a result, MPTCP applications are still hardly used.

3.2 Traffic engineering in SDN

SDN is a key technique to achieve intelligent TE. Under the architecture of SDN, TE applications are applied to traffic in the data plane though the SDN controller.

One typical example is B4 which is a private WAN connecting Google's data centers. It uses an SDN architecture to control TE services centrally. The corresponding paper shows that B4 can make links to nearly 100% utilized and balance capacity against application priority/demands by splitting application flows among multiple paths [14]. In [17], J. Lin *et al.* proposed a system named MonArch to monitor and measure resources based on the concept of Software Defined Infrastructure (SDI). Users can generate monitoring tasks to conduct TE. In [9], Y. Guo *et al.* explored TE in an SDN/OSPF hybrid network. In their scenario, the SDN controller can arbitrarily split the coming flows into outgoing links. Under this scenario, they proposed an algorithm to solve the TE problems and obtain lower maximum link utilization. In [25], Porxas *et al.* proposed a multi-tenancy management framework to meet the quality-of-service requirements through tenant isolation, prioritization and flow allocation. In a centralized SDN scenario, it can improve the network performance regarding traffic congestion and packet delay.

Like the aforementioned solutions, most solutions about TE problems are based on static scheduling, which can not guarantee the scheduling requirements in actual scenarios.

3.3 Dynamic control in SDN

SDN is a core technology for adaptive and flexible networks. It enable the network to dynamically optimize itself to the needs of diverse services. There is some research about dynamic control in SDN.

In [20], Rashid *et al.* used the SDN controller to efficiently manage resources. The virtual network is dynamically adjusted to various network states in the substrate network. The central controller can monitor the resources in virtual networks, in particular the average loading of links and switches in substrate networks. Such information will help the controller to control flows, and consequently more requests can be accepted with less resource cost. In [1], Iris *et al.* focused on TE considering the characteristics of running applications in the network. They proposed a software

framework based on SDN to allow dynamic provisioning to meet the QoS of different services. In [18], Hui *et al.* worked on workload balancing. They proposed a novel path-switching algorithm under the architecture of SDN to dynamically balance the traffic during the transmission.

Furthermore, Hyunwoo *et al.* proposed to dynamically control MPTCP flows under the architecture of SDN [21]. The results show that their mechanism can improve the network performance. In terms of adaptive rate video streaming, they got faster download and better QoE.

3.4 Summary

Multipath routing has attracted a lot of attention. However, we still lack of algorithms that take advantage of the centralized SDN to provide an optimal solution when transfer a file with multipath. Also, some dynamic control can be added to keep the optimal solution up-to-date when network status changes.

Chapter 4

Algorithm

In this chapter, we first present our network model and introduce our research problem. Then, we give an exact algorithm with pseudo-polynomial time complexity to solve the problem. Based on this result, we also propose a fully polynomial-time approximation algorithm and prove its correctness and time complexity.

4.1 Network model

Before we introduce the algorithm, we first model the question we want to solve. We use the network model from [16]. Our network can be modeled as a weighed directed graph $G(V, E)$, where V is the set of nodes and E is the set of edges. For each link $e \in E$, we have bandwidth $b(e) > 0$ and delay $d(e) \geq 0$. For a node pair $(s, t) \subseteq V$, $p = (s, v_1, v_2, ..., t)$ is a path from node s to node t. The bandwidth of a path p is

$$b(p) = \min_{e \in p} b(e) \tag{4.1}$$

and the delay of a path p is

$$d(p) = \sum_{e \in p} d(e) \tag{4.2}$$

Assume we are transferring a file of size σ with a set $\mathcal{P}$ of paths from node s to node t. For each path $p_i \in \mathcal{P}$, a sub-file of size σ_i is transferred with the used bandwidth $f(p_i) \leq b(p_i)$, where $\sum_{i \in [1,k]} \sigma_i = \sigma$. The total bandwidth used to transfer the file, $f(\mathcal{P})$, is

$$f(\mathcal{P}) = \sum_{p_i \in \mathcal{P}} f(p_i) \tag{4.3}$$

and sub-file transfer time $T(p_i)$ on path p_i is

$$T(p_i, f(p_i), \sigma_i) = d(p_i) + \frac{\sigma_i}{f(p_i)}, i \in [1, k] \tag{4.4}$$

The total file transfer time $T(\mathcal{P}, f(\mathcal{P}), \sigma)$ on the path set $\mathcal{P}$ is

$$T(\mathcal{P}, f(\mathcal{P}), \sigma) = \max_{i \in [1,k]} T(p_i, f(p_i), \sigma_i) \tag{4.5}$$

4.2 Problem definition

Big File Multipath Scheduling (BFMS) problem: Let σ be the size of a file that needs to be transferred from node s to node t, find a set of feasible paths $\mathcal{P}$, a feasible $s-t$ flow f, and a file distribution among the paths $p_i \in \mathcal{P}$, such that $T(\mathcal{P}, f(\mathcal{P}), \sigma)$ is minimized.

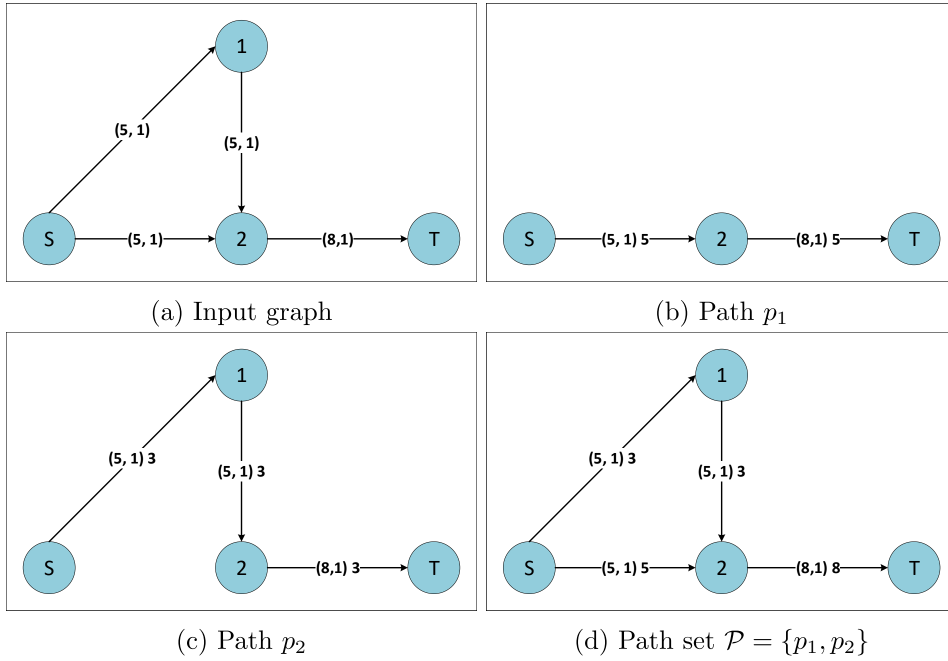


Figure 4.1: Example input graph $G(V, E)$ and the corresponding optimal solution when $\sigma = 13$.

An example is shown in Figure 4.1. Figure 4.1a shows the topology of the network where the number on each link gives the information of the link weight in the form (bandwidth, delay). Assume we want to transfer a file of size $\sigma = 13$ units. Solutions should contain three parts, namely the path set, the bandwidth usage, and the file distribution. In this example, the optimal solution should be a path set $\mathcal{P} = \{p_1 = (s, 1, 2, t), p_2 = (s, 2, t)\}$ with bandwidth usage $f(p_1) = 3, f(p_2) = 5$, and a sub-file of 3 unit size is transferred on p_1 while a sub-file of 10 unit size is transferred on p_2. Thus, according to Equation 4.5, the total transferring time $T(\mathcal{P}, f(\mathcal{P}), \sigma) = 4$ units.

Like the original Single File Transfer Scheduling (SFTS) problem in [16], our BFMS problem is NP-hard.

4.3 Exact pseudo-polynomial algorithm

To solve the BFMS problem, we propose the BFMS-E algorithm which is a variant of the sfts-a algorithm in [16]. Before introducing our algorithm, first we discuss some techniques used in the sfts-a algorithm.

In the sfts-a algorithm, auxiliary graphs, also known as time-expanded networks, are used to solve sub-file distribution problems. The original sfts-a algorithm only returns a solution when it is feasible to transfer the file within a time constraint τ. In our problem, we do not have a pre-defined time constraint, we set the initial value of τ to be the transfer time needed in the shortest path solution. Since the shortest path solution is one of the possible solutions to our problem, the τ we set is definitely feasible for the algorithm to solve. An auxiliary graph $G^A(V^A, E^A, \tau)$ can be constructed from $G(V, A)$ with the following steps:

1. For each node $u \in V, u \neq s$, τ nodes, u^0, u^1, ..., u^τ, are added to V^A. For node $s \in V$, add only node s^0 to V^A.

2. For each link $(u, v) \in E, u \neq s$, $(\tau - d(u, v))$ links are added to E^A in the form $(u^i, t^{i+d(u,v)})$, where $i = 0, 1, ..., \tau - d(u, v) - 1$. For each link $(s, u) \in E$, add link $(s^0, u^{d(s,u)})$.

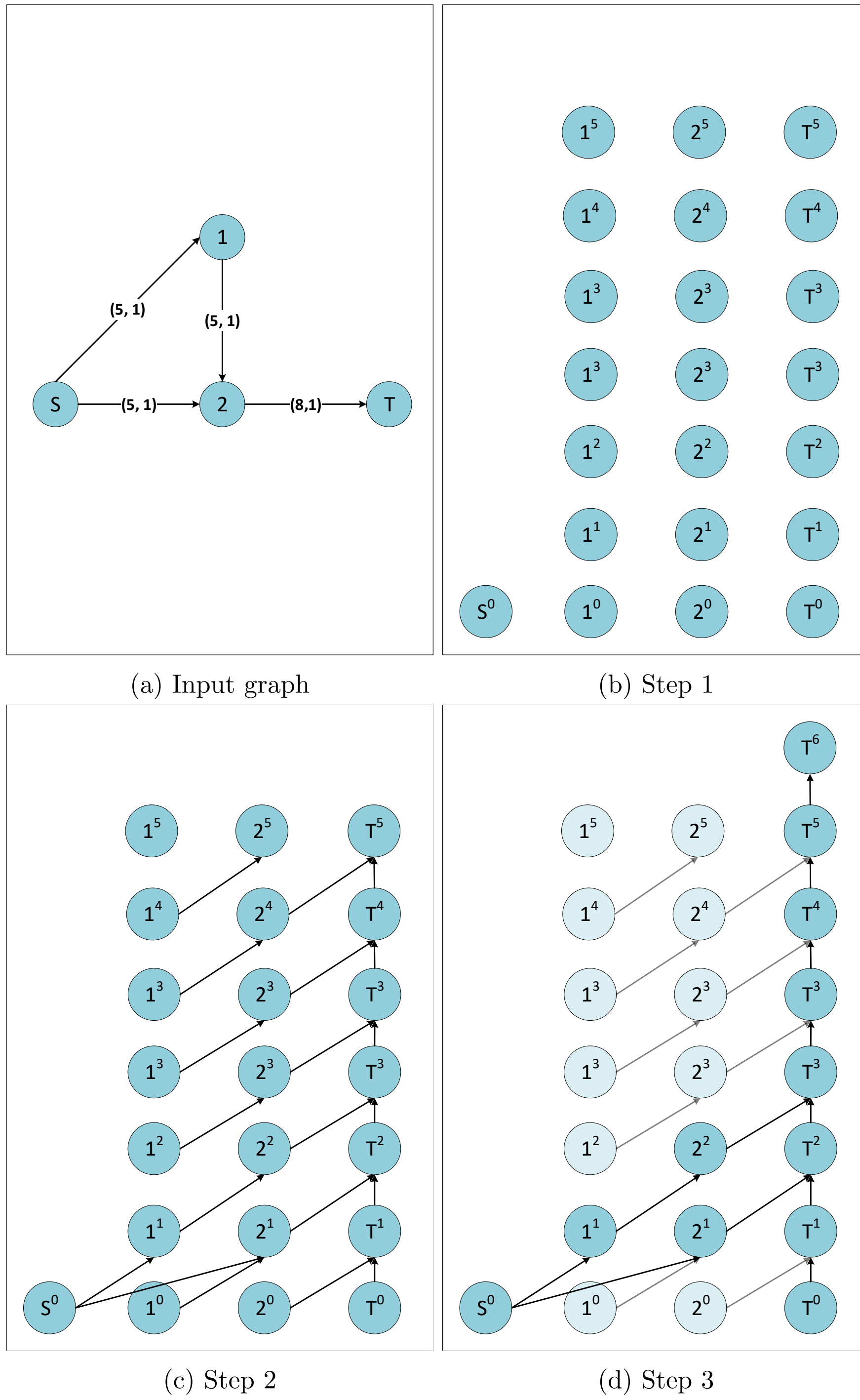
3. Add node t^τ to V^A and link $(t^{\tau-1}, t^\tau)$ to E^A. For each node $u \in V^A, u \neq s^0, t^i$, where $i = 0, 1, ..., \tau$, remove node u whose in-degree or out-degree is 0.

Figure 4.2 shows an example of an input graph and the process of constructing its corresponding auxiliary graph following the aforementioned steps. Assume we want to transfer a file of size $\sigma = 20$ units. The shortest path from s to t in the input graph is path $p = (s, 2, t)$. The maximum bandwidth $b(p) = 5$ units and delay $d(p) = 2$ units. According to Equation 4.4, the transfer time $T(p, b(p), \sigma) = 6$ units, which is then assigned to τ. In Figure 4.2d, the dark nodes show the final auxiliary graph and the light nodes are removed.

When comparing the auxiliary graph G^A with the original input graph G, we can find G^A contains two important pieces of information:

1. Path information: The same path information is "copied" from G to G^A. For each path from s^0 to t^i in G^A, there is also a corresponding path from s to t in G.

2. Delay information: The additional link path delay information is introduced in G^A. The superscript of each node shows the path delay from s^0 to itself.



(a) Input graph (b) Step 1

(c) Step 2 (d) Step 3

Figure 4.2: Example input graph $G(V, E)$ and corresponding auxiliary graph $G^A(V^A, E^A, \tau = 4)$.

So we can conclude that G^A shows all the paths from s to t in G with its path delay less than the time limitation τ. Then the scheduling problem, called Maximum Auxiliary Flow (MAF) problem, can be expressed as follows:

$$\min \sum_{\forall (u,v)\in E^A} d(u,v)\cdot f(u,v) - f(G^A)\cdot \tau \tag{4.6}$$

$$\text{s.t.} \sum_{(s^0,v^{d(s,v)})\in E^A} f(s^0, v^{d(s,v)}) = f(G^A) \tag{4.7}$$

$$\sum_{(v^i,t^{i+d(v,t)})\in E^A} f(v^i, t^{i+d(v,t)}) = f(G^A) \tag{4.8}$$

$$\sum_{(x,y)\in E^A} f(x,y) = \sum_{(y,z)\in E^A} f(y,z), \qquad \forall y \in V^A, y \neq s,t \tag{4.9}$$

$$\sum_{u^i\in V^A} f(u^i, v^{i+d(u,v)}) \leq b(u,v), \qquad \forall (u,v)\in E^A \tag{4.10}$$

$$f(u,v) \geq 0, \qquad \forall (u,v)\in E^A \tag{4.11}$$

such that the maximum amount of file that can be transferred from node s to node t in graph G^A within time limitation τ is

$$M(G^A,\tau) = f(G^A)\cdot\tau - \sum_{\forall (u,v)\in E^A} d(u,v)\cdot f(u,v) \tag{4.12}$$

Algorithm 1 BFMS-E algorithm

1: **procedure** MyProcedure
2: Compute the shortest path p^* from s to t in G.
3: $\tau \leftarrow T(p^*, b(p^*), \sigma)$.
4: $x \leftarrow d(p*) + 1$; $y \leftarrow \tau$.
5: **do**
6: Construct the auxiliary graph $G^A(V^A, E^A, \tau)$.
7: Solve the MAF formalation.
8: **if** $M(G^A,\tau) \geq \sigma$ **then**
9: $y \leftarrow \tau$
10: **else**
11: $x \leftarrow \tau$
12: $\tau \leftarrow \lfloor (x+y)/2 \rfloor$
13: **while** $y \neq x+1$
14: $\tau \leftarrow y$
15: Output the s-t path set $\mathcal{P}$ obtained from the MAF solution with the corresponding flow allocation $f(p_i)$ for each $p_i \in \mathcal{P}$.

Based on this formulation, we propose algorithm 1 (Algorithm BFMS-E). As explained before, we do not just want to transfer the file within a time

constraint, instead we try to transfer the file as fast as possible. The main idea is using binary search to narrow the difference between the upper and lower bounds until we find the optimal solution. We initial the upper bound y of time constraint τ to be the time needed to transfer the file with the shortest path solution. Since the shortest path solution must be a possible solution, we can view it as a known upper bound. The initial value of the lower bound is set to be the total link delay of the shortest path between node s and t plus 1 time unit. We set this lower bound because no matter how we optimize the transfer time, we can not make it less than the minimal link delay between the two nodes. Then, we change the value of τ in a binary search way. Each time after the MAF formation is solved, if a feasible solution is found, assign the current τ to the upper bound y, otherwise, assign τ to the lower bound x. Then solve the new MAF formation with τ to be the halve of the sum of x and y. The execution ends when the minimal transfer time τ is found. Although Algorithm BFMS-E can give the optimal result, the introduction of the time-expanded network makes its time complexity pseudo-polynomial [8].

4.4 Fully polynomial-time approximation algorithm

Although the proposed algorithm can solve the problem exactly, its pseudo-polynomial time complexity makes it still computational unfeasible when the problem size grows too big. In this section, we propose a fully polynomial-time approximation algorithm to solve this problem. We use the FPTAS derived by Lorenz for the restricted shortest path problem [19]. We prove that this scheme can also be applied to our problem and give an ε-solution with time complexity to be $O(poly \cdot (\frac{n}{\varepsilon}\log\frac{n}{\varepsilon} + n\log n\log\log(\frac{UB}{LB}))$.

4.4.1 Algorithm and proof

The main idea of this algorithm is to apply the rounding and scaling technique to the original sfts-a algorithm. We use $\sim$ to indicate the notation is for the scaled graph and $*$ to indicate the optimal. We scale an instance of the problem by setting $\tilde{d}(e) = \lfloor\frac{d(e)}{S}\rfloor + 1$ and $\tilde{b}(e) = b(e)S$. Also, when we apply a solution for the original problem to its scaled problem, we scale the bandwidth usage $\tilde{f}(\mathcal{P},\sigma) = f(\mathcal{P},\sigma)S$. Similarly, when applying the solution of a scaled problem to its original problem, $f(\mathcal{P},\sigma) = \frac{\tilde{f}(\mathcal{P},\sigma)}{S}$.

Lemma 4.4.1. *Let $T^*(\mathcal{P}, f^*(\mathcal{P},\sigma),\sigma)$ be the optimal solution, $\forall p \in \mathcal{P}$, $T^*(\mathcal{P}, f^*(\mathcal{P},\sigma),\sigma) = T(p, f^*(p,\sigma),\sigma_p^*)$.*

Proof. We use a proof by contradiction. Suppose the claim is false. We first assume that p_i is the latest path to finish the transfer and p_j be any path that finishes earlier than p_i, σ_i and σ_j are the corresponding file sizes

Algorithm 2 SFTS-A$(G(V,E),s,t,\tau)$

```
1: procedure
2:     Construct the auxiliary graph G^A(V^A, E^A, τ).
3:     Solve the MAF formulation.
4:     if M(τ, G^A) ≥ σ then
5:         Compute the shortest s − t path p.
6:         x ← d(p) + 1; y ← τ
7:         while y ≠ x + 1 do
8:             τ̄ ← ⌊(x + y)/2⌋
9:             Construct the auxiliary graph G^A(V^A, E^A, τ̄).
10:            Solve the MAF formulation.
11:            if M(τ̄, G^A) ≥ σ then
12:                y ← τ̄
13:            else
14:                x ← τ̄
15:        τ̄ ← y
16:        return the s-t path set P obtained from the MAF solution with
       the corresponding flow allocation f(P, σ).
17:    else
18:        return FALSE
```

Algorithm 3 Scaled SFTS-A [SSA] $(G(V,E),s,t,\sigma,U,L,\epsilon)$

```
1: procedure
2:     S ← Lε/(n+1)
3:     G̃(Ṽ, Ẽ) ← G(V, E)
4:     for each e ∈ Ẽ do
5:         define d̃(e) ≡ ⌊d(e)/S⌋ + 1
6:         define b̃(e) ≡ b(e)S
7:     Ũ ← ⌊U/S⌋ + n + 1
8:     return SFTS-A(G̃(Ṽ, Ẽ), s, t, σ, Ũ)
```

transferred over p_i and p_j. Since $T^*(p_i, f^*(p_i,\sigma),\sigma_i) > T^*(p_j, f^*(p_j,\sigma),\sigma_j)$, we can always find a new file distribution $f'(\mathcal{P},\sigma)$ that $\sigma'_i+\sigma'_j=\sigma_i+\sigma_j$, $T'(p_i,f'(p_i,\sigma),\sigma'_i) = T'(p_j,f'(p_j,\sigma),\sigma'_j)$ and other paths remain the same. Obviously, $T'(p_i,f'(p_i,\sigma),\sigma'_i) < T^*(p_i,f^*(p_i,\sigma),\sigma_i)$, which means the optimal result can still be improved. This is a contradiction. □

Lemma 4.4.2. *Let p be any path, and $\tilde{d}(p)$ is the scaled path delay, then $d(p) \le \tilde{d}(p)S \le d(p)+nS$.*

Proof. For each $e\in E$ we have $d(e)/S \le \tilde{d}(e) \le d(e)/S+1$ hence $d(e) \le \tilde{d}(e)S \le d(e)+S$ and

$$d(p)=\sum_{e\in p} d(e) \le S\sum_{e\in p}\tilde{d}(e) = \tilde{d}(p)S \le d(p)+nS.$$ □

Algorithm 4 BFMS-A $(G(V,E), s, t, \sigma, UB, LB, \epsilon)$

```
1: procedure
2:     B_L ← LB
3:     B_U ← ⌈UB/2⌉
4:     while B_U/B_L > 2 do
5:         B ← (B_L · B_U)^{1/2}
6:         if D(1, B) = YES then
7:             B_L ← B
8:         else
9:             B_U ← B
10:    return SSA(G(V,E), s, t, σ, 2B_U, B_L, ε)
```

Lemma 4.4.3. *Any path set $\mathcal{P}$ returned by Algorithm SSA (Algorithm 3) satisfies*

$T^* \le T(\mathcal{P}) \le U + (n+1)S = U + L\epsilon.$

Proof. By definition, $T^* \le T(\mathcal{P})$. Since $\tilde{T}(\mathcal{P}) \le \tilde{U}$, we have $\tilde{T}(\mathcal{P})S \le \tilde{U}S \le U + (n+1)S = U + L\epsilon$. □

Lemma 4.4.4. *If $U \ge T^*$ then Algorithm SSA (Algorithm 3) returns a feasible path set, $\mathcal{P}$, that satisfies $T(\mathcal{P}) \le T^* + L\epsilon$.*

Proof. For each path $p^* \in \mathcal{P}^*$ and for each $e \in p^*$, we have $\tilde{d}(p^*) \le d(p^*)/S + n$. Let p_i^* be the latest path to finish the transfer when apply the optimal path set $\mathcal{P}^*$ to the scaled problem. Thus,

$$\begin{aligned}
\tilde{T}(\mathcal{P}^*) &= \tilde{d}(p_i^*) + \frac{\sigma_{p_i^*}}{f^*(\sigma, p_i^*)S} \\
&\le d(p_i^*)/S + n + \frac{\sigma_{p_i^*}}{f^*(\sigma, p_i^*)S} \\
&= T^*/S + n \\
&\le U/S + n \\
&\le \tilde{U}
\end{aligned}$$

Since the algorithm finds an optimal solution, we must have $\tilde{T}(\mathcal{P}) \le \tilde{T}(\mathcal{P}^*)$. Combining with Lemma 4.4.2 we get

$$\begin{aligned}
T(\mathcal{P}) &\le \tilde{T}(\mathcal{P})S \\
&\le \tilde{T}(\mathcal{P}^*)S \\
&= \tilde{d}(p_i^*)S + \frac{\sigma_{p_i^*}}{\tilde{f}^*(\sigma, p_i^*)}S \\
&\le d(p_i^*) + \frac{\sigma_{p_i^*}}{f^*(\sigma, p_i^*)} + nS \\
&= T(\mathcal{P}^*) + ns \\
&\le T^* + L\epsilon
\end{aligned}$$

$\square$

Lemma 4.4.5. *The test*

$$D(1,B) = \begin{cases} YES & \text{if } SSA(G(V,E), s, t, \sigma, B, B, 1) \text{ returns FALSE} \\ NO & \text{otherwise} \end{cases}$$

is a 1-test.(I.e., an ε-test with $\varepsilon = 1$).

Proof. We have to proof that SSA$(G(V,E), s, t, \sigma, B, B, 1)$ returns FALSE when $T^* > B$, otherwise $T^* \leq 2B$. The first part follows from Lemma 4.4.4 with $U = B$ and $\varepsilon = 1$, since if $T^* \leq B$, SSA must return a feasible path set. The second part follows from Lemma 4.4.3 since $L = U = B$ and any path set returned by SSA satisfies $T^* \leq T(\mathcal{P}) \leq U + (n+1)S = U + L\varepsilon = 2B$. $\square$

Complexity. Due to the introduction of the time-expanded network, the size of the MAF formulation in Algorithm 2 is linearly depended on the time bound τ [8]. We denote the time complexity of Algorithm 2 to be $O(poly \cdot \tau)$, where $O(poly)$ is polynomially bounded by m and n.

Complexity. In Algorithm 3, we call Algorithm 2 with $\tau = \tilde{U}$. With the binary search style, the MAF formulation can be solved at most $\log(\tilde{U})$ times. The overall complexity of Algorithm 3 is

$O(poly \cdot \tilde{U}\log(\tilde{U})) = O(poly \cdot (\frac{nU}{\varepsilon L} + n)\log(\frac{nU}{\varepsilon L} + n))$.

If we have $U \geq L$, and $\epsilon \leq 1$, then

$O(poly \cdot \log(\tilde{U})) = O(poly \cdot \frac{nU}{\varepsilon L}\log(\frac{nU}{\varepsilon L}))$

Complexity. Calling Algorithm 3 with $U = L = B$ and $\epsilon = 1$ requires $O(poly \cdot n\log(n))$ steps.

Theorem 4.4.6. *Given valid bounds $0 < LB \leq T^* \leq UB$, an ε-approximate solution can be found in $O(poly \cdot (\frac{n}{\varepsilon}\log\frac{n}{\varepsilon} + n\log n \log\log(\frac{UB}{LB}))$.*

Proof. Consider Algorithm 4. We first find a lower bound B_L, and an approximate upper bound B_U such that their ratio $B_U/B_L \leq 2$ and $2B_U$ is a valid upper bound. Then we call Algorithm 3 with the two bounds.

The bounds are found using a binary search on $\log B_L$, $\log B_U$ in the range $\log LB$ to $\log(UB/2)$, which terminates when $B_U/B_L \leq 2$. After each iteration the new value of $\log(B_U/B_L)$ is either $\log(B_U B_L)^{1/2}/B_L$ or $\log(B_U/(B_U B_L)^{1/2})$. Since $\log(B_U B_L)^{1/2}/B_L = \log(B_U/(B_U B_L)^{1/2}) = \frac{1}{2}\log B_U/B_L$, the new value of $\log(B_U/B_L)$ is half of the old one. Therefore, the binary search requires $O(\log\log(UB/LB))$ tests. Each test requires $O(poly \cdot n\log n)$, thus, we can find the final values of B_L and B_U in $O(poly \cdot n\log n\log\log(UB/LB))$ steps.

By Lemma 4.4.5, $D(1,B)$ is a valid test, thus lines 7 and 9 in Algorithm 4 ensure that at each iteration B_L is a valid lower bound and $2B_U$ is a valid upper bound. After we call Algorithm 3 with valid bounds, by Lemma 4.4.4, $T(\mathcal{P}) \leq T^* + B_L\varepsilon \leq T^*(1+\varepsilon)$. The call to Algorithm 3 with $U/L = 2B_U/B_L = O(1)$ requires $O(poly \cdot \frac{n}{\varepsilon}\log\frac{n}{\varepsilon})$. $\square$

Chapter 5

Proposal and Design

This chapter explains the design of our proposed big file transfer engine. The traditional way of transferring a single file is calculating one path and transferring all the file data on that path. The goal of our file transfer mechanism is to find multiple paths between the source and destination, divide the file data properly on each path and transfer them as fast as possible. Additionally, we also make use of our network monitoring function to deal with the network status changes by using dynamic path control. The whole project contains about 2000 lines of codes.

5.1 System design

The file transfer engine makes use of SDN as presented in Figure 5.1. The advantage of SDN's centralized design provides us an easy way to monitor and deal with variable network status and makes it possible for us to find a better file scheduling strategy. The engine is composed of three layers. The first layer is the infrastructure layer, which has several OpenFlow-enabled switches. Those switches provide the topology of the network in our system. The second layer is the control layer in which the Ryu controller [26] is used to control and communicate with the switches in the infrastructure layer through OpenFlow protocol. The Ryu controller also provides northbound APIs to the third layer, the application layer, to run applications on top of the control layer. The application layer consists of two components: (1) a monitoring component and (2) a forwarding component.

Figure 5.2 shows how modules at the application layer work with each other and how controller and switches work with the applications. Since knowing the current network state is needed for deciding which paths we want to use, a monitoring component is added to the system to collect the relative statistics from all the OpenFlow switches in the infrastructure layer through the controller. After getting topology, delay and bandwidth data from the monitoring component, the forwarding component calculates

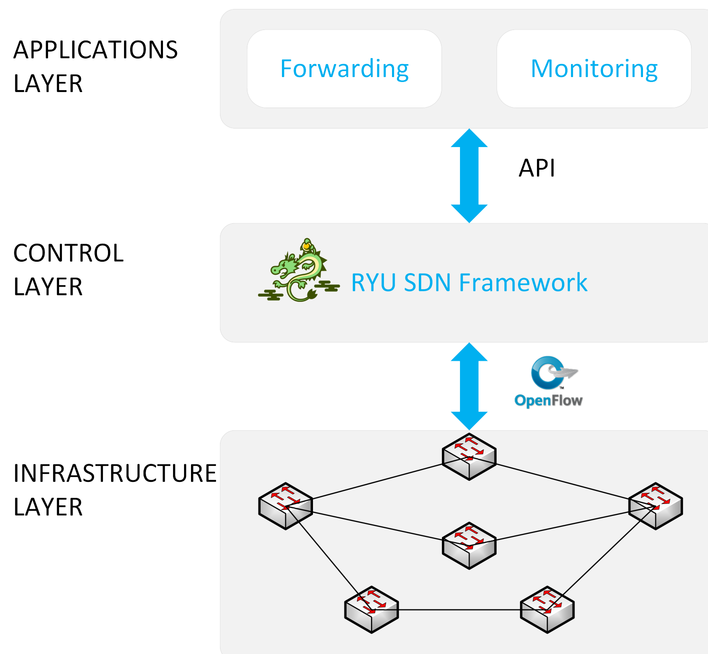


Figure 5.1: The architecture of our SDN-based file transfer system

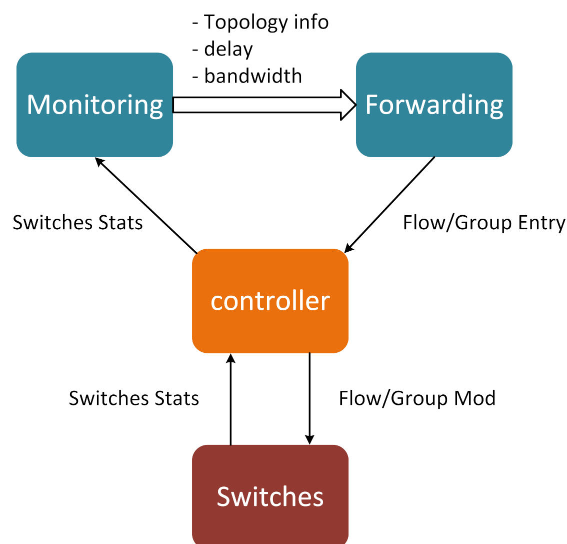


Figure 5.2: Application layer

the best paths based on the delay and also, it should properly allocate a portion of the file to each path to reduce the total transfer delay. Following the calculation, the controller generates and installs the corresponding flow entries or group entries to each OpenFlow switch in the infrastructure layer.

5.2 Monitoring component

Beside basic topology information, there are two kinds of network information needed, namely link delay and link load. This means the monitoring component should be able to get those data from the statistics provide by the OpenFlow switches. Since OpenNetMon [27] has been proved to have enough accuracy and has been used in many other SDN applications where monitoring is needed [4], we adopt the similar techniques used in OpenNetMon to implement our monitoring module.

5.2.1 Link Delay

The OpenFlow protocol does not require the switches to provide any information about link delay directly. Thus, the link delay can only be estimated by testing the Round-Trip Time (RTT) on each link. The monitoring component gets the estimated link delay by two steps, measuring the time packet travel on the circle of controller-link-controller and measuring the packet travel time between controller and switch as shown in Figure 5.3.

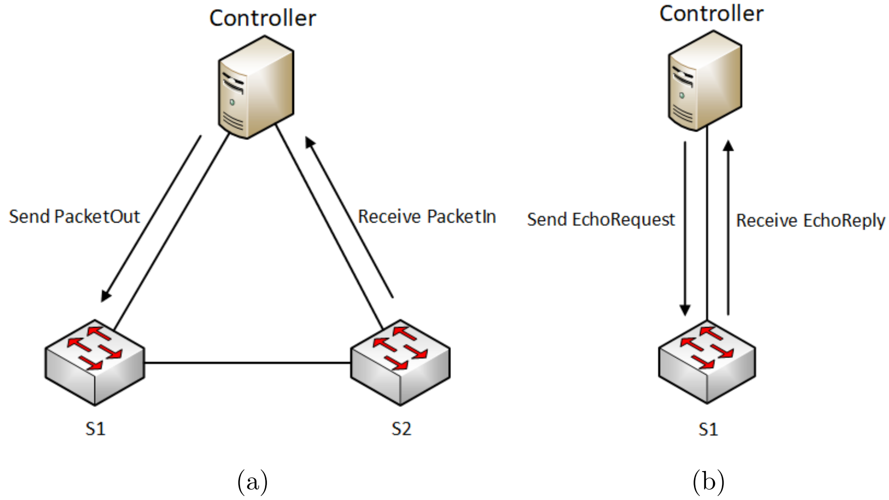


Figure 5.3: Process of detecting link delay

Figure 5.3a shows the procedure of measuring the time D_{circle} that a packet travels through a controller-link-controller circle. The link between s1 and s2 is the link being estimated. The controller packs the current time t_1 into a packet, and sends s1 a *PacketOut* message with that packet and lets s1 send the packet to s2. When s2 receives the packet, it then triggers a *PacketIn* message and sends the packet back to the controller. The controller records the time t_2 when it receives the packet back. Then we can get $D_{cricle} = t_2 - t_1$.

Figure 5.3b illustrates the procedure of measuring the communication time D_{s1} between the controller and switch s1. The controller first packs current time t_3 with an *EchoRequest* message and sends it to the switch s1. Switch s1 then sends an *EchoReply* message back after it receives the *EchoRequest* message. The controller then records the current t_4 when it received the reply. D_{s1} can be calculated as $(t_4 - t_3)$. A similar procedure can be applied to the controller and switch s2; the communication delay between them is denoted as D_{s2}.

The delay on the link between s1 and s2 can be calculated as $D_{circle} - (D_{s1} + D_{s2})/2$.

5.2.2 Link Load

Link load can be measured by counting how much data are transferred on that specific link in a fixed time interval $t_{interval}$. The data transferred on a link equals the data sent or received by the ports connected to that link, which can be requested by the *ofp_port_stats_request* message. For example, in Figure 5.4, the load of link s1 to s2, $link_{s1 \to s2}$, will be measured. In every time interval $t_{interval}$, the controller sends an *ofp_port_stats_request* message to the egress port of $link_{s1 \to s2}$ on switch s1. The total transmitted data, Num_{now}, can be extracted from the reply message, assuming in the previous time interval we get the data Num_{pre}, then the link load can be estimated as $(Num_{now} - Num_{pre})/t_{interval}$.

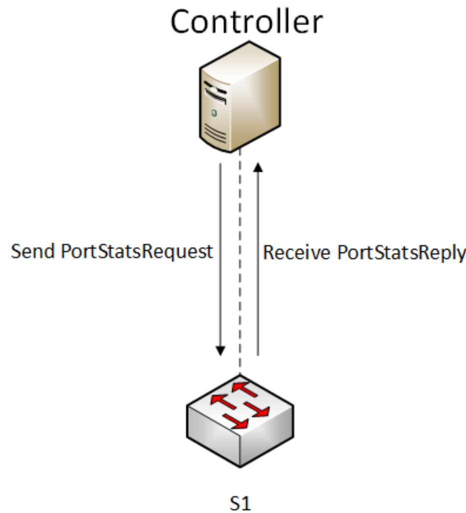


Figure 5.4: Process of link load monitoring

5.3 Forwarding component

Forwarding component is the core of the system. It can take advantage of the information gathered by our monitoring component and run the preset algorithm to decide which path or paths are going to be used and how to separate the file for each path. In our case, we use the two algorithms proposed in Chapter 4. After getting the result, the forwarding component then dispatches flow entries and group entries to switches on the selected paths to make them start the transfer.

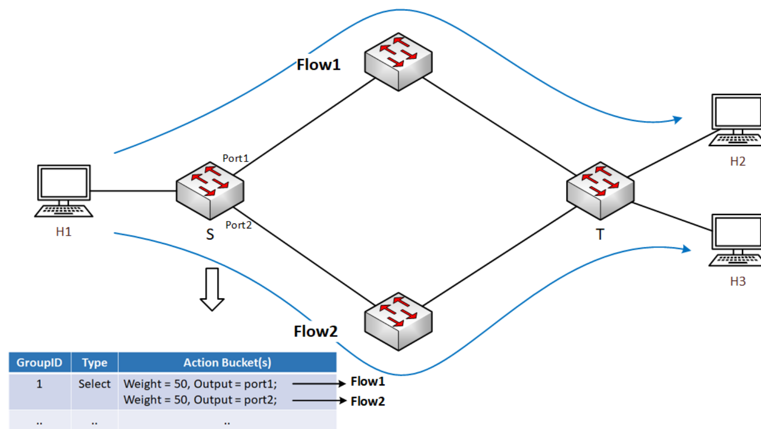
5.4 Enable multipath forwarding in Open vSwitch

Although a lot of multipath research has been done in the SDN domain, they either did it at the flow level [5] [30] [24], which means packets from the same flow using the same path, or work together with MPTCP [23] [6] [2], of which the usage can be limited by the end systems. How to realize packet level multipath forwarding in SDN without the help of MPTCP remains a problem. In this section, we propose a novel implementation method to achieve packet level weighed multipath forwarding in normal OVS-enabled switches to fit our big file transfer engine.

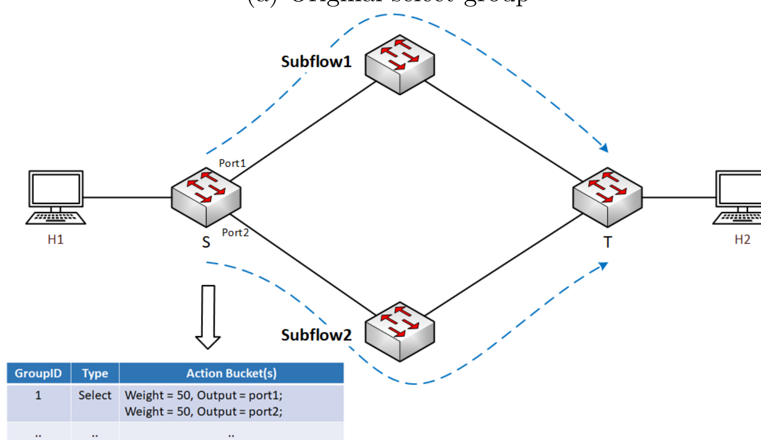
Multipath forwarding can only be achieved when the switch supports it. However, Open vSwitch does not originally support multipath forwarding at the packet level. Some modifications and other methods are needed to make the switches work according to our routing algorithm. The modified source codes can be found in Appendix A.

5.4.1 Modify group bucket selection algorithm

With a group table configured to the select type, the switch can achieve load balancing by forwarding packets to different ports. In Open vSwitch and most OpenFlow-enabled switches, the bucket selection is done at the flow level. The selection uses a hash algorithm on the mac address of the destination host. As a result, the packets coming from the same flow will be attached to the same action bucket and it cannot perform multipath forwarding within a single flow. For example, in Figure 5.5a two flows go from h1 to both h2 and h3. To balance the load, a group table with two action buckets to each outgoing port is set on switch s. Since the two flows have different destination hosts, s will select different action buckets for each flow, and then the load can be well balanced over the two paths. However, the packets coming from one flow have the same destination mac address, and then the same action bucket will be selected for all packets in that flow. So multipath forwarding cannot be achieved on Open vSwitch with the original group bucket selection algorithm.



(a) Original select group



(b) Modified select group

Figure 5.5: Example of original and modified select group in Open vSwitch

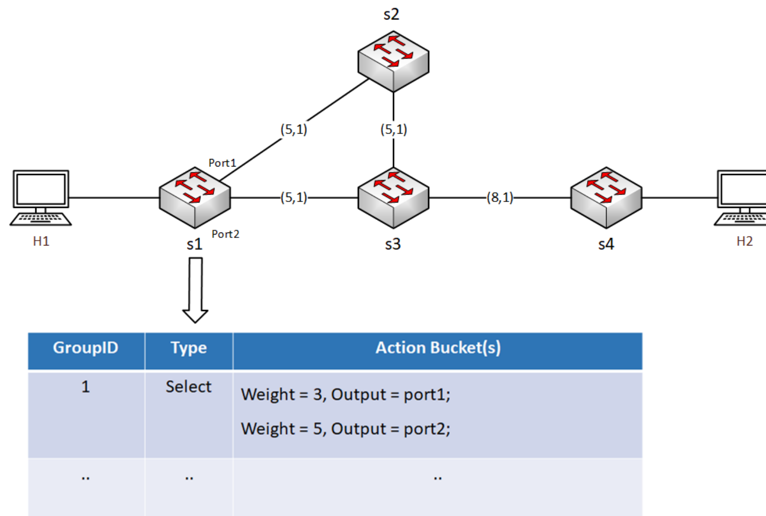
To extend the bucket selection to the packet level, the original hash selection algorithm is replaced by a random selection algorithm. Then all the packets match to a group table with select type, no matter whether they come from the same flow or not, and will select an action bucket randomly. Now, for each flow that we want to transfer using multipath, we set up an exclusive group table on each switch. The packets from that flow then can be all matched to this group table and randomly select one action. For example, in Figure 5.5b, there is one flow from h1 to h2. To perform multipath forwarding, the same group table is configured on s as in Figure 5.5a. We only match the packets from this flow to this group table so that other flows would not be influenced by this multipath setting. Since the modified selection algorithm works on each packet instead of each flow, all packets will randomly select an action bucket and then forward to different ports. OpenFlow does not specify how to select a bucket in a group table, so the

design is still OpenFlow compatible.

5.4.2 Ensure file partition and bandwidth usage

In the our algorithms, not only multipath, but also the file size transferred in each path is settled to reduce the overall delay in the file transfer. To help the switches control the number of packets forwarded to each egress port and meet the requirement of file partition and output speed from the algorithm, other methods like queue and group weight are introduced.

To properly divide the ingress flow to each egress port, group bucket weight can be set according to the portion of bandwidth that is going to be used in each path to control the process of the bucket selection. A simple example is shown in Figure 5.6, assuming a 25 Mb file will be sent from h1 to h2, the bandwidth and delay of each link is shown in the (Mbps, second) format above that link. The algorithm shows the optimal solution will be transferring 6Mb file at a speed of 3Mbps on the path s1-s2-s3-s4 and 15Mb file at 5Mbps on the path s1-s3-s4. In theory, the total delay of the transfer will be 5 seconds. Then group bucket weight in s1 can be set according to the selected bandwidth on each path, thus 3Mbps and 5Mbps respectively in the given example. Since the ingress data speed equals the total of all egress ports speeds, the egress port bandwidth can be calculated as $ingressSpeed \times weidthOfPath/totalWeight$. In the example, the maximum total bandwidth is 8Mbps, so the used bandwidth on each path will be 3Mbps and 5Mbps.



GroupID	Type	Action Bucket(s)
1	Select	Weight = 3, Output = port1; Weight = 5, Output = port2;
..	..	..

Figure 5.6: Example of file partition using select group.

When the bandwidth of each path meets the requirement of the algorithm, the file size transferred on each path should also meet the requirement. On

path s1-s2-s3-s4, the actual working time is 2 seconds (total transfer delay - total path delay), so the data transferred on the path is $3Mbps \times 2seconds = 6Mb$, and the same way shows 15Mb file transferred on path s1-s3-s4.

A more accurate way to control the bandwidth is to combine the group bucket weight with the OVS queue management. In [15], OVS queue management is used to provide guaranteed bandwidth reservation. This can also be introduced to our project to achieve more accurate and guaranteed file partition. However, in our virtual testing environment, the usage of queue is limited, so the queue is left behind as an improvement of the current design.

5.5 Enable dynamic control in path selection

Network state varies over time, low load paths may meet congestion and flows on the heavy load paths may also finish their work and make the paths a better choice during the file transfer. A dynamic change on the current path selection can help our system to take advantage of current network state.

5.5.1 Process of dynamic path control

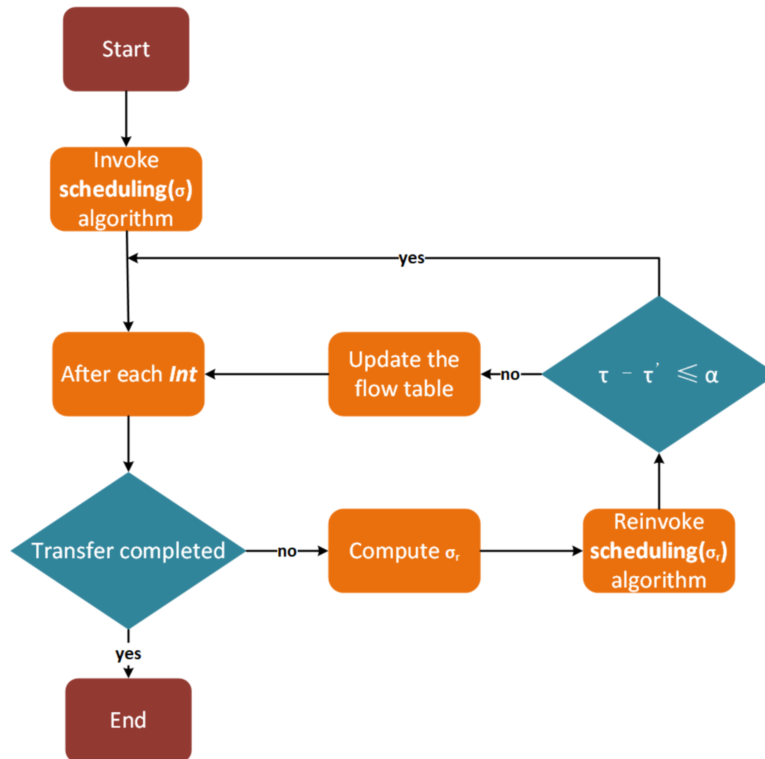


Figure 5.7: Dynamic path control process.

To enable the dynamic control in path selection, two additional parameters are introduced. One is the detection interval Int, another is the changing threshold α. The detection interval is used to control the frequency of the topology detection. After each Int, if the transfer is not completed, then we compute the remaining file size σ_r and recompute our algorithm according the latest topology information. The changing threshold controls whether we need to update our new path set or not. After the recomputation, we will get new transfer time τ'. If the absolute difference between the new transfer time τ' and current expected remaining transfer time τ is greater than the threshold α, we update the flow tables and group tables.

Chapter 6

Evaluation

This chapter shows the evaluation results from the proposed big file transfer engine. We start with a complexity evaluation of our algorithms to show they are practically usable and then test our entire proposed engine in an SDN environment. The network experiments begin with a description of the experiment scenario and tools used to build it, and finishes with the obtained results. First, an experiment of multipath file transfer over different redundant paths is conducted to show the performance of the proposed model when multiple paths are available. After that the influence of delay on each path will be evaluated. Finally, we test our engine in a variable network environment to show its abilities to make use of the updating network status to transfer the file.

6.1 Algorithm test

Since our big file transfer engine is expected to be used in a real network situation, the running time efficiency of the algorithms is important to decide whether our engine is usable or not. In this section, we test the complexity of our two scheduling algorithms, and we also show errors introduced in our approximation algorithm.

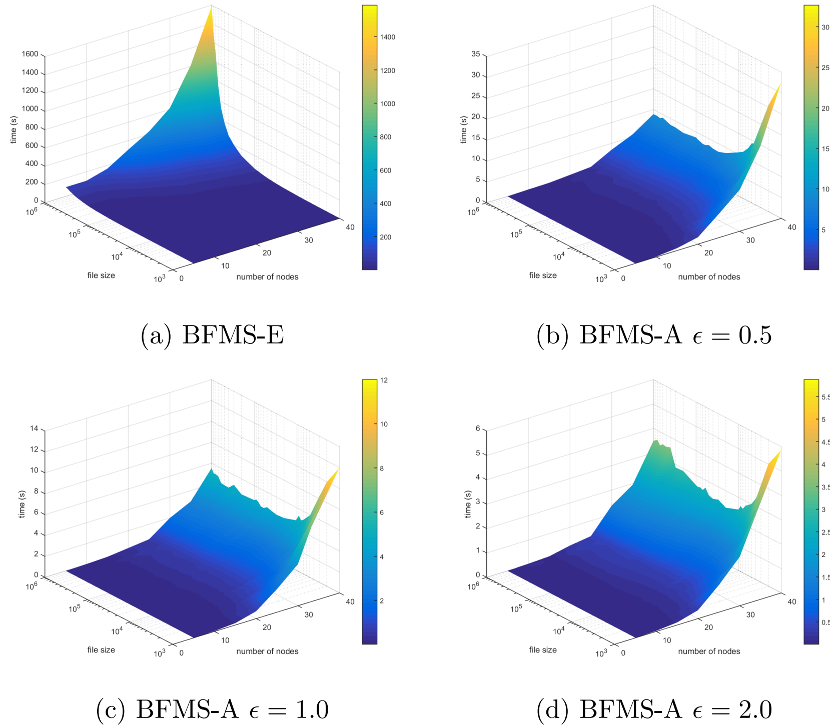
In all the BFMS-A algorithm tests, we set UB to be the time that transferring all the file on the shortest path between s and t, since the shortest path must be a possible solution. We set LB to be the time to transfer all the file with smaller one between the sum of all egress bandwidth of node s and the sum of all ingress bandwidth of node t. This is based on the fact that no multipath solution can achieve higher throughput than the total egress bandwidth of node s and the total ingress bandwidth of node t.

6.1.1 Random graph

Test topology

To show the results in a more general situation, we test our algorithm on Erdős-Rényi graphs [7]. The model can be used to generate a random graph $G(n,p)$, where n is number of the nodes in the graph and p is the connectivity probability between each pair of nodes. We test our algorithms with several random graphs with n changing from 5 to 40, and we set p to be 0.5 to provide enough redundant paths between our source and destination nodes. The link delay $d(e)$ is uniformly distributed between 100 and 500 units, and the bandwidth $b(e)$ is uniformly distributed between 5 to 10 units. For each graph, we randomly select two nodes and transfer files varying from 1000 to 1000000 units between them.

Test result



(a) BFMS-E

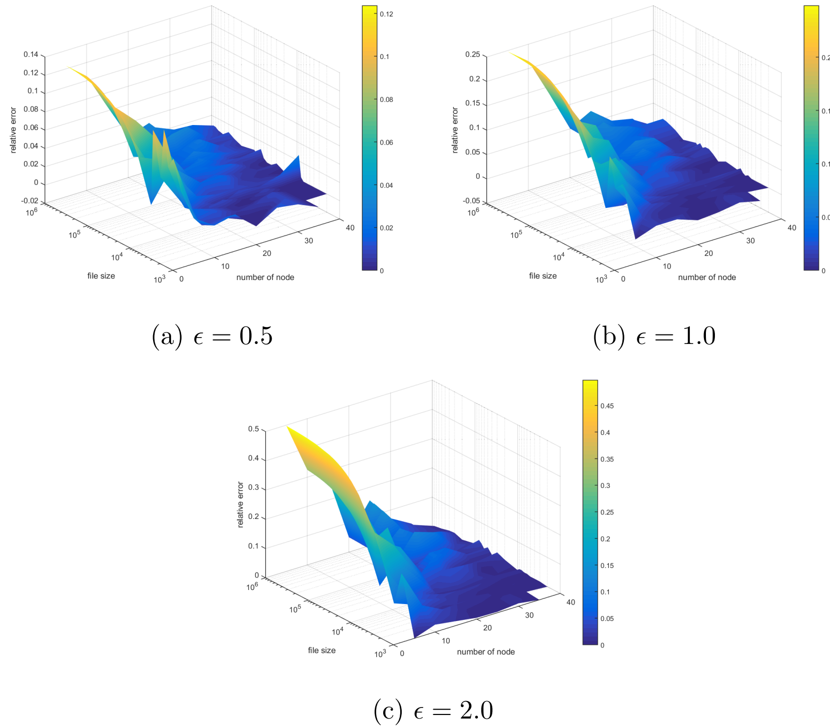
(b) BFMS-A $\epsilon = 0.5$

(c) BFMS-A $\epsilon = 1.0$

(d) BFMS-A $\epsilon = 2.0$

Figure 6.1: Running time results on the random graphs.

Figure 6.1 shows the running time results of BFMS-E and BFMS-A with $\epsilon = 0.5$, 1.0 and 2.0 on different sizes of random graphs. In all the figures, we increase the file size exponentially to show the time complexity of BFMS-E in the worst case and the advantage of BFMS-A. In Figure 6.1a, due to



(a) $\epsilon = 0.5$

(b) $\epsilon = 1.0$

(c) $\epsilon = 2.0$

Figure 6.2: Relative errors in BFMS-A on the random graphs.

the essential of a pseudo-polynomial algorithm, the running time increases dramatically when the problem size increases exponentially. In the biggest problem setting where the file size is 1000000 and the nodes number is 40, BFMS-E needs 1588 seconds to solve the problem on our test machine. To the contrary, BFMS-E performs much better when the file size grows. When nodes number is small, the file size almost has no influence on the running time. And when the nodes number increases to more than 20, we can even find that the running time decreases when files size goes bigger. This is caused by our setting of UB and LB. When the file size is small, the propagation time is also small. In this case, The delay of the shortest path has a bigger influence on UB, which makes the ratio of UB and LB also larger. Although BFMS-A may need more time when file size is small, the overall performance is still much better than BFMS-E. Even in the worst case, BFMS-A only needs 32 seconds, 12 seconds and 5.8 seconds when ϵ is set to be 0.5, 1.0, and 2.0.

Figure 6.2 and Figure 6.3 show the relative errors of BFMS-A when compare to the results from BFMS-E. In Figure 6.2, we can see that the highest errors are all shown when nodes number is small. Figure 6.3 shows that when ϵ increases, the relative error also rises, although no test case shows an error that is close to the theoretical maximum error.

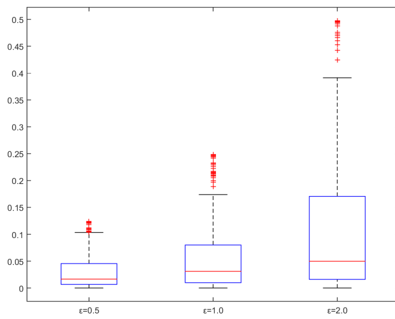
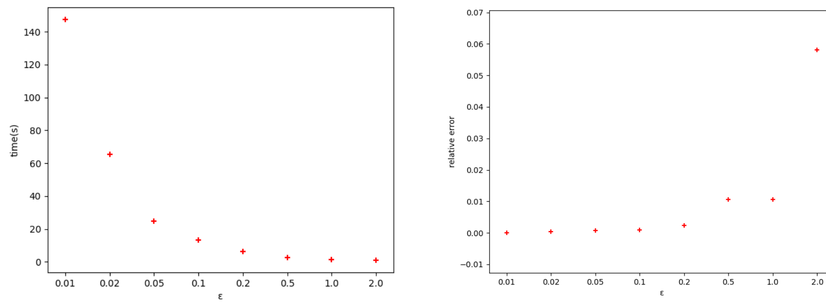


Figure 6.3: Relative errors in BFMS-A with different ϵ



(a) running time (b) relative error

Figure 6.4: Results of tests with 30 nodes and 1000000 units file when ϵ changes.

To show the influence of different ϵ in a clear way, we also test the scenario, 30 nodes and 1000000 units file, with a varying ϵ. Figure 6.4a shows how the running time changes with different ϵ. When we set $\epsilon = 0.01$, the algorithm still needs more than 140 seconds to solve the problem, although we can see, in Figure 6.4b, there is no error introduced to the result. The running time drops very fast when we increase ϵ, the problem can be solved in seconds when ϵ is higher than 0.2. The relative error also increases, but still in a very low level (less than 0.1) in our test scenario.

6.1.2 Square lattice graph

Test topology

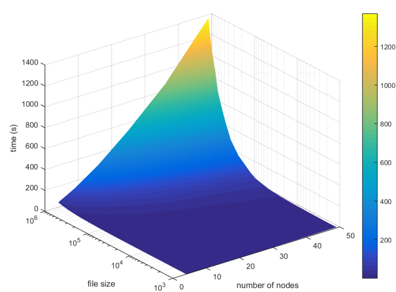
Now we test our algorithms on n $\times$ n square lattice graphs. The number of n increases from 2 to 7 in different tests. The link delay $d(e)$ is uniformly distributed between 100 to 500 units, and the bandwidth $b(e)$ is uniformly distributed between 5 to 10 units. For each graph, we transfer files from the top left node to the bottom right node. The file size varies from 1000 to 1000000 units.

Test result

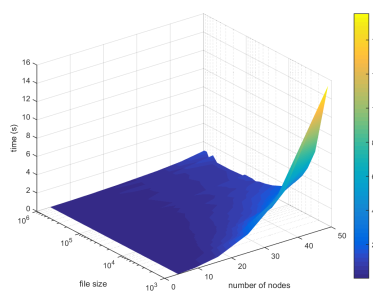
Figure 6.5 presents the running time test results of BFMS-E and BFMS-A on the square lattice graphs. The results are similar to those in the random graphs. We can see that BFMS-A shows great advantages when the problem size is relatively large. In the largest problem, it costs BFMS-E 1369 seconds to solve it, while for BFMS-A, it only uses 15.5 seconds, 6.5 seconds and 3.3 seconds when $\epsilon = 0.5$, 1.0, and 2.0 in the worst cases. One difference we find when compared to the random graph tests is that the running time of BFMS-E on square lattice graphs dose not increase as fast as that on the random graphs. This is caused by the reason that the number of edges increases exponentially in the random graphs, while in the square lattice graph, the increment is still polynomially bounded.

The relative errors shown in Figure 6.6 also do not change much from the results of the random graphs. Bigger errors are shown when the graph is smaller. Consider all the test cases with nodes number to be 4, when we set $\epsilon = 0.5$ and 1.0, we can still find that no error is introduced to the result in some test, however, when ϵ increases to 2.0, all test results show errors. The maximum relative errors in tests with $\epsilon = 0.5$, 1.0, and 2.0 are 0.16, 0.34, and 0.34 respectively. The maximum relative errors when $\epsilon = 1.0$ and 2.0 are close, but in Figure 6.7, we can still find the results with $\epsilon = 1.0$ are better, when we compare all tests.

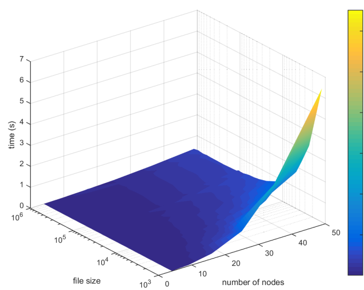
Figure 6.8 shows the results of test with different ϵ on a square lattice graph with 36 nodes and transferring a 1000000 units file. The outcome is



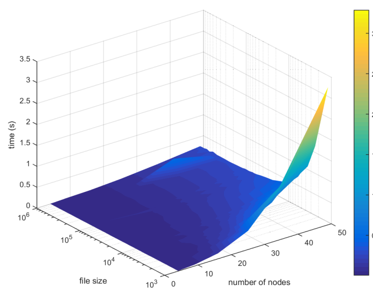
(a) BFMS-E



(b) BFMS-A $\epsilon = 0.5$

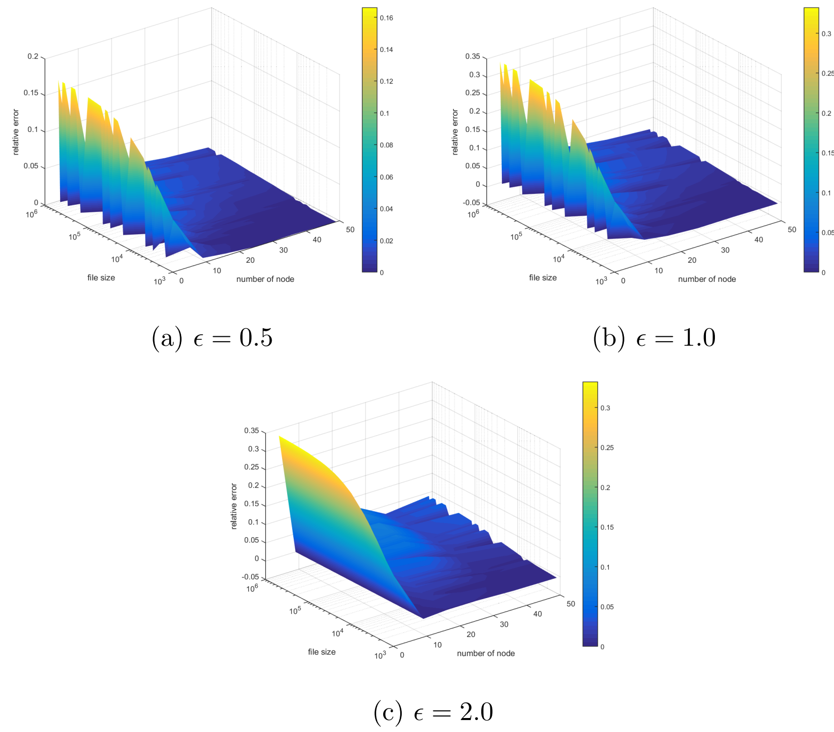


(c) BFMS-A $\epsilon = 1.0$



(d) BFMS-A $\epsilon = 2.0$

Figure 6.5: Running time results on the square lattice graphs.



(a) $\epsilon = 0.5$ (b) $\epsilon = 1.0$

(c) $\epsilon = 2.0$

Figure 6.6: Relative errors in BFMS-A on the square lattice graphs.

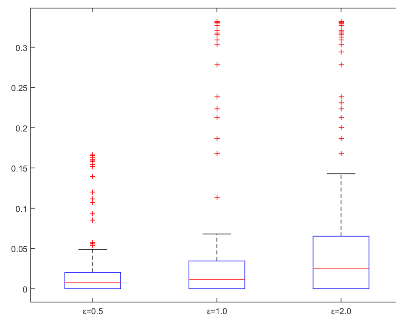
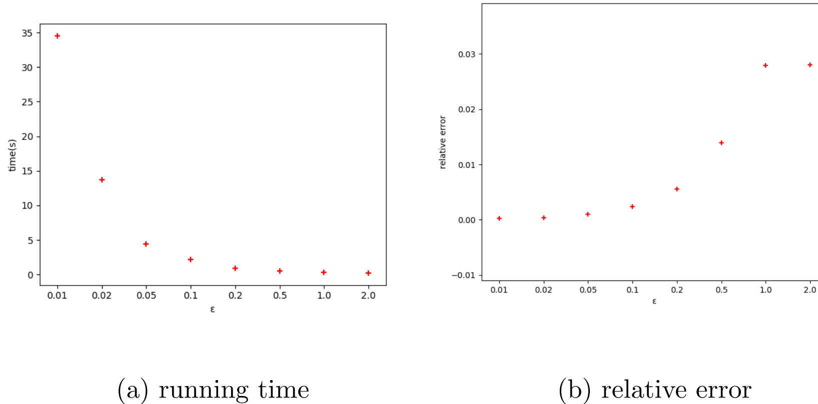


Figure 6.7: Relative errors in BFMS-A with different ϵ.



(a) running time (b) relative error

Figure 6.8: Results of tests with 36 nodes and 1000000 units file when ϵ changes.

similar to our previous random graph test. The increment on ϵ results on a rapid decrease on the running time, but slightly raises the relative error.

6.2 SDN network experiments

In our previous tests, We have established that Algorithm BFMS-E can be used when the problem size is relatively small, while Algorithm BFMS-A shows more advantages in some big size problems. Now we test our file transfer engine in SDN networks set up by Mininet and Open vSwitches. Due to the limitation of our current experiment environment, we cannot create a very complex scenario. We only show the results with Algorithm BFMS-E, which is more suitable for our small size tests.

6.2.1 Test over different number of paths

In theory, multipath file transfer can utilize more free bandwidth and free paths to achieve a decrease on the file transfer delay. This test evaluates the performance of the proposed multipath file transfer model when different numbers of redundant paths are available.

Experiment setup

In this experiment, different numbers of redundant paths need to be created for each test, so the topology of the testing network varies from test to test. Figure 6.9 illustrates the example topology when three redundant paths are available for the transfer. Two hosts h1 and h2 are connected to the source switch s1 and the destination switch s5 respectively. The file is sent from

h1 to h2. Three switches s2, s3 and s4 are connected to both s1 and s3 to provide three paths. Other tests in this experiment will also use a similar topology where the number of switches between the source and destination switches changes to provide different numbers of redundant paths. In each test of this experiment, the link delay is set to be 100ms and a file of 1GB is transferred through the network.

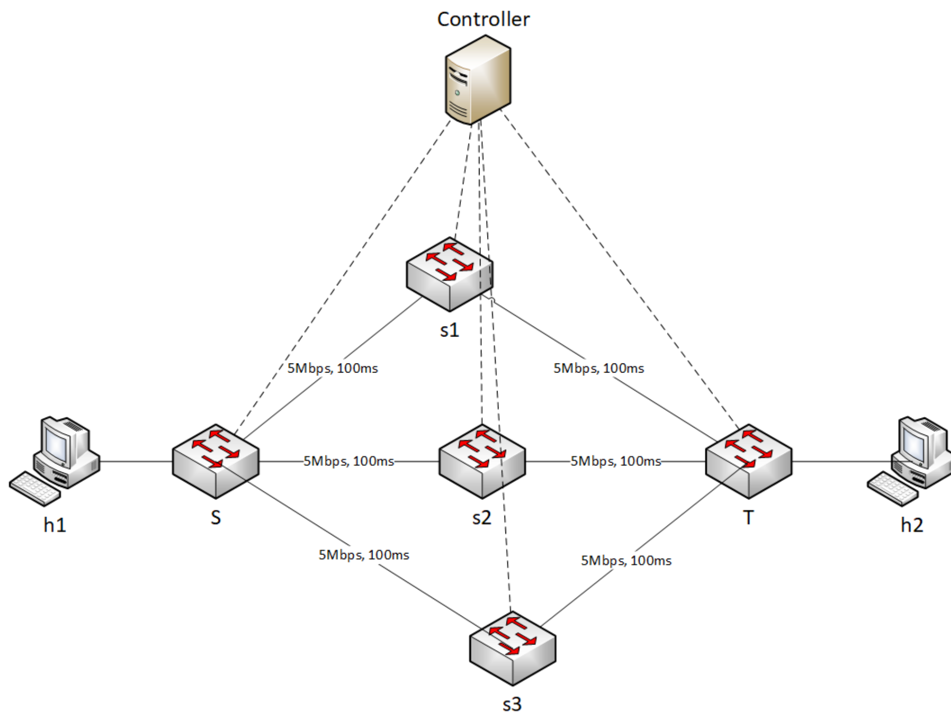


Figure 6.9: Example topology when 3 redundant paths are available.

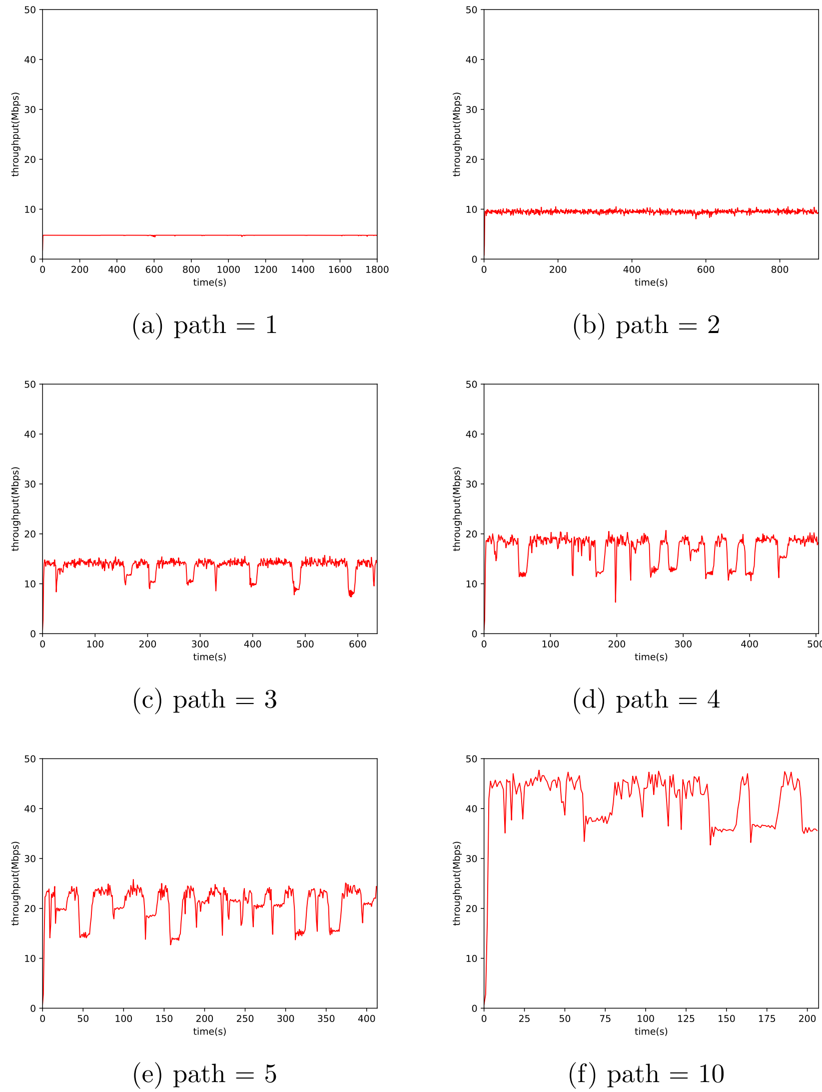
Experiment result

Figure 6.10 shows the test outcomes when different numbers of paths are provided. In each graph, the red line illustrates the real time transfer speed during the test, while the blue line shows the overall average throughput.

Although we can see the average throughput increases almost linearly when the number of paths goes up, the real time transfer speed fluctuates wildly as long as there are multiple paths available. According to previous research, we think the fluctuation may be a side effect of the multipath transfer. As stated in [11], the fluctuation could be due to the different delays in different paths, so packets sent later may arrive earlier. Duplicate acknowledgments are sent to trigger the fast retransmission. Part of the bandwidth is wasted to retransfer the packets that are not actually lost. So we set the Linux tcp_fack parameter to 0 to turn off TCP fast retransmission. Unfortunately, we do not get any considerable improvement after the change. We use Wireshark to inspect the network during the transfer, not much fast retransmission packets are shown in the tests. Instead, we capture a lot of regular retransmission packets. So our file transfer is suffering from network congestion due to the lack of available bandwidth.

Now we try to figure out the real performance of our system when no network congestion occurs. In [15], OVS egress traffic shaping was used to ensure the bandwidth reservation. However, in our experiments environment, Mininet is used to provide the virtual network environment and both Mininet and OVS use Linux netem to implement functions associated with QoS. To use the OVS egress traffic shaping, we would need to wipe out the previous configuration set by Mininet, which destroys our entire test environment. So, due to the limitation of our experiment environment, the same method can not be used in our experiments to achieve the bandwidth guarantee. Here we just slightly increase the bandwidth set on each link and use higher bandwidth to avoid network congestion. Though we have higher bandwidth now, we can still transfer the file according to the result of our algorithm. That means we do not take any advantage of the extra bandwidth for the file transfer, the extra bandwidth is only to ensure other flows do not occupy the bandwidth we need for the file transfer.

We redo the experiments with the same topologies mentioned before, the only change is the bandwidth on each link increases from 5Mbps to 6Mbps to avoid network congestion. Our system shows a remarkable result when bandwidth guarantee is provided. From Figure 6.11, we can see that in all tests, the process of each transfer is very stable. This supports our idea about the fluctuations shown in Figure 6.10 that our design does not introduce the fluctuations, we only make the TCP connection difficult to apply its current congestion control. The average transfer speed goes up linearly when more paths are available to be used.



(a) path = 1

(b) path = 2

(c) path = 3

(d) path = 4

(e) path = 5

(f) path = 10

Figure 6.10: Transfer speed over different path numbers

6.2.2 Test with delay different delays

Previous experiments showed that our multipath transfer design works quite well when each path has a similar delay. In this experiment, we test how different delays can influence the performance of our design.

Experiment setup

In this experiment, only two paths are provided to get rid of the influence of path numbers. As shown in Figure 6.12, links in the upper path have a

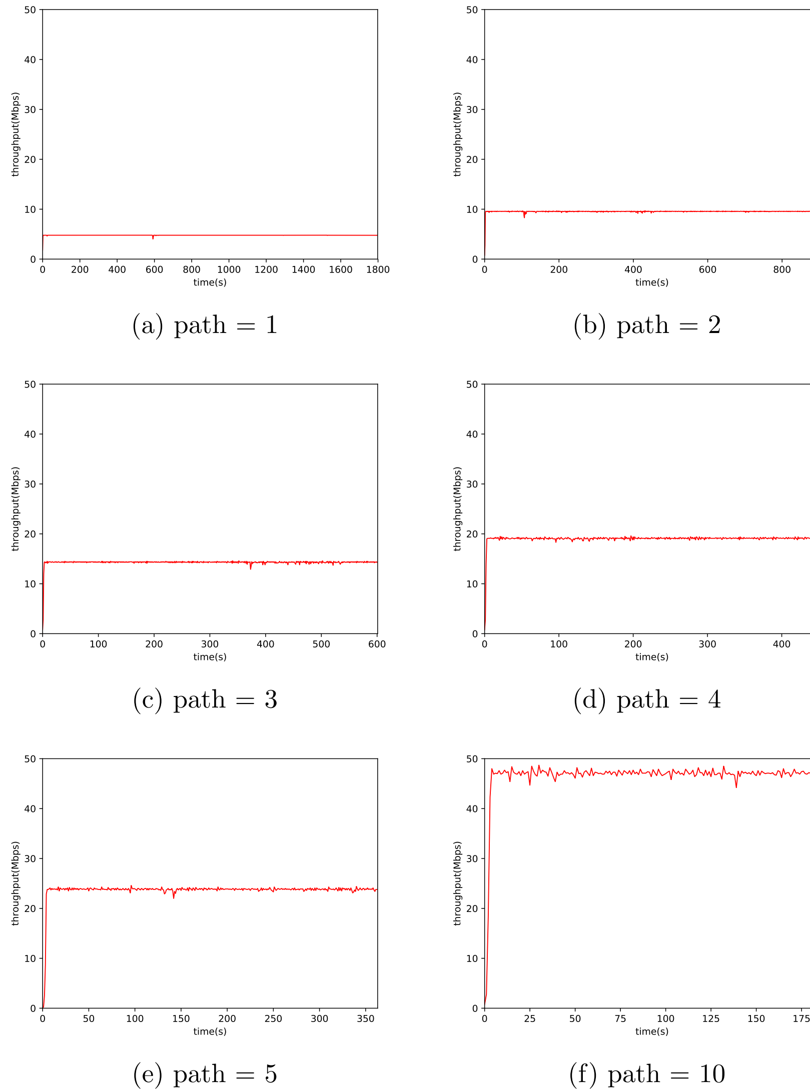


Figure 6.11: Transfer speed over different path numbers when bandwidth is guaranteed

fixed delay, 100ms, while the delays of links in the bottom path vary in each test. A file of 512MB is transferred from host h1 to host h2 during the tests.

Experiment Result

Figure 6.13 shows the test outcomes when increasing the link delay of the bottom path from 100ms to 600ms. When the link delay is less than 300ms, the delay has hardly any influence on the transfer and the advantage of multipath is fully taken by our system to reduce the transfer time. When

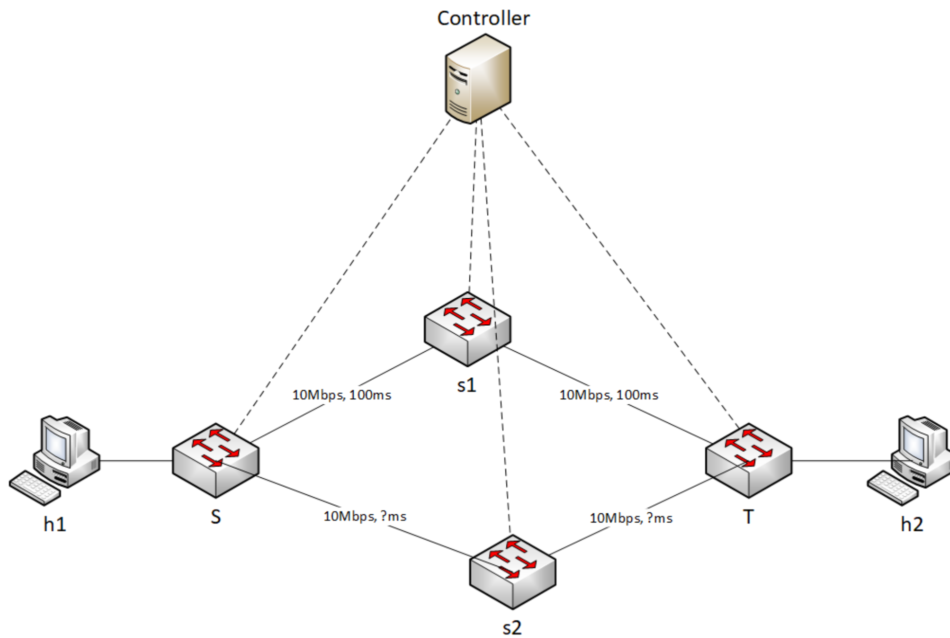


Figure 6.12: Topology of delay differential experiment

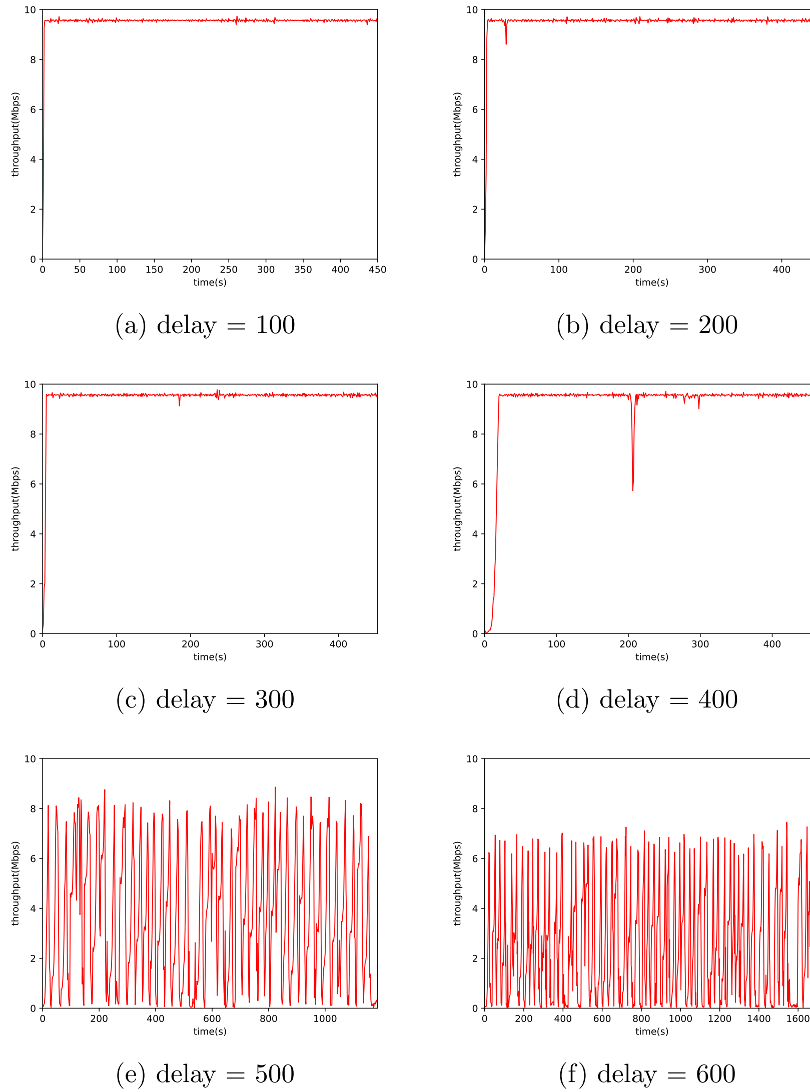
increasing the link delay to 400ms, the impact of delay on each path increases. The connection takes about 20 seconds to adjust itself to the current network, however, after it reaches the maximum speed, the transfer still is fairly stable. When we continue to increase the link delay to 500ms, the transfer becomes completely unstable. The connection can not adjust itself to reach a stable state and transfer the file at maximum speed. The average transfer speed is only 3.62Mbits/sec, which is worse than a normal single path transfer. An even worse result with a average speed of 2.51Mbits/sec appears in the test with the delay set to 600ms.

6.2.3 Test with dynamic path control

In the previous chapter, we introduced a dynamic path control module to deal with the change of the network status. When the network status changes, for example the end of an other flow releases free bandwidth, there is a high chance for us to find a more suitable path set to transfer our remaining file. In this section, we test our engine with a variable network situation to show the advantage of our dynamic path control module.

Test scenario

Figure 6.14 shows the test scenario for our dynamic control module. We have 6 switches in our network and in total they can provide 4 paths from



(a) delay = 100 (b) delay = 200

(c) delay = 300 (d) delay = 400

(e) delay = 500 (f) delay = 600

Figure 6.13: Transfer speed over different path delay

H1 to H2. A file with 1GB is transferred through the network from H1 to H2. Initially, only the path S1-S2-S3-S6 and path S1-S3-S6 are usable. After 50 seconds, we start switch S4 to provide a new path S1-S4-S6. After another 50 seconds, we start switch S5 to provide all usable paths. We test our file transfer engine with and without dynamic path control.

Test result

Figure 6.15a shows the throughput when the dynamic path control module is enabled. In this test, we set the detecting interval Int to be 2 seconds. Ini-

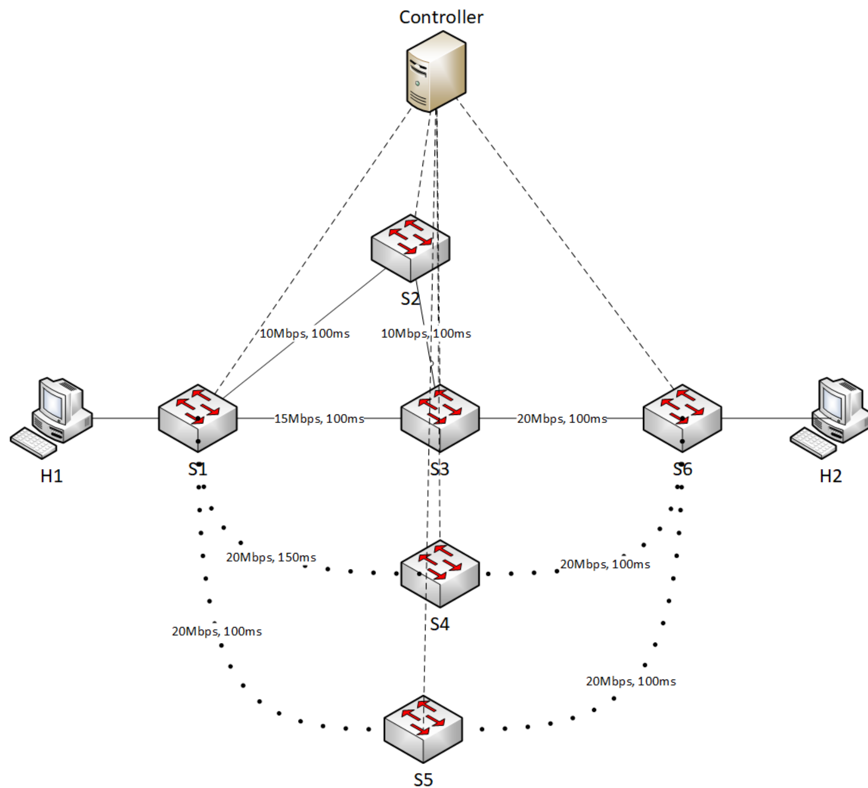
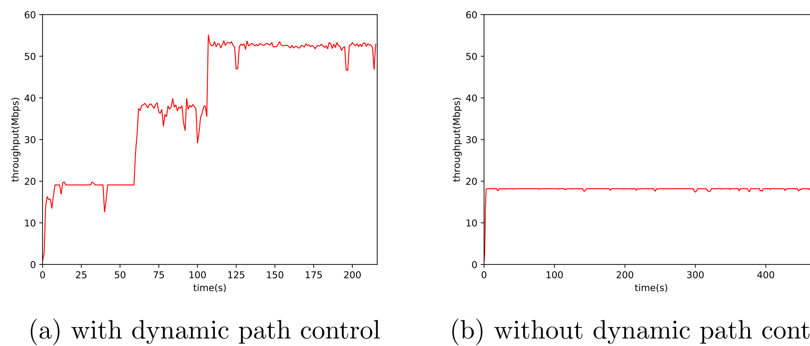


Figure 6.14: Scenario for dynamic path control experiment



(a) with dynamic path control (b) without dynamic path control

Figure 6.15: Throughput with and without dynamic control. Additional paths are available after 50 and 100 seconds.

tially, our algorithm can find two paths and the data is sent with a throughput of about 20Mbps. After the new path S1-S4-S6 becomes available at 50 seconds, our engine successfully detects the change in the network. And at the time of 58 seconds, we start to take advantage of the new path and

send the file data at a throughput of about 40Mbps. A similar situation happens when path S1-S5-S6 becomes available. At time 107 seconds, our engine starts to use the new path available at time 100 seconds to send the file at a speed of more than 50Mbps. In total, we use only 219 seconds to finish the transfer while in the test without dynamic path control, as shown in Figure 6.15b, it can not be aware of the additional new paths at time 50 and 100 seconds, and the whole transfer lasts 475 seconds.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

The main contribution of this thesis are methods to provide multipath file transfer in SDN networks. SDN is a centralized network architecture, which allows network managers to write their own applications on it. Different than traditional network architectures, SDN's centralized vision of the entire network can provide the real-time network status. Based on the network status, a file can be transferred with multipath when the resources are provided. The aim of this thesis is to model the problem that minimize the transfer time of a file with multipath and propose solutions for it.

First, delay and bandwidth are chosen to be the parameters, and with the introduce of time-expanded network, the problem is modeled using the linear programming formulation. A pseudo-polynomial algorithm BFMS-E is then proposed to give an exact solution to this problem. Based on this solution, a FPTAS algorithm BFMS-A is then proposed.

Second, an SDN-based proof-of-concept file transfer engine is implemented. The file transfer engine consists of two components. The monitoring component collects the real-time network status and then provides the information to the forwarding component. When a file needs to be transferred, the forwarding component uses the network status as the input to run the proposed algorithms and dispatches flow and group entries to the modified Open vSwitches. The modified Open vSwtiches can provide per-packet multipath forwarding using group table and remain OpenFlow compatible. A dynamic control mechanism is also added to the file transfer engine to update our transfer process when a new solution is found to be sufficiently better than our current solution due to the change of network status.

Finally, the proposed two algorithms are tested on simulated graphs and also with the file transfer engine in SDN networks. The simulated graph tests show that, when the problem size grows, algorithm BFMS-A achieves a much better time complexity over algorithm BFMS-E with only small

errors introduced under our test setting. Then with the file transfer engine, files can be transferred with multipath, dynamic control is also shown when network status changes.

7.2 Future Work

The proposed BFMS-A algorithm is now polynomially bounded by the upper and lower bound ratio $\frac{UB}{LB}$. This makes it possible that, in some situations, the algorithm still gives a bad time complexity. A more promising solution can be derived by finding a polynomial bound on $\frac{UB}{LB}$. The SDN test results show that congestion can have a huge influence on our transfer process, an end-to-end bandwidth guarantee is also necessary to add to our file transfer engine.

Bibliography

[1] Iris Bueno, Jose Ignacio Aznar, Eduard Escalona, Jordi Riera, and Joan García-Espín. An opennaas based sdn framework for dynamic qos control. In *Future Networks Serv. (SDN4FNS), 2013 IEEE SDN for*, volume 11, pages 1–7, 11 2013.

[2] C. Cajas, C. Valdivieso, D. Mejía, and I. Bernal. On programming an mp-tcp analyzer plugin using opendaylight beryllium as the sdn controller. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, Oct 2017.

[3] Cisco. Cisco visual networking index: Forecast and methodology, 2016–2021, September 2017.

[4] A. F. de la Cruz, J. P. Muñoz-Gea, P. Manzanares-Lopez, and J. Malgosa-Sanahuja. Network failures support for traffic monitoring mechanisms in software-defined networks. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 691–694, April 2016.

[5] M. Doshi and A. Kamdar. Multi-constraint qos disjoint multipath routing in sdn. In *2018 Moscow Workshop on Electronic and Networking Technologies (MWENT)*, pages 1–5, March 2018.

[6] J. Duan, Z. Wang, and C. Wu. Responsive multipath tcp in sdn-based datacenters. In *2015 IEEE International Conference on Communications (ICC)*, pages 5296–5301, June 2015.

[7] P. Erdös and A. Rényi. On random graphs, I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[8] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Rand Corporation research study. University Press, 1962.

[9] Y. Guo, Z. Wang, X. Yin, X. Shi, and J. Wu. Traffic engineering in sdn/ospf hybrid network. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 563–568, Oct 2014.

[10] M. Handley, A. Ford, C. Raiciu, and O. Bonaventure. Tcp extensions for multipath operation with multiple addresses. RFC 6824, RFC Editor, January 2013.

[11] C. He, K. L. Yeung, and S. Jamin. Packet-based load-balancing in fat-tree based data center networks. In *2014 IEEE International Conference on Communications (ICC)*, pages 4011–4016, June 2014.

[12] L. He. Online survivability in software defined elastic optical networks. Msc thesis, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands, Jan 2018.

[13] F. Hu, Q. Hao, and K. Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys Tutorials*, 16(4):2181–2206, Fourthquarter 2014.

[14] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, Aug. 2013.

[15] H. Krishna, N. L. M. van Adrichem, and F. A. Kuipers. Providing bandwidth guarantees with openflow. In *2016 Symposium on Communications and Vehicular Technologies (SCVT)*, pages 1–6, Nov 2016.

[16] C. Lin and F. A. Kuipers. Time-constrained data transfer scheduling. *Technical report for SURFnet*, Dec 2016.

[17] J. Lin, R. Ravichandiran, H. Bannazadeh, and A. Leon-Garcia. Monitoring and measurement in software-defined infrastructure. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 742–745, May 2015.

[18] H. Long, Y. Shen, M. Guo, and F. Tang. Laberio: Dynamic load-balanced routing in openflow-enabled networks. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 290–297, March 2013.

[19] Dean H. Lorenz and Danny Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Operations Research Letters*, 28(5):213 – 219, 2001.

[20] R. Mijumbi, J. Serrat, J. Rubio-Loyola, N. Bouten, F. D. Turck, and S. Latré. Dynamic resource management in sdn-based virtualized networks. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 412–417, Nov 2014.

[21] H. Nam, D. Calin, and H. Schulzrinne. Towards dynamic mptcp path control using sdn. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 286–294, June 2016.

[22] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.5*, March 2015.

[23] J. Pang, G. Xu, and X. Fu. Sdn-based data center networking with collaboration of multipath tcp and segment routing. *IEEE Access*, 5:9764–9773, 2017.

[24] S. T. V. Pasca, S. S. P. Kodali, and K. Kataoka. Amps: Application aware multipath flow routing using machine learning in sdn. In *2017 Twenty-third National Conference on Communications (NCC)*, pages 1–6, March 2017.

[25] A. Xifra Porxas, S. C. Lin, and M. Luo. Qos-aware virtualization-enabled routing in software-defined networks. In *2015 IEEE International Conference on Communications (ICC)*, pages 5771–5776, June 2015.

[26] Ryu project team. Ryu sdn framework. `https://osrg.github.io/ryu/`. Accessed August 12, 2017.

[27] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8, May 2014.

[28] Ronald van der Pol, Michael Bredel, Artur Barczyk, B. Overeinder, Niels L. M. van Adrichem, and Fernando Kuipers. Experiences with mptcp in an intercontinental openflow network. In *Proceedings of the 29th TERENA Network Conference (TNC2013)*, 2013.

[29] Open vSwitch community. Open vswitch 2.7. `https://github.com/openvswitch/ovs/tree/branch-2.7`. Accessed August 12, 2017.

[30] Q. Wang, J. Xue, G. Shou, Y. Liu, Y. Hu, and Z. Guo. Implementation of multipath network virtualization scheme with sdn and nfv. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6, Oct 2017.

[31] N. Ya, X. Wang, and M. Huang. Multipath load-balancing routing mechanism in data center network. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 167–172, Dec 2017.

Appendix A

Open vSwitch 2.7.0 Group Table Modification

As we specified in our thesis, to support packet level multipath transfer on OVS, we need to change the action buckets selection method from its original hash based method to our random selection. In this appendix, we will show where and how the source code need to be modified. All the modifications are based on OVS 2.7.0 [29].

A.1 ovs/ofproto/ofproto-dpif-xlate.c

```
1  #include <stdlib.h>
2  #include <time.h>
3
4  static bool is_initialized = false;
5
6  static struct ofputil_bucket *
7  group_best_live_bucket(const struct xlate_ctx *ctx,
8                         const struct group_dpif *group,
9                         uint32_t basis)
10 {
11     if (!is_initialized)
12     {
13         srand((unsigned int) time(NULL));
14         is_initialized = true;
15     }
16
17     struct ofputil_bucket *bucket;
18     uint16_t total_weight = 0;
19     LIST_FOR_EACH (bucket, list_node, &group->up.buckets)
20     {
21         if (bucket_is_alive(ctx, bucket, 0))
22         {
23             total_weight += bucket->weight;
24         }
```

```
25      }
26
27      uint16_t rand_num = rand() % total_weight + 1;
28
29      struct ofputil_bucket *best_bucket = NULL;
30
31      // struct ofputil_bucket *bucket;
32      uint16_t summed_weight = 0;
33      LIST_FOR_EACH (bucket, list_node, &group->up.buckets)
34      {
35          if (bucket_is_alive(ctx, bucket, 0))
36          {
37              summed_weight += bucket->weight;
38              if (rand_num <= summed_weight)
39              {
40                  return bucket;
41              }
42          }
43      }
44
45      return best_bucket;
46  }
47
48  static void
49  xlate_default_select_group(struct xlate_ctx *ctx, struct
        group_dpif *group)
50  {
51      struct flow_wildcards *wc = ctx->wc;
52      struct ofputil_bucket *bucket;
53      uint32_t basis;
54
55      ctx->xout->slow |= SLOW_CONTROLLER;
56
57      basis = flow_hash_symmetric_l4(&ctx->xin->flow, 0);
58      flow_mask_hash_fields(&ctx->xin->flow, wc,
        NX_HASH_FIELDS_SYMMETRIC_L4);
59      bucket = group_best_live_bucket(ctx, group, basis);
60      if (bucket)
61      {
62          xlate_group_bucket(ctx, bucket);
63          xlate_group_stats(ctx, group, bucket);
64      } else if (ctx->xin->xcache)
65      {
66          ofproto_group_unref(&group->up);
67      }
68  }
69
70  static void
71  xlate_select_group(struct xlate_ctx *ctx, struct group_dpif *
        group)
72  {
73      const char *selection_method = group->up.props.
        selection_method;
74
```

```
75      /* Select groups may access flow keys beyond L2 in order to
76       * select a bucket. Recirculate as appropriate to make this
        possible.
77       */
78      if (ctx->was_mpls)
79      {
80          ctx_trigger_freeze(ctx);
81      }
82
83      ctx->xout->slow |= SLOW_CONTROLLER;
84      xlate_commit_actions(ctx);
85      xlate_default_select_group(ctx, group);
86
87  }
```

Listing A.1: Part of ovs/ofproto/ofproto-dpif-xlate.c