THE UNIVERSITY OF QUEENSLAND

Optimal Routing in Software-Defined Networks

Kimberley Manning

School of Information Technology and Electrical Engineering,
The University of Queensland

Submitted for the degree of
Bachelor of Engineering
in the division of
Mechatronic Engineering

November 2014

November 10, 2014

Kimberley Manning

247 Fig Tree Pocket Rd

Fig Tree Pocket, QLD 4069

Prof. Paul Strooper

Head of School

School of Information Technology and Electrical Engineering

The University of Queensland

St Lucia, QLD 4072

Dear Professor Strooper,

In accordance with the requirements of the degree of Bachelor of Engineering in the School of Information Technology and Electrical Engineering, I submit the following thesis entitled:

"Optimal Routing in Software-Defined Networks".

This thesis was performed under the supervision of Assoc. Prof. Marius Portmann. I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Kimberley Manning

To my parents
for all twelve million

Acknowledgements

I would like to thank my supervisor A/Prof Marius Portmann for his consistent encouragement and support of this thesis, as well as the flexibility he gave me in setting a direction for this work which matched my interests. I also thank my work colleagues for their instructive comments about the practical applications of this field, and my friends for providing welcome distraction at (mostly) the right times.

Abstract

Software-defined networking (SDN) is a relatively new concept in networking which aims to give network operators greater programmability and control over their networks, through separation of the control and data planes. A centralised controller sends network control packets to other switches on the network, usually along a separate, parallel control network. With this new, centralised view of the network, it is possible to apply well-understood mathematical techniques from optimisation and operations research, possibly improving routing efficiency.

This thesis presents a framework to facilitate comparison of different routing metrics. The framework includes a controller, consisting of modules for topology discovery, flow statistics and routing control, and an experiment module to run network experiments on virtual networks, including several prewritten topologies. Three routing metrics (shortest path, widest path and residual capacity) are implemented.

Additionally, this thesis presents the results of network experiments performed using this framework, comparing the performance of the three metrics above on two topologies with different flow patterns. The residual capacity metric, which attempts to find a globally-optimal solution, improved on the performance of shortest path by up to 57% in some trials, with widest path only improving shortest path by 34%. However, residual capacity performed significantly worse when no solution could be found, such as for experiments with many flows. There, widest path improved on shortest path by up to 91%, while residual capacity was worse by 36%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software-defined networking (SDN) is a set of philosophies and concepts on how networks should be designed. The core idea is separation of the control and data planes; specifically, separation in a way that is standards-compliant and vendor-independent. Although the movement is not intrinsically tied to any specific technologies, the OpenFlow protocol is quite strongly linked to it. In a software-defined network, a central controller makes global decisions on routing and network policy (the 'control' plane) and sends OpenFlow control messages to instruct switches to update their individual forwarding tables where necessary (the 'data' plane).

OpenFlow is fundamentally a low-level network protocol. There now exist a number of projects which are designed to act as extensible SDN controllers, so that a programmer can simply use provided software APIs to construct and interpret OpenFlow messages instead of directly sending network packets. However, this still requires understanding of OpenFlow and network programming and functions as a convenience for programmers rather than a simple interface for end users. Most descriptions of SDN architectures include a layer above the control layer, often called the 'application' layer, for this reason. These systems are generally seen as a natural progression of the layer model as applied to network design.

This thesis presents `mcfpox`, a framework designed to ease research into optimisation techniques as applied to centrally-controlled, software-defined networks. The goal of the framework is to allow a researcher to write a high-level implementation of a particular routing metric, given information about the network topology and flows and their corresponding demands. The controller then uses this metric to allocate routes to flows in the network, and the performance of the metric (in terms of overall resulting throughput in the network) can then be evaluated, without directly interacting with the OpenFlow protocol.

In this way, optimisation techniques can be easily applied to select routes, in the hope that such techniques will result in greater network efficiency. Three routing metrics are evaluated, representing a transition from naive to globally-optimum routing: shortest path, where the path with the fewest hops is always selected; widest path, which selects the path with the highest remaining capacity after considering the placement of existing flows; and residual capacity, where the combination of paths which maximises the minimum residual capacity over all links is selected, using linear programming. Intelligent routing metrics such as the latter require the global view of the network that SDN provides, and as such are difficult to implement on traditional networks. This thesis investigates whether such metrics result in a significant improvement over traditional, shortest-path-based routing methods.

`mcfpox` is intended as the beginning of a larger project developed by Marius Portmann's SDN group for network research. This thesis focuses on the routing aspects of the system; support modules for statistics gathering and topology discovery were implemented to the extent required to make the system functional, but better implementations of these are the focus of other members of the SDN group. Eventually, the system will be used for research into routing in software-defined wireless mesh networks, which will produce extra challenges in gathering required information, but the modular structure of the controller means that replacing the support modules with updated versions should be straightforward.

1.1 Aims

There are two aims of this thesis:

1. To create a framework to allow easier research, testing and comparison of routing metrics for multicommodity routing.
2. To perform some basic analysis using the framework, both to demonstrate its use and to test the ease of use of the API.

1.2 Scope

Three routing metrics will be evaluated. Evaluation will be performed on virtual networks using the network emulator Mininet only, with overall resulting throughput compared on two different topologies. Flows will only consist of TCP over IP traffic generated using the `iperf3` tool.

1.3 Deliverables

The following will be produced by the end of the thesis:

1. A working controller consisting of POX modules
2. Implementations of each routing metric in Python
3. Mininet topology scripts used for testing
4. A framework to launch experiment scripts
5. A quantitative comparison of the performance of each metric
6. Documentation on installation and use of the framework

Chapter 2

Background

2.1 Software-Defined Networking

Software-defined networking, as described earlier, is a set of philosophies and concepts about network design. Crucially, networking should be programmable, with separation of the control and data planes allowing higher-level, central control without the need to directly manage switch forwarding tables, allowing innovation to progress faster due to reduced vendor lock-in, and without the need for direct support from hardware companies.

McKeown describes the networking industry as vertically integrated and proprietary, with little innovation [19], comparing this to the state of computing prior to the adoption of high-level operating systems using standard instruction sets to communicate with the hardware. Similarly, the control plane in SDN can be seen as a network operating system, with OpenFlow as the instruction set for communicating with switches. This echoes the development towards greater abstraction evidenced by the move towards flow- over packet-based networking and the rise of quality-of-service considerations. Such requirements can be difficult to meet without some control over switch forwarding tables.

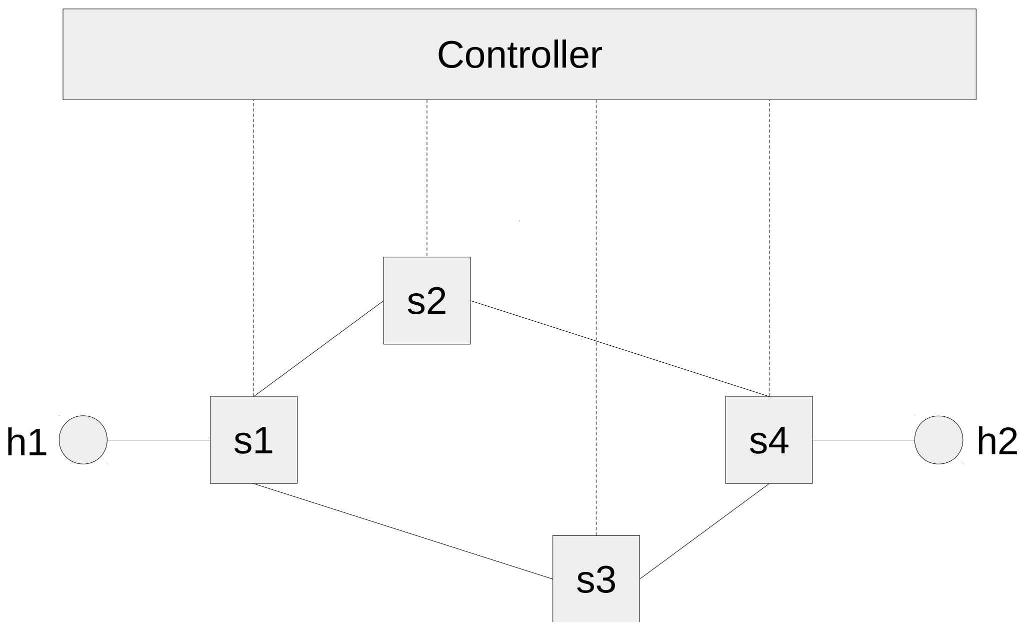


Figure 2.1: Centralisation of control in software defined networks
Each switch maintains an out-of-band connection to a controller, which instructs switches to update their forwarding tables via the OpenFlow protocol.

OpenFlow was originally conceived [20] as a research tool to enable academics to test new protocols easily and receive rapid feedback, while allowing vendors to continue to protect the inner workings of their switches. More recently [6] the focus has shifted to large datacenters and commercial networks with complex routing requirements. In [19], McKeown refers to the "ossified network": due to the black-box nature of modern network components such as switches and routers, experimentation is not encouraged and researchers and network administrators must stick to standard protocols until new ones are supported by vendors. This process can take a long time and the feedback cycle is slow. In contrast, SDN offers the ability to easily implement new protocols on real networks, and programmatically and automatically test and monitor many more aspects of the system.

Software-defined networks rely on two concepts: flow-based routing and the existence of a central controller. Flow-based routing means routing based on flows (linked series of packets such as all TCP traffic, all traffic from one MAC address or to a particular IP subnet). This concept was already reasonably well-developed [25, 23].

The other element of SDN is the existence of a central network OS which can communicate with all nodes. This raises the obvious objection that in many cases, for example in mesh networks, centralisation is not ideal; however, this is not centralised routing where all packets must pass through the controller. The authors of Aster*x [16] make the distinction between "logically centralised" and "distributed through the network": once rules are installed, packet forwarding happens at individual switches as usual. The controller can push an initial set of rules as soon as a switch connects to it if desired. Later, when the first packet of a new kind of flow arrives at a switch, it encapsulates the packet and sends it to the controller. The controller then installs the appropriate rules on the appropriate switches [19]. Depending on configuration (fail standalone mode or fail secure mode) these rules can remain in place if the controller connection is lost, or the switch can drop all packets [7].

2.1.1 Examples

The following SDN projects had a particular influence on some aspect of the design of this thesis and are therefore described in more detail here.

Dely et al [15] implemented an SDN controller and tested it on the KAUMesh testbed, using the metrics of forwarding performance, amount of control traffic and rule activation time. The performance was within acceptable limits for their small-scale test, but they noted that scalability could be an issue when deploying on larger networks. The scalability of routing algorithms considered in this thesis is therefore evaluated, in addition to the resulting throughput.

A project that had particular influence on the experimental design in this thesis was the Hedera project [9], aimed at large-scale enterprise applications. For part of the work, they compared the performance of two routing metrics, global-first-fit and simulated annealing. The paper describes a wide range of traffic flow patterns used to evaluate each metric, some of which where adapted for use in this thesis.

Additionally, two of the authors also presented a paper [8] on the fat-tree topology used for their experiments, which was also used in experiments for this thesis.

Several other projects used linear programming to solve a particular problem known as the multicommodity flow problem (MCFP). These attempts are outlined in the following section.

2.2 Multicommodity Flow Problems

SDN does not make any judgment on what metric should be used by the controller to allocate flows. Indeed, it is possible to use prewritten modules in controller frameworks such as POX [4] to emulate traditional routers or simple forwarding switches. However, SDN offers one particular advantage over traditional routing that is of interest here: the ability to easily maintain a global view of the network. The routing problem is then reduced to one which is well-known in graph theory: the multicommodity flow problem.

In the MCFP, flows are modelled as commodities moving between various source and sink nodes in the network [13, pp. 862–863]. Each flow has an associated demand, representing the bandwidth consumed by that flow. The approach is mentioned in [24] and [14]. This is an optimisation problem where the objective function can be adjusted depending on the desired outcomes, such as maximising overall throughput or maximising worst-case performance. The authors of [24] note that such approaches can be sensitive to accurate predictions of demand, but had success merging it with more dynamic heuristic algorithms.

A subcategory of the multicommodity flow problem which is more relevant to network routing is known as the unsplittable flow problem (UFP), and is also well-studied in the optimisation literature [10, 11, 12, 22]. The problem is motivated by the effective restriction when routing TCP traffic that all packets should travel along the same path. If multiple paths are taken, this increases the risk that packets will arrive out of order. TCP interprets this as congestion in the network and slows down its sending rate, thereby decreasing the throughput for that flow. The UFP adds the restriction that flows must be routed along exactly one path.

2.3 Linear Programming

Both the MCFP and the UFP are usually expressed as an objective to be maximised or minimised, and a series of constraints. The objective function represents the overall 'goal' of the system and can vary by formulation, as different applications warrant a different focus. However, the constraints reflect the limitations imposed by the underlying network and therefore do not significantly vary.

Problems which can be expressed in this way are known as linear programs and can be solved (an optimal solution found, within the bounds of the constraints) using general solvers such as the GNU Linear Programming Kit (GLPK) [1]. Most techniques used by such solvers use algorithms using linear algebra, such as the simplex and branch-and-cut methods; however, it is usually not necessary to know the algorithm being used, as interfaces to such solvers often only require a high-level description of objective and constraints and return the combination

Chapter 3

Routing Algorithms

One of the goals of this thesis is to evaluate the performance of a selection of routing metrics, and to assess the feasibility of using more complex algorithms in exchange for improved network efficiency.

Given the scope of this thesis, three metrics are evaluated, labelled shortest path, widest path and residual capacity. These metrics were carefully selected to represent a transition from 'naive' routing, where flows are placed regardless of link capacity or the presence of other flows, to routing that considers the entire network and finds a globally-optimal set of routes. The following sections provide the theoretical basis for these metrics, suggested implementations and illustrations of their use.

3.1 Shortest Path

Shortest-path routing is the basis for many existing distributed routing algorithms currently in use. The algorithm is very simple: any new flow is allocated the path with the fewest 'hops' (links) between source and destination. If there are multiple such paths, behaviour is dependent on implementation. Knowledge of other flows in the network is not required.

Flow	Source	Destination	Demand
● 1	h1	h2	1 Mbps
● 2	h2	h1	2 Mbps

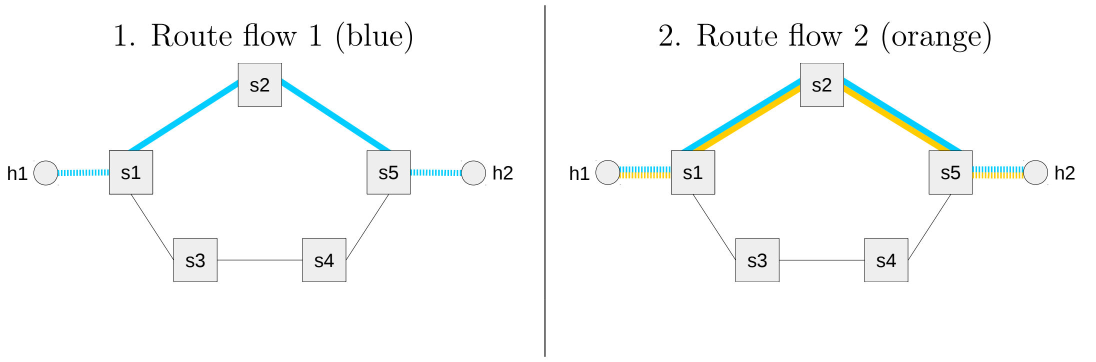


Figure 3.1: Shortest path route allocation
The path with the fewest hops is always chosen for each flow.

An efficient method for calculating the shortest path from a source node to any other node is given by Dijkstra's algorithm [13, pp. 684–693]. The algorithm is well-known enough that many existing implementations are available, including ones with optimisations and run-time improvements not present in the original algorithm. In `mcf-pox`, the Python library NetworkX [3] is used to represent the network graph as seen by the controller; this library provides a function `shortest_path(G,source,target)`, which returns the shortest path from `source` to `target` through the graph `G`.

3.2 Widest Path

In widest path routing, each flow is allocated to the path with the highest remaining capacity. Flows are routed in order from largest to smallest. The widest path must be claculated for each flow individually, then the network updated to reflect the reduced capacity along that path when routing the next flow. The widest path metric therefore represents an intermediate step between naive and globally optimal routing. A simple example is show in Figure 3.2.

Flow	Source	Destination	Demand
● 1	h1	h2	1 Mbps
● 2	h2	h1	2 Mbps

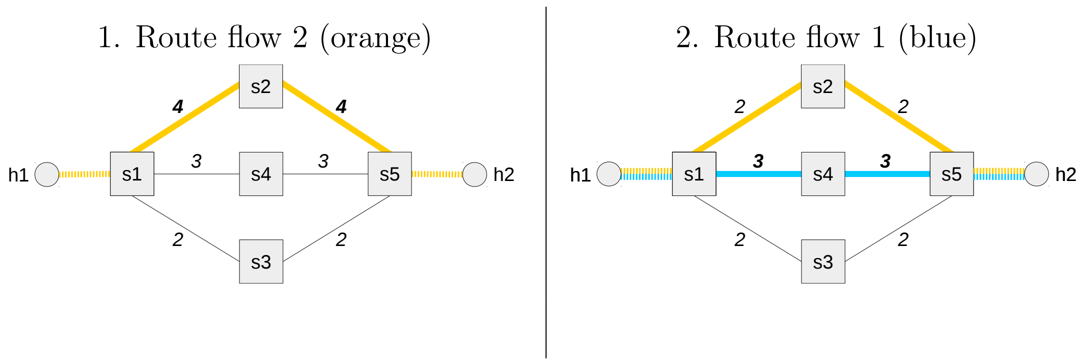


Figure 3.2: Widest path route allocation
Numbers show remaining capacity (Mbps) per link. Each flow is allocated to the path with the highest remaining capacity, starting from the largest flow.

It is possible to calculate the widest path in a network for a single source/destination pair by modifying Dijkstra's algorithm to treat edge weights as capacities rather than costs to be minimised. The particular formulation used in this thesis is based on one by Medhi [21]. The algorithm to calculate the widest path from node i to any other node in the network is reproduced in Figure 3.3.

3.3 Residual Capacity

The final metric considered is an attempt at load-balancing flows across the entire network. The 'best' combination of routes is the one which maximises the minimum residual capacity over all links (the capacity remaining after the total size of all flows passing through is subtracted). A global solution will not be calculated for the arrival of every flow; rather, every few seconds all routes would be recalculated to find a globally-efficient solution, and routes arriving in between would be allocated an interim path. If flows are evenly allocated, this reduces the likelihood that the

```
1. Discover list of nodes in the network, $N$, and available bandwidth of link
   $k - m$, $b^i_{km}(t)$, as known to node $i$ at the time of computation, $t$.
2. Initially, consider only source node $i$ in the set of nodes considered,
   i.e., $S = \{i\}$; mark the set with all the rest of the nodes as $S'$.
   Initialise $B_{ij}(t) = b^i_{ij}(t)$.
3. Identify a neighbouring node (intermediary) $k$ not in the current list $S$
   with the maximum bandwidth from node $i$, i.e., find $k \in S'$ such that
   $B_{ik}(t) = \max_{m \in S'} B_{im}(t)$.
4. Add $k$ to the list $S$, i.e., $S = S \cup \{k\}$.
5. Drop $k$ from $S'$, i.e., $S' = S' \backslash \{k\}$. If $S'$ is empty, stop.
6. Consider nodes in $S'$ to update maximum bandwidth path,
   i.e., for $j \in S'$, $B_{ij}(t) = \max\{B_{ij}(t),, \min\{B_{ik}, b^i_{kj}(t)\}\}$
7. Go to Step 3.
```

Figure 3.3: Widest path algorithm, computed at node i
Adapted from the formulation given in [21].

interim path will be congested while other paths are unused. This is true regardless of the metric used to calculate the interim path, but a metric such as widest path would likely lead to increased performance as it would be able to take advantage of differences in the available capacities.

Figure 3.4 outlines a manual solution to a small version of this problem. In practice, explicitly enumerating and evaluating all paths is expensive; instead, it is possible to use linear programming, as described in section 2.2, to find the optimal solution to such problems. This is example of a multicommodity flow problem, specifically, an unsplittable flow problem. The following is a formulation of the unsplittable flow problem as adapted from Walkowiak [22].

Begin with a network G with vertices V and edges E, with edge capacities $c : E \rightarrow \mathbb{R}^+$. The set P comprises p commodities to be routed through G, where the ith commodity corresponds to a flow with a source $s_i \in V$, destination $t_i \in V$ and demand $d_i \in \mathbb{R}^+$. For each $i \in P$, there are l_i possible routes between s_i and t_i, so define the set $\Pi_i = \{\pi^k_i : k = 0, ..., l_i\}$ to represent these.

In the unsplittable flow problem, each commodity must be routed entirely along one path. Each potential path π_i^k is therefore associated with a corresponding variable $x_i^k \in \{0,1\}$, indicating whether that path was chosen for commodity i. The total traffic across edge $j \in E$ must be less than the capacity of the link. The constant $a_{ij}^k \in \{0,1\}$ indicates whether path π_i^k uses edge $j \in E$.

The problem can be expressed as a linear program as follows.

$$\max z \text{ s.t.}$$

$$\sum_{\pi_i^k \in \Pi_i} x_i^k = 1 \qquad \forall i \in P \tag{3.1}$$

$$x_i^k \in 0,1 \qquad \forall i \in P; \pi_i^k \in \Pi_i \tag{3.2}$$

$$f_j = \sum_{i \in P} \sum_{\pi_i^k \in \Pi_i} a_{ij}^k x_i^k d_i \qquad \forall j \in E \tag{3.3}$$

$$f_j \leq c_j \qquad \forall j \in E \tag{3.4}$$

$$z \leq c_j - f_j \qquad \forall j \in E \tag{3.5}$$

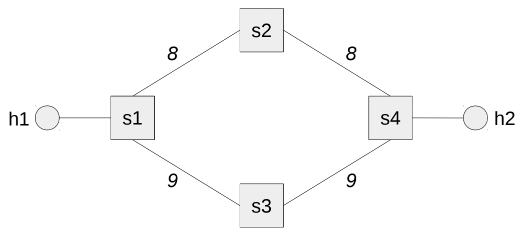
In short: the goal is to maximise z. Constraint (3.5) defines z so that for every edge (or link), the spare capacity on that link is at least z. The total traffic routed through each link, defined in (3.3), must be less than its capacity, by (3.4). Each possible route for a given commodity is either selected or not, and only one route per commodity is selected, by (3.2) and (3.1) respectively.

Flow	Source	Destination	Demand
● 1	h1	h2	2 Mbps
● 2	h1	h2	3 Mbps
● 3	h2	h1	6 Mbps

1. Consider possible routes

Path			Residual Capacity		
● 1	● 2	● 3	Upper	Lower	Min
U	U	U	-3	9	-
U	U	L	3	3	3
U	L	U	0	6	0
U	L	L	0	6	0
L	U	U	-1	7	-
L	U	L	1	5	1
L	L	U	2	4	2
L	L	L	-2	8	-

Original network



2. Assign routes simultaneously

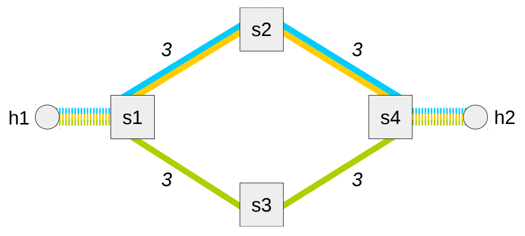


Figure 3.4: Residual capacity route allocation
Numbers show remaining capacity (Mbps) per link. The combination of paths which maximises the minimum residual capacity over all links is selected.

Chapter 4

Controller Design

4.1 Architecture

A major deliverable for this project is the production of a configurable SDN controller, to allow testing of different routing metrics or objective functions. A high-level overview of the system architecture is given in Figure 4.1. The controller framework consists of three modules, described individually in detail later: `topology`, `statistics` and `multicommodity`. While the objective function is configurable, it can be considered part of the controller as well.

4.1.1 Communication

Immediately before running the objective function, the multicommodity module proactively requests the network graph and list of current flows. The topology module uses adjacency lists from the `discovery` and `host_tracker` modules to return an updated graph. Finally, the multicommodity module calls the provided objective function with the graph and flows as arguments, and receives a list of paths for each flow back. This list is used to install forwarding rules on the switches.

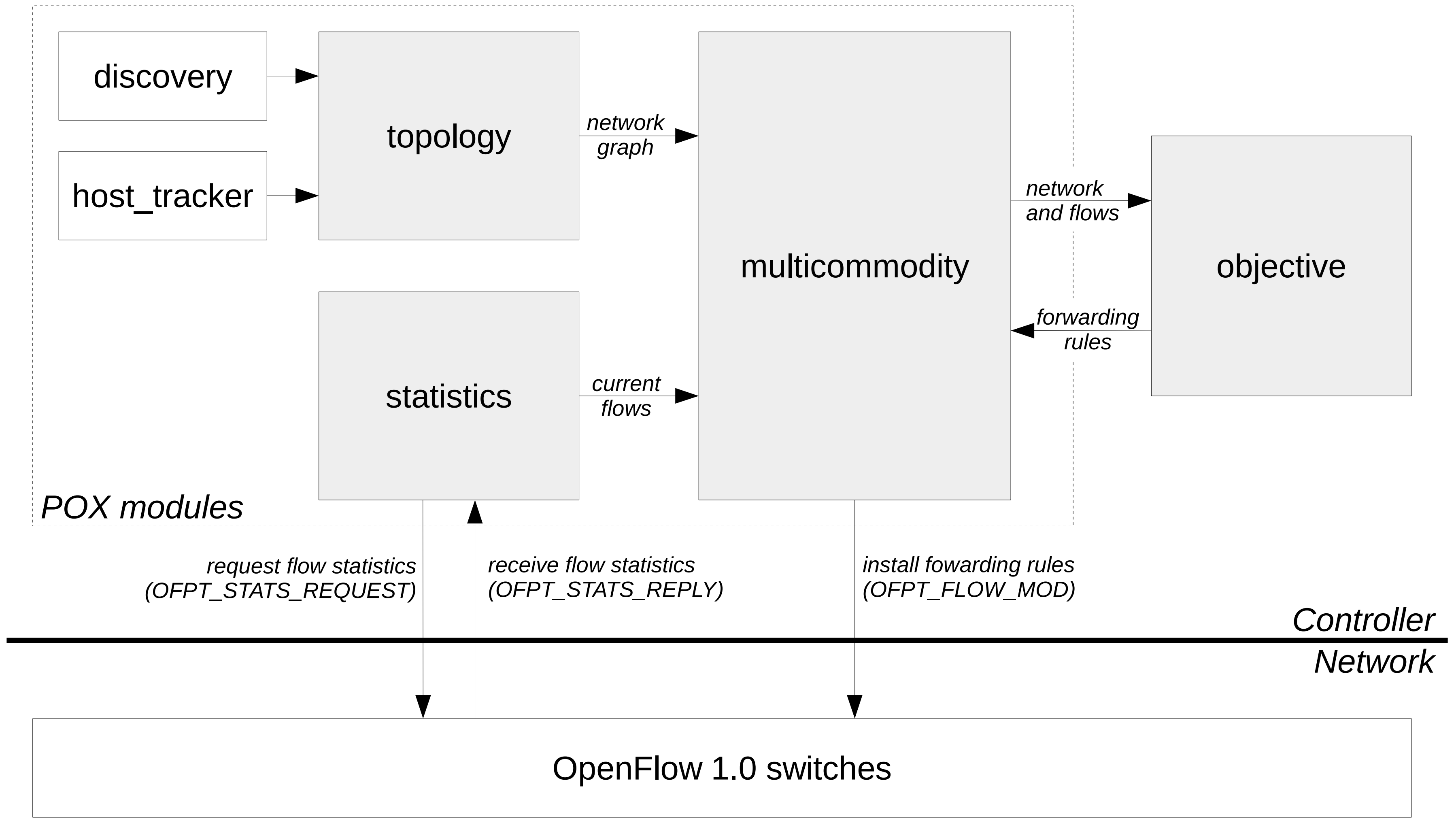


Figure 4.1: Overview of controller architecture

The controller communicates with switches in the network via the OpenFlow protocol. The statistics module exchanges `OFPT_STATS_REQUEST/REPLY` modules periodically to update the list of flows in the network, including their size over the most recent period. After paths are calculated, the multicommodity module sends `OFPT_FLOW_MOD` messages to instruct the appropriate switches to update their forwarding tables.

Additionally, while the topology module does not send or receive OpenFlow messages directly, it makes use of information sourced from the `discovery` and `host_tracker` modules in POX; the former uses `OFPT_PACKET_OUT` messages to send and track LLDP messages for switch discovery, and the latter inspects `OFPT_PACKET_IN` messages for ARP messages indicating the location of hosts. These messages are not shown in Figure 4.1 as they are not controlled by `mcfpox` modules.

4.2 Dependencies

4.2.1 POX

There are a number of controller frameworks today which allow programmers to send OpenFlow messages to switches in the network; selection in the first case is based on familiarity with the development language used. One such framework, based on Python, is POX [4], developed at Stanford primarily for ease of research over speed and performance. POX only supports OpenFlow 1.0 [5].

Functionality in POX is implemented as reuseable modules, which can be combined with other modules to meet particular requirements. POX ships with a number of prewritten modules for basic switch forwarding, network discovery and so on; of particular interest are the `openflow.discovery` and `host_tracker.host_tracker` modules, which are used by the `mcfpox` topology module for network discovery.

4.2.2 NetworkX

NetworkX is a Python package for the 'creation, manipulation, and study of the structure, dynamics, and functions of complex networks'. It is a stable and well-maintained graph library with a flexible, easy-to-read annotation system for nodes and edges, which is used to store information such as IP addresses and link capacities. `mcfpox` uses NetworkX to store a representation of the network as discovered by the controller. The topology module builds a NetworkX graph, which is passed to the objective function; it can then make use of provided mathematical functions, such as the `shortest_path` and `all_simple_paths` functions.

4.2.3 GLPK/PuLP

Finally, GLPK is a general solver for optimisation problems, including mixed integer problems such as the residual capacity metric described in section 3.3. GLPK is written in C, for speed and efficiency reasons, and uses techniques

PuLP is a Python module which is used to describe optimisation problems programmatically. After an objective function and constraints are added, PuLP writes the description of the problem to a file, and then calls an external solver, such as GLPK, to solve the model and return the optimal solution. The interface is entirely through Python code, which fits well with the rest of the framework.

4.3 POX Modules

The major software contribution of this thesis lies in the series of POX modules that make up the `mcfpox` controller. The central base module launches the other three modules: one each for topology discovery, statistics gathering and routing control (described in detail below). Of these, the most relevant is the routing control module, which allows the routing metric under study to be passed to the controller. The other two modules are essential prerequisites for correct operation of the routing module, but not the major focus of this thesis.

In order to launch the controller programmatically in experiment scripts, an important modification was made to the startup sequence of POX. As of the most recent branch on the `github.com/noxrepo` repository, `dart`, POX is only designed to be started from the command line. Though the command line interface provides a format for passing arguments to individual modules, arguments must be serialised to text, and are limited in length to the maximum length of arguments. However, arguments to `mcfpox` modules can be quite complex; for example, the multicommodity module accepts an objective function and a dictionary of flow rules to install.

In initial development of `mcfpox` this was achieved by specifying the name of a Python module to import containing this data, but this imposes restrictions on the layout of the module; for example, the objective function must have a predefined name to make the import process easier. The POX initialisation code was therefore modified to allow starting the controller directly from a Python script, which allows the passing of arbitrary Python objects such as functions and dictionaries directly as arguments, without the need for serialisation to the command line. Correspondence with the principal maintainer of POX, Murphy McCauley, indicated that he is interested in possibly incorporating this feature into a later release; until then the `mcfpox` project officially relies on the `github.com/krman` fork of POX.

4.3.1 Network Discovery

Network discovery of switches, hosts, and links is done using the existing POX modules `discovery` and `host_tracker`. The topology module uses these two modules to build a NetworkX graph of the underlying network, which is passed to the objective function for routing calculations.

The `discovery` module uses LLDP packets to discover switches and links between them, which are specified with the same dpid that switches use to identify themselves to the controller. The module builds up an adjacency list of pairs of switches which are connected, including the ports by which the switches are connected. The `host_tracker` module tracks ARP requests and replies and uses them to locate hosts in the network, but stores this information in a significantly different format to `discovery`; this makes it difficult to use both modules together. The main purpose of the topology module is therefore to bridge these two modules, by maintaining a NetworkX graph containing information about both hosts and switches. NetworkX allows annotations on both nodes and edges in the graph; these are used to store information on IP addresses, switch ports and link capacities.

The information provided by `discovery` and `host_tracker` is limited: both only specify that a link exists, but not important properties of the link such as its capacity. In the current implementation, all links between switches are recorded as having a 10 Mbps capacity, regardless of actual capacity; to make this work, all Mininet topologies are created with 10 Mbps links. This limits the types of topologies that can be tested. While this is obviously undesirable, capacity discovery is a significant problem of its own. Another student in Marius Portmann's SDN group is currently researching using packet-pair probing and other techniques to dynamically discover link capacities; as such this is outside the scope of this thesis.

4.3.2 Flow Statistics

OpenFlow provides a number of message types to query switch statistics, such as number of packets or number of bytes seen for particular flows. The statistics module sends `OFPT_STATS_REQUEST` messages to each switch known by the controller periodically (the time between requests is configurable at startup). Each switch replies with a `OFPT_STATS_REPLY`, which lists the number of bytes which have been processed for each flow corresponding to an installed flow rule. Using this information, the statistics module calculates the number of bits per second seen, recently, per flow. Since the replies are per switch, occasionally the figures differ very slightly (for example, if a flow has only partially traversed the network); in such cases the statistics module records the greatest number seen on any switch.

In order to facilitate the measurement of statistics, flows are initially routed using the simple shortest-path metric. This allows the statistics module to record an estimate of the flow size, before full-network routing is calculated. While this works well when the network is not under load, as the number of flows in the network increases the statistics become less accurate over time. This is due to the difference between what the authors of a similar framework [9] refer to as the *natural demand* and the *measured demand*: if flows are initially routed badly, then congestion will occur. The TCP congestion control algorithm will slow down the sending rate to avoid excessive packet loss, which means that although switches are accurately reporting the bytes passing through, this number may not accurately reflect how much the sender would transmit given unlimited network capacity.

Continuing, the globally-optimal routing metrics will then underestimate the size of that flow, and perhaps route it inefficiently, instead of allocating it to a bigger link where it could transmit at a greater speed. This reduces the overall throughput in the network. The authors of the Hedera framework [9] implemented an algorithm to estimate the natural demand of a flow, but similarly to link capacity detection, this is outside the scope of this project.

4.3.3 Routing Control

The final but most important `mcfpox` module is multicommodity, the routing control module. Routing is controlled in two stages. When a new flow arrives, it is assigned an interim path and rules are installed on switches along this path. Periodically (how often is configurable, though for experiments described in this thesis it is only run once) the module runs an objective function, which calculates optimal routes for all current (recently-seen) flows, removes existing switch rules for these flows and installs the new ones. The interim path is simply the shortest path between source and destination nodes.

The objective function is configured at startup time. After a period of time, the multicommodity module takes a snapshot of the current view of the network from the topology module, and the most recent flow statistics from the statistics module, and passes these to the given objective function, which uses them to calculate the set of routes for each flow. For each calculated path, expressed as a list of hops, the multicommodity module installs a rule on each switch in the path that forwards packets corresponding to that flow out on the appropriate port.

As described later in Chapter 5, the iperf clients which form the flows in the network begin measuring throughput 10 seconds after being launched, concurrently with the launch of POX. The recalculation of forwarding rules is timed to coincide with this point, so that the iperf measurements are based on the new paths rather than the initial shortest-path routes. In the current implementation, the controller is stopped after only after one recalculation, as this is all that is required for the experiments, but in a normal situation this would occur repeatedly, as new flows entered and left the network.

One feature of the multicommodity module that was not used for the final experiments but was extensively used in development was the ability to install mock flow rules, in order to test the throughput in simple networks with known paths, to check that the experiment framework worked as expected. Flow rules could be calculated externally or manually written, then passed to the controller and directly installed. This can also be used as an alternative to shortest path for initial routing.

Chapter 5

Experimental Methods

The goal of this thesis is to implement a configurable controller which can be used in network experiments to compare the performance of different routing algorithms. For the purposes of this thesis, performance is measured exclusively in terms of overall throughput in the network. In order to measure throughput for different routing metrics, a series of network experiments are devised.

Chapter 4 outlines the layout of the controller used in the experiments. Chapter 3 outlines the three routing metrics of interest: the shortest-path metric, given its simplicity and the number of traditional routing metrics which are based on it, can be considered a benchmark by which the other two metrics can be compared. This chapter describes the structure and implementation of the experiments.

5.1 Dependencies

5.1.1 Mininet

Mininet is a network emulator which uses container-based emulation [18] to create a virtual network of hosts and switches. Mininet makes it simple to run reproducible network experiments under realistic conditions, with topologies and behaviour programmable in Python. The resources allocated to each host can be controlled and monitored, and benchmarking [17] has shown that the performance and timing characteristics are accurate. OpenFlow-enabled switches such as OpenVSwitch can be used in the simulations so the system is ideal for SDN research.

5.1.2 iperf3

`iperf3` [2] is a command line tool which can be used to perform network throughput measurements. Testing is performed by running a server on the destination host and a client on the source host, which will attempt to contact the server over TCP or UDP. Various options are available, including configuration of the TCP/UDP ports used, and the maximum bandwidth (size) of the flow. It is also possible to begin sending packets for a flow, but omit the first n seconds from throughput calculations. Results, including the final measured throughput for each client/server pair, can be reported in JSON format for ease of futher processing.

5.2 Experiment Design

All network experiments follow a defined pattern. Figure 5.1 shows the timing of each experiment trial. Firstly, POX and Mininet are started. There is a non-trivial setup time for Mininet topologies to be created, followed by a non-trivial time for POX's discovery module to discover the network. After waiting for this to occur (20 seconds is enough time for the largest topology considered for this thesis), a series of `iperf3` flows are started, limited to a specified bandwidth.

For the first 10 seconds, `iperf3` simply sends data at a maximum of this specified bandwidth, but ignores the number of bytes sent for this period. This is because the throughput for the flow will be lower than expected, for several reasons: the TCP

0s	20s	30s	40s
Start POX/Mininet Discover network topology	Start iperf flows Calculate initial demand	Recalculate routes Measure throughput	End

Figure 5.1: Timing of events in each trial

slow-start period, the time taken for ARP requests and replies to be exchanged, and for the controller to be notified of the hosts involved and set an initial path.

After 10 seconds, the controller's multicommodity module recalculates all routes in the network according to the objective function used, and replaces the initial flow rules with updated forwarding rules based on these routes. The `iperf3` clients continue to send data at the same rates for this period, but begin to record the number of bytes sent and received, and do so for 10 seconds. A few seconds after this, to allow time to write this data to log files, the experiment concludes and the network is shut down, including all `iperf3` servers and clients still running.

The topology, flow pattern and routing metric considered can all be configured via a simple launch script, such as the sample shown in Figure 5.2. The following sections explain how to firstly select these parameters so that useful information can be obtained, and secondly to write experiment scripts to run the desired experiment.

5.3 Parameter Selection

While it is possible to use mcfpox to run experiments using many combinations of topologies and flow patterns, not all of these combinations will yield useful data. For example, if all flows in a pattern are much larger or smaller than the link capacities in the topology, either all flows will be constrained or none will be. A carefully-selected combination, however, will yield a situation where flows will only be constrained if an inefficient routing metric is used, but not otherwise. In such a situation it is possible to measure the variation in performance for the two routing metrics.

Topology	Pattern	Description
pentagon	bidirectional(b) 2 flows	h1 - h2, b Mbps h2 - h1, b Mbps $b = 1, 2, .., 10$
alfares	pairs(b) 8 flows	8 random pairs from h1-h16 such that each host is part of exactly one b Mbps flow $b = 1, 2, .., 10$
	random(n,b) n flows	n random pairs from h1-h16 Hosts may appear in more than one pair or none but no pair is repeated b = 1,5,7

Table 5.1: Traffic patterns studied for each topology

As noted in section 4.3.1, due to constraints on the ability of the controller to dynamically discover link capacities, all implemented topologies have 10 Mbps links between switches. The experiments in this thesis therefore examine flows with individual sizes ranging from 1 - 9 Mbps. The exact flow patterns used depend on the topology and are specified in Table 5.1.

Diagrams of the two topologies used in the experiments can be seen in Figures 5.3 and 5.4; the second topology, which was used in most of the experiments, is based on the description of a fat-tree network given in [8]. This topology scales to much larger sizes based on a parameter k, the number of ports per switch; larger versions are used to compare the scalability of different routing metrics.

5.4 Experiment Scripts

One of the goals of this thesis was to create an easy-to-use experiment framework. `mcfpox`, therefore, includes an experiment module which abstracts most of the boilerplate involved in setting up a routing experiment, allowing the experimenter to run the entire experiment from a script such as the one shown in Figure 5.2.

```python
from mcfpox.topos import alfares
from mcfpox.objectives import shortest_path
from mcfpox.experiments.boilerplate import start

pairs = [('h1','h8'), ('h8','h1')]
flows = {
    19: [(i,j,5) for i,j in pairs]
}

scenario = {
    'net': alfares,
    'flows': flows
}

controller = {
    'objective': shortest_path.objective,
}

results = start(scenario, controller)
```

Figure 5.2: Sample experiment script

There is a conceptual distinction, which is referenced in the structure of the script, between 'scenario' and 'controller' configuration. A scenario comprises a topology and a set of flows to be started at specified times. Controller configuration is where options to be passed to the controller are stored. To compare the performance of two objective functions, the scenario is kept constant while only the controller objective is changed. When flows are allocated randomly, the same allocation of flows is used for each metric, and at least 10 different random allocations are considered and then averaged, to minimise the effects of particularly difficult flow patterns.

5.5 Emulation Hardware

Mininet's scalability is dependent on the specfications of the host machine running the virtual network. The network experiments in this thesis were run on a quad-core i5-4670K (3.4 GHz) desktop with 4 GB RAM.

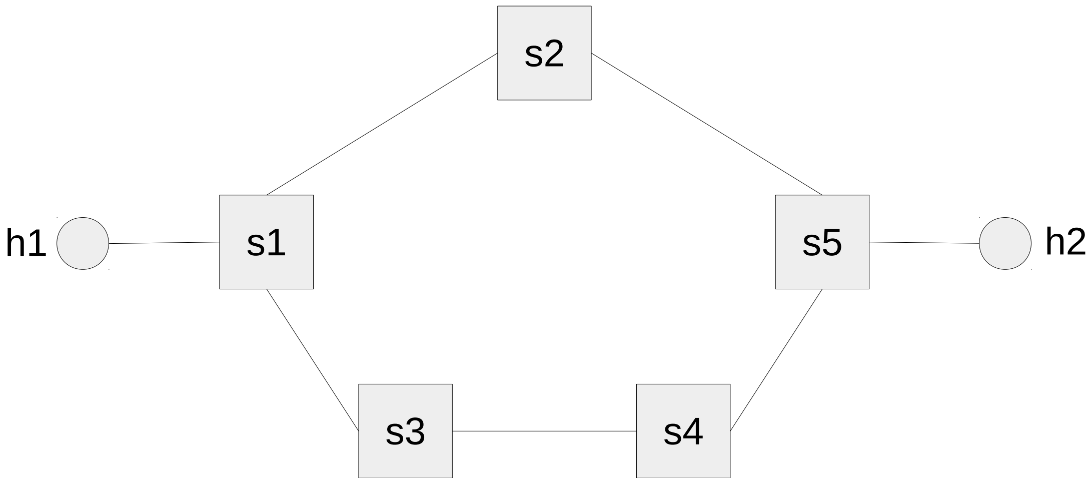


Figure 5.3: Pentagon topology ('pentagon')

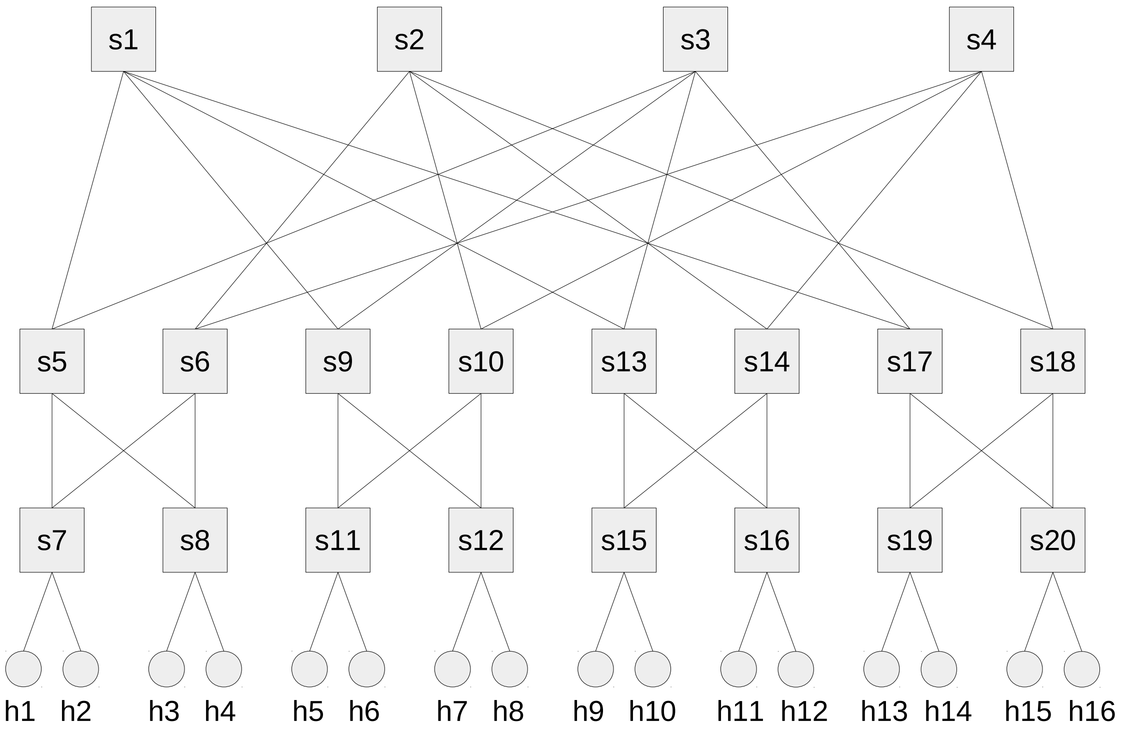


Figure 5.4: Al-Fares fat-tree topology ('alfares')
This topology belongs to a class of related networks described in [8], parametrised by k, the number of ports per switch. Here, $k = 4$. Larger versions of this topology are used for scalability testing of routing metrics.

Chapter 6

Results

This chapter presents the results of the analysis performed using the framework.It is divided into two parts: the first presents a comparison of the performance of the three routing metrics, in terms of overall resulting throughput in the network; the second discusses how well each metric would scale to larger topologies than it is possible to use in these experiments.

6.1 Comparison of Routing Metrics

6.1.1 Experiment 1

The pentagon topology was primarily selected as a 'sanity check' for the experiment framework. The topology is simple enough that predictions about the routes generated by each of the routing metrics can be made: shortest path will always route flows along the upper path, while the widest path and residual capacity metrics will attempt to load-balance by splitting flows along both paths. It is therefore expected that throughput in the shortest path experiments will reach a maximum earlier than the other two metrics; specifically, the latter two will continue to match an offered load up to twice that of shortest path before reaching their limit.

This can be seen in Figure 6.1, which compares the aggregate offered load against the aggregate throughput, as measured by `iperf3` clients. The behaviour is mostly as expected. Widest path and residual capacity show very similar behaviour, as expected since both metrics would allocate the same routes. Given that the pentagon experiment was intended as a test of the experiment framework, however, the behaviour of shortest path is concerning. While it does appear to reach maximum throughput at about the right time, the measured throughput hovers around 11 Mbps, when theoretically it should be limited to 10 Mbps.

It is possible that `iperf3` is sending more data than requested, but if so Mininet should still restrict the bandwidth seen by the client and server. Examination of the client/server logs shows that `iperf3` is faithfully calculating the total from each time period, but that throughput is consistently seen as higher than the expected 10 Mbps. Further investigation is required; for other experiments, it must be assumed that `iperf3` continues to over-report throughput, but as this does not appear to be dependent on the routing metric, and the difference is minor, conclusions on the relative performance of routing metrics may still be drawn.

6.1.2 Experiment 2a

It is hard to distinguish efficient routing metrics on small topologies such as the pentagon. The final two experiments, therefore, use the fat-tree network from [8] with $k = 4$, referred to here as the 'alfares' topology. Figure 6.2 shows the results of an experiment in which the 16 hosts were grouped randomly into 8 pairs for each trial, creating equal-sized flows with bandwidth $b = 1, 2, ..., 10$ Mbps. Since the pair allocation was random, the results of 10 trials (with different pair allocations) were averaged to gain the mean aggregate throughput for the network. For fairness, each pair allocation was used for every routing metric.

The improved performance of the widest path and residual capacity metrics can be seen in Figure 6.2. Unlike the pentagon topology, there are enough different possible paths to be able to distinguish the performance of widest path from residual capacity. It was speculated that since the latter metric attempts to find a globally-optimal solution rather than allocating routes sequentially, this would result in greater network efficiency and therefore higher aggregate throughput. The graph does appear to show that this is the case. The size of individual flows is limited by the 10 Mbps capacity of the links, so the largest offered load is 72 Mbps, corresponding to eight 9 Mbps flows; at this point, the residual capacity achieves a throughput of 57% that of shortest path, while widest path also improves on shortest path by 34%. The improvement appears to increase as the offered load increases. This is expected as it should not be difficult for even an inefficient routing metric to place flows in a lightly-loaded network.

6.1.3 Experiment 2b

The experiment using the 'pairs' flow pattern is limited to a maximum aggregate offered load of 72 Mbps (eight 9 Mbps flows; larger flows would not individually fit on any path). The 'random' flow pattern is an attempt to address this and investigate the performance of routing metrics for larger aggregate loads. In this flow pattern, n random pairs are selected, without replacement, from the list of combinations of hosts h1 - h16. Flows are created from these pairs with a fixed bandwidth $b = 1, 3, 7$. A larger aggregate load is therefore constructed from a large number of smaller flows, rather than a fixed number of variable-sized flows.

The results of an experiment using this flow pattern on the alfares topology are shown in Figure 6.3. For loads up to about 72 Mbps, relative performance for each routing metric is similar to that of Figure 6.2, though with a little more variation likely due to the different nature of the flow pattern. The number of trials used to calculate the mean aggregate throughput was increased to 15 per load/topology combination to address this; more were not possible in the time available but it is worth considering for future work.

More interestingly, after the 72 Mbps mark the performance of the residual capacity metric drops considerably: for 105 Mbps offered load, while widest path improves on shortest path by 91%, residual capacity is actually worse by a factor of 36%. The turn is so dramatic that initially external factors on the host machine were suspected; however, the effect was consistent over the 15 trials, which were taken in three sets of 5 at different times. For each of the 15 trials the flow pattern uses a different pair allocation, which means it is unlikely to be the unlucky effect of several 'difficult' flow patterns in a row; in any case, the three routing metrics are tested against the same pair allocations at each point, but no similar drop is observed.

It is unclear whether it is significant that the drop is observed after the 72 Mbps mark, which represented in the previous experiment all hosts sending or receiving flows at close to the link capacity between switches, but this is worth further investigation. The cause of the drop is also unclear; one possibility is that it occurs when the linear programming solver is unable to find a solution to the problem. This may be due to the artificially rigid constraints on the problem: theoretically, all flows have a demand and cannot fit onto links with capacity less than this demand, but in practice if this occurs the flow will simply reduce in size by slowing down its sending rate; obviously it is ideal if flows are able to be placed onto links which can accommodate their natural demand, but if they cannot it is far preferable that they be placed on a smaller link than not routed at all.

Whether this happened here, however, is not clear from the data, but again further investigation would be interesting. It would also be worth improving the multi-commodity module so that flows are always guaranteed a path, even one that is inefficient.

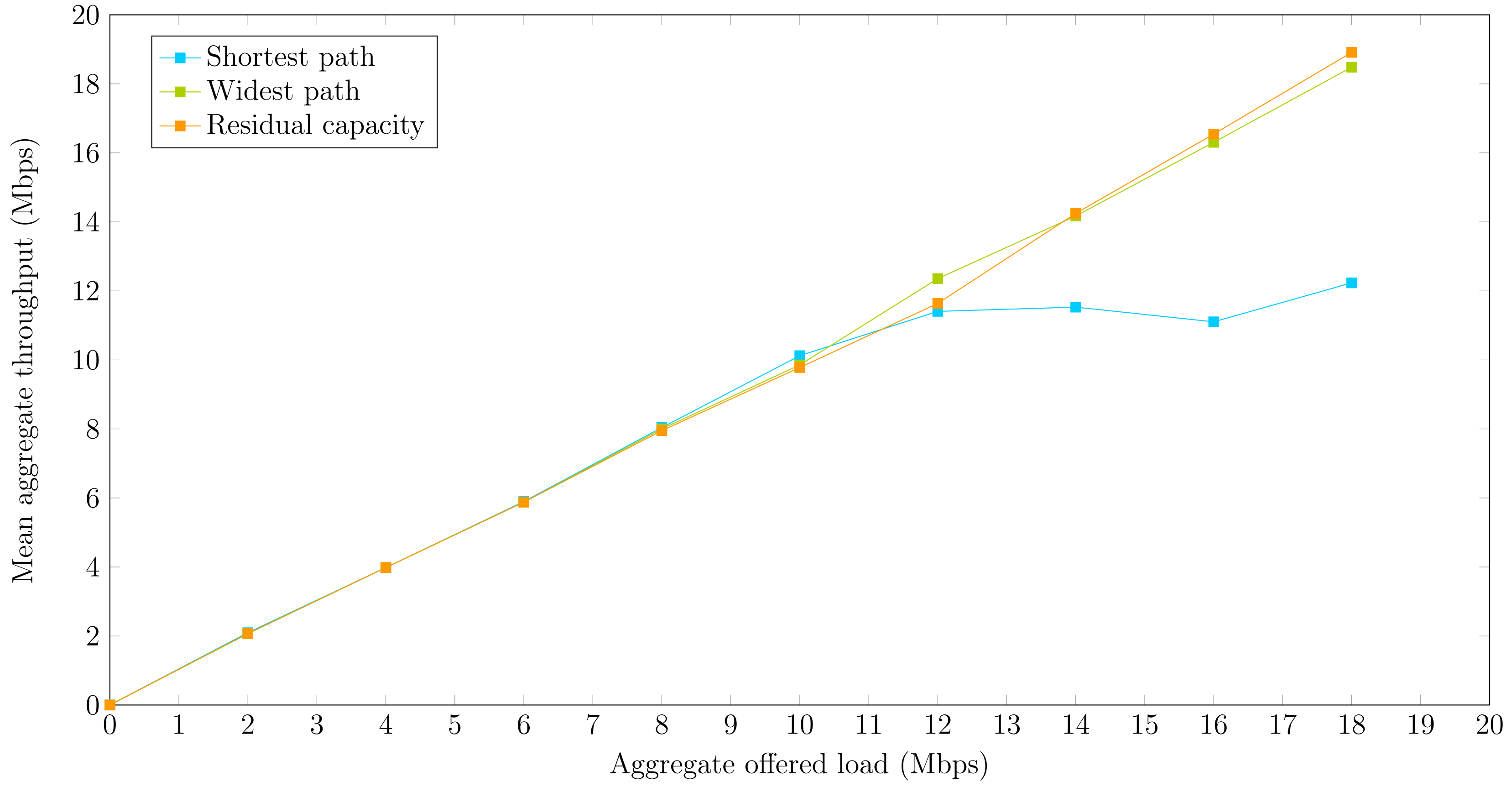


Figure 6.1: Aggregate throughput comparison for flow pattern ‘bidirectional’

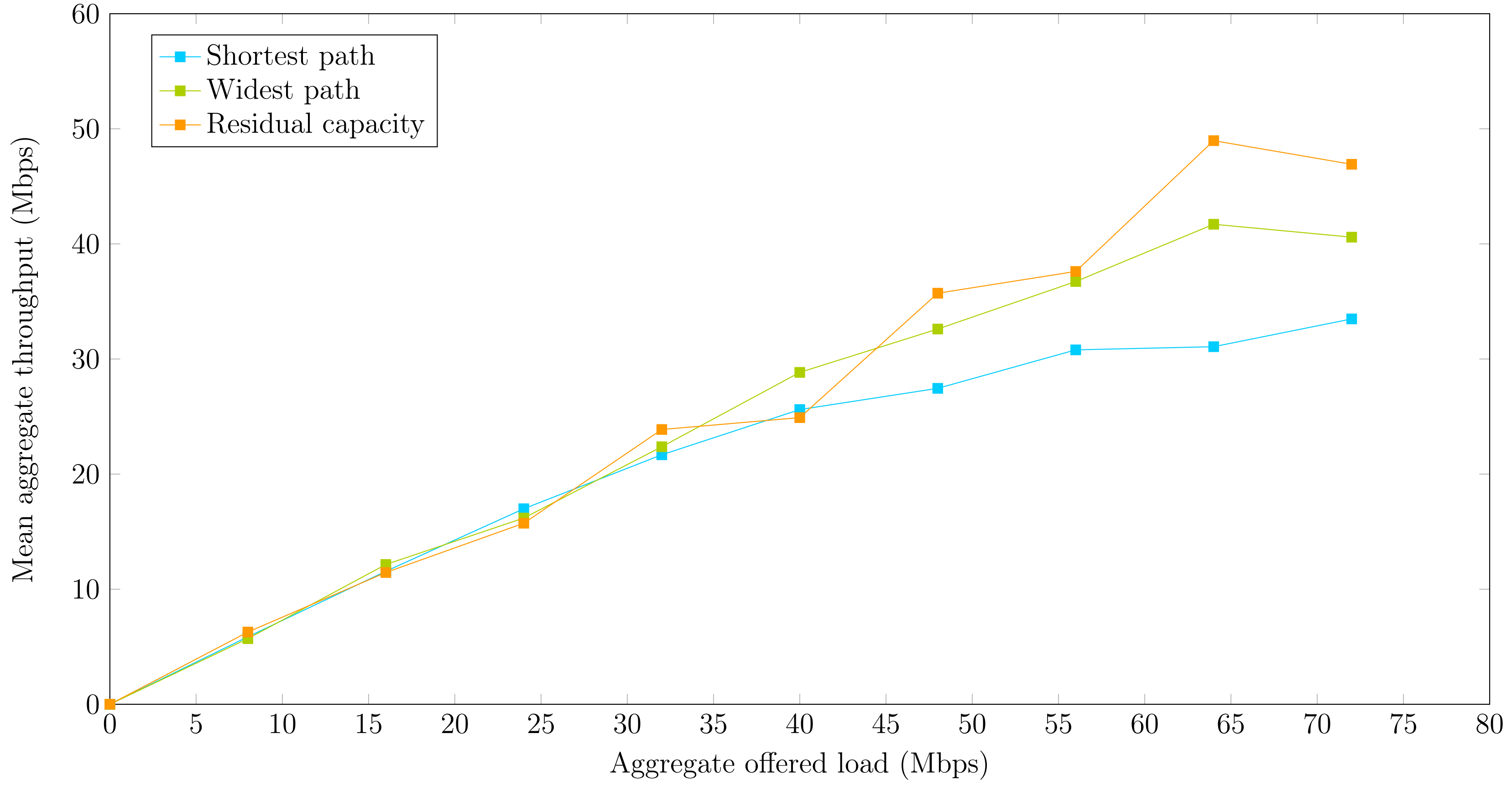


Figure 6.2: Aggregate throughput comparison for flow pattern ‘pairs’
Each mark represents the mean of 10 trials for that load/topology combination.

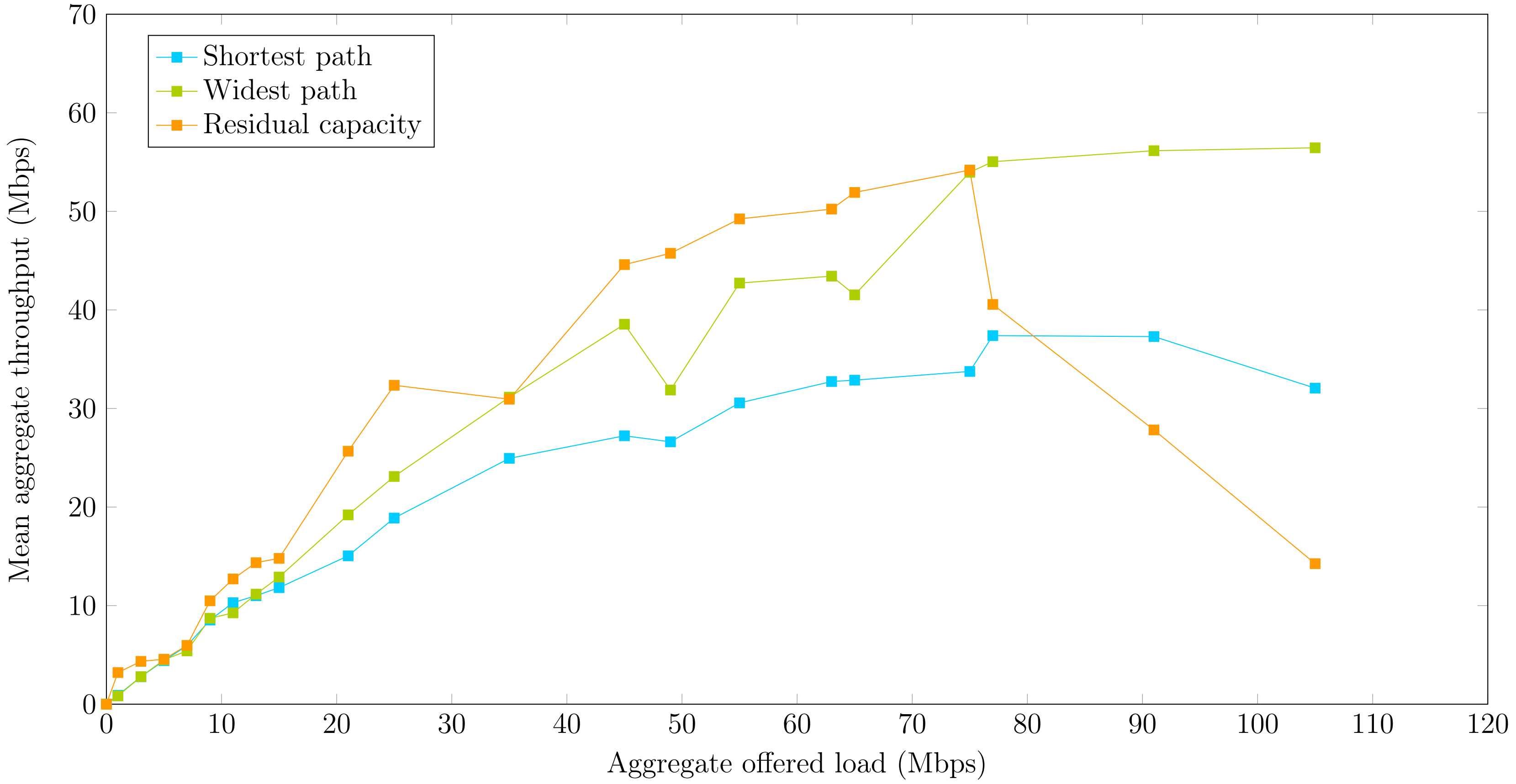


Figure 6.3: Aggregate throughput comparison for flow pattern ‘random’
Each mark represents the mean of 15 trials for that load/topology combination.

6.2 Scalability of Objective Functions

From the perspective of the experiments in this thesis, where the largest network considered was an Al-Fares fat-tree with k=4, the time taken to calculate routes is negligible for all routing metrics.

However, in general an important consideration when selecting a routing metric is the computation time required to calculate routes for all flows in the network. This is particularly important if the objective function is to be rerun every few seconds in order to ensure optimal paths are being used. It is therefore desirable to know how well the objective function scales to larger networks.

Running experiments on Mininet means that there are practical limitations of the system, chiefly availability of RAM, which limit the size of networks which can be emulated with Mininet; this limit is hit long before the scalability of objective functions becomes a factor. However, it is possible to test the scalability of individual

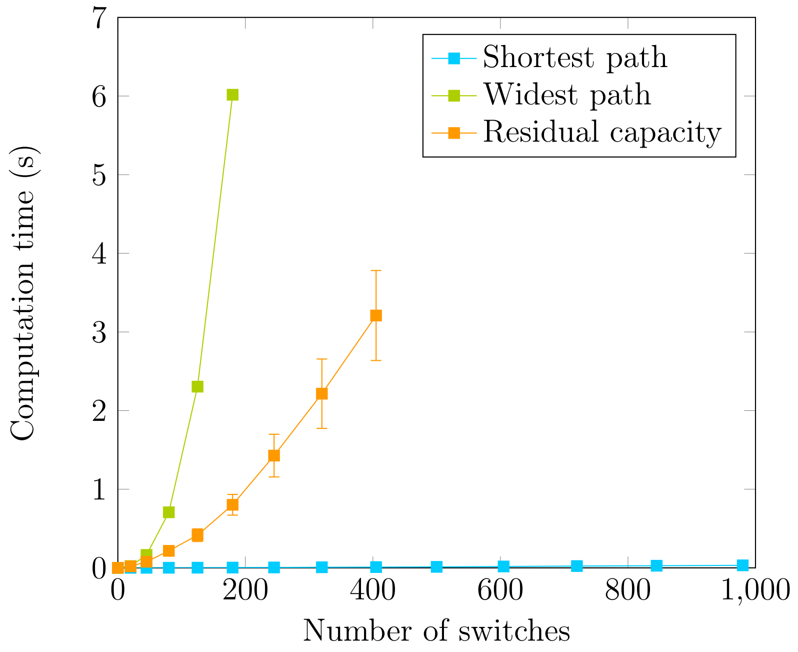


Figure 6.4: Comparison of routing algorithm scalability
Computation time is the time taken to route 16 random flows in Al-Fares fat-tree networks with increasing parameter k. Error bars show 95% confidence intervals but are too small to be visible for the shortest path and widest path metrics.

objective functions without the surrounding framework, by passing mock network graphs and flow statistics to the function.

The Al-Fares fat-tree is parametrisable by a parameter k, which indicates the number of ports per commodity switch used in the network; this produces a network with $(k/2)^2 + k^2$ switches and $k^3/4$ hosts in total, and is therefore easy to scale up to large networks for scalability testing. In this test, flows were randomly generated between 16 random hosts, for Al-Fares networks starting from k = 4 until further tests became impractically slow to run (greater than 5 seconds). The results of this testing are shown in Figure 6.4, for each of the three routing metrics.

As expected, the shortest-path metric scales extremely well. The basic algorithm is well-understood, and the particular implementation used here is the one provided with NetworkX, which includes a number of small optimisations as well.

The residual spare capacity implementation is based on the formulation described originally by Walkowiak [22]. As seen in section 3.3, the performance of the algorithm as a whole depends mostly on the number of possible paths in the network. In its original form, the solution becomes unworkably slow at $k = 6$, with just 45 switches considered. One major optimisation was therefore implemented: the length of possible paths considered is limited to 6 hops. This is enough hops to allow multiple paths to each aggregation and core switch, but not to bounce indefinitely and unnecessarily between the core, aggregation and edge layers. Figure 6.5 demonstrates how much the scalability of the residual capacity metric is dependent on the maximum path length considered.

The widest path metric performs quite poorly for such a well-known routing metric. The algorithm is based on a modified Dijkstra's algorithm, as described in section 3.2, which is not known to be particularly inefficient; the poor performance of the metric in these experiments is likely to be due to poor implementation. In particular, the algorithm must consider every path in the network, so, as for the residual ca-

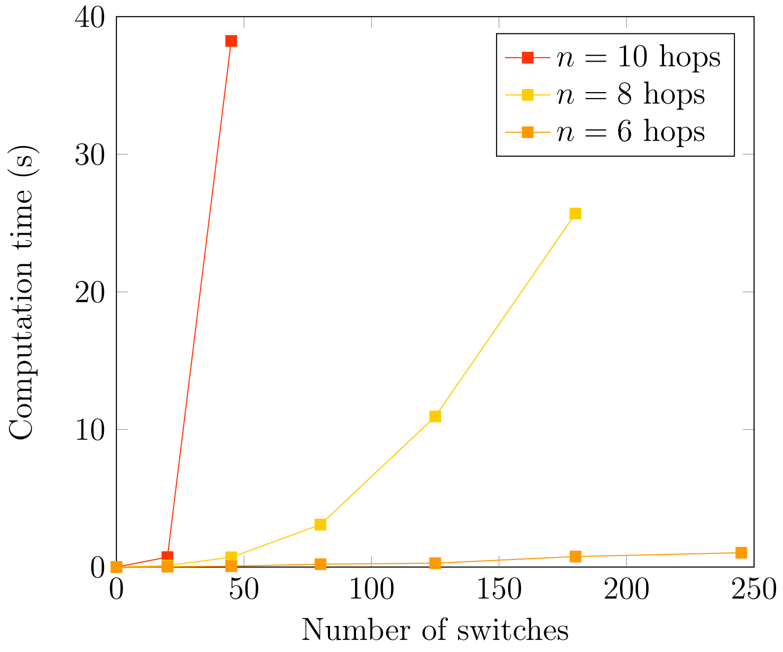


Figure 6.5: Effect of maximum path length for residual capacity metric
Computation time is as for Figure 6.4. Instead of considering all possible paths between hosts, only consider paths with a maximum path length of n hops.

pacity metric, the number of paths in an Al-Fares fat-tree can be very high and this is likely to be a significant factor in the algorithm's scalability. Unlike for residual capacity, however, limiting the number of paths considered was non-trivial due to the particular implementation, so this optimisation was not implemented. However, the dramatic effect of path length on the residual capacity metric seems to indicate that this could provide better scalability in this similar metric as well.

In general, it seems that the approach used in this thesis would be reasonable: route flows initially along the shortest path, then every few seconds, recalculate globally-optimal routes. In a normal scenario, however, the routing recalculation is performed every few seconds, and it is likely that many flows would be similar between calculations. This assumption can be made stronger if flows are consolidated; for example, group 'all HTTP traffic' along similar paths instead of calculating flow rules for individual flows, increasing the likelihood that the grouped flow will still be present in the future.

In many optimisation techniques, such as branch-and-bound (used in this work by GLPK [1]) and simulated annealing (used in [9]), starting from an estimated solution which is close to the optimum greatly decreases the time taken to reach optimal levels. Here, the previous route allocation could be used as such an estimated solution. However, the implementation of this is non-trivial, and limits the use of third-party interfaces such as PuLP, since there is no way to specify a starting estimate for most solvers. This is far outside the scope of this project and definitely in the realm of 'future work'.

Chapter 7

Conclusion

The project has successfully achieved both of its aims: to create a framework to allow evaluation of routing metrics for multicommodity routing, and to perform basic analysis using the framework and report the results.

The primary software contribution of this thesis is a configurable SDN controller and experiment framework for analysis and testing. This software is supported by a set of objective functions and Mininet topologies (as discussed in this report), as well as sample experiment scripts used to produce the graphs, and various helper functions. Documentation in the form of a user guide and API reference have also been produced and are available on the companion disk.

Analysis of the performance of three different routing metrics was performed. This analysis was intended to be primarily to test ease of use of the software, and indeed these efforts identified improvements to overall workflow of the system. The results of the analysis appeared to support the hypothesis that global consideration of the network can increase routing efficiency, with the widest path and residual capacity metrics both improving performance over shortest path. Where residual capacity was able to find an optimal solution, performance was generally better than widest path; however, throughput decreased significantly when no solution could be found.

7.1 Future Work

The analysis performed in this thesis was limited in scope, primarily due to it not being the main focus of this research. The number of routing algorithms studied was limited to three and the number of topologies to just two, one small and one large. Future work could use the framework as it stands to conduct more experiments using a wider range of metrics and topologies. Improvements to particular experiments are discussed in Chapter 6; notably, an investigation into the sudden but consistent drop in performance for the residual capacity metric would be worthwhile.

The flow patterns were also limited: only TCP/IP flows were used in experiments, and all flows arrived at the same time and continued to the end of the experiment. The experiment framework currently does allow flows to be started at separate, arbitrary times; however, the controller only runs route recalculation once, at a predetermined time, which effectively limits when flows can be run. Modifying the controller to run route calculations periodically would remove this restriction and allow testing of a greater and more realistic range of flow patterns.

Finally, as mentioned previously, `mcfpox` is intended to be extended by other members of Marius Portmann's SDN group, intended eventually to be used for research into routing in wireless mesh networks. Work on this thesis has identified a number of issues which, while out of scope here, nevertheless deserve attention. Most notably, flow sizes are very difficult to estimate for TCP flows due to congestion control; this is a significant problem as global solutions are often sensitive to the estimated demands and bad estimates can result in inefficient placement of flows. Additionally, the current implementation cannot dynamically calculate link capacities and relies on hardcoded values; this greatly limits the application of this work to wireless networks, where link capacities can vary dynamically depending on physical location and other factors.

Appendix A

Companion Disk

A.1 Directory Structure

The following directories are contained on the companion CD.

results Processed graphs for each experiment

source Source code for all controllers, Mininet topologies and experiment scripts

documentation Short user guide and API reference for controller modules

Appendix B

Source Code

B.1 POX Modules

B.1.1 Network Discovery

```
#!/usr/bin/python

"""
Multicommodity flow module: decides where different flows should go
"""

from pox.core import core
from pox.lib.addresses import IPAddr
from pox.lib.recoco import Timer
from pox.lib.revent import *
import pox.openflow.libopenflow_01 as of
from pox.openflow.of_json import *

import pox.openflow.discovery as discovery
import pox.host_tracker.host_tracker as host_tracker

from collections import namedtuple
import networkx as nx
from pulp import *

log = core.getLogger()

class Network(EventMixin):
    _core_name = "thesis_topo"

    def __init__(self):
        """
        Initialise topology module.
        """
        self.graph = nx.Graph()
        self.ht = host_tracker.host_tracker()

        core.openflow_discovery.addListeners(self)
        self.listenTo(self.ht)
        core.addListeners(self)


    def refresh_network(self):
        """
        Update view of network from discovery and host_tracker adjacency lists.
        """
        G = nx.DiGraph()
        G.clear()
        self.host_count = 0

        # add switches
        for link in core.openflow_discovery.adjacency:
            s1 = "s{0}".format(link.dpid1)
            s2 = "s{0}".format(link.dpid2)
            G.add_nodes_from([s1,s2])
            G.add_edge(s1, s2, {'capacity':10e6, 'port':link.port1})
            G.add_edge(s2, s1, {'capacity':10e6, 'port':link.port2})

        # add hosts
        for src, entry in self.ht.entryByMAC.items():
            if entry.port == 65534: # controller port
                continue
            if not core.openflow_discovery.is_edge_port(entry.dpid, entry.port):
                continue

            self.host_count += 1
            h = "h{0}".format(self.host_count)
            s = "s{0}".format(entry.dpid)
            G.add_nodes_from([h,s])
            if entry.ipAddrs.keys():
                G.node[h]['ip'] = str(next(iter(entry.ipAddrs.keys())))
            G.add_edge(h, s, {'capacity':20e6})
            G.add_edge(s, h, {'capacity':20e6, 'port':entry.port})

        self.graph = G


def launch():
    core.registerNew(Network)
```

B.1.2 Flow Statistics

```python
#!/usr/bin/python

"""
Statistics-gathering module: track size of flows in the network
"""

from pox.core import core
from pox.lib.recoco import Timer
import pox.openflow.libopenflow_01 as of
from pox.openflow.of_json import flow_stats_to_list
from collections import namedtuple

import mcfpox.controller.lib as lib

log = core.getLogger()


class Statistics:
    _core_name = "thesis_stats"

    def __init__(self, period=5, length=1):
        """
        Initialise statistics module.
        period: number of seconds between switch statistics requests
        length: window size for moving-window estimation, unused
        """
        self.period = 2.0
        Timer(self.period, self._request_stats, recurring=True)
        self.flows = []
        core.openflow.addListeners(self)


    def _request_stats(self):
        """
        Send OFPT_STATS_REQUEST messages to every known switch.
        """
        for connection in core.openflow._connections.values():
            connection.send(
                    of.ofp_stats_request(body=of.ofp_flow_stats_request()))


    def _handle_FlowStatsReceived(self, event):
        """
        Handle OFPT_STATS_REPLY messages from switches.
        Add a new entry if flow is previously unseen, else update the old.
        """
        stats = flow_stats_to_list(event.stats)
        switch = event.dpid
        local = [e for e in self.flows if e.switch == switch]
        log.info("stats on switch {0}".format(switch))
        log.info(stats)

        for flow in stats:
            f = lib.match_to_flow(flow['match'])
            if not f:
                continue

            try:
                entry = next(e for e in local if e.flow == f)
            except StopIteration:
                entry = lib.Entry(switch, f)
                self.flows.append(entry)

            bc = int(flow['byte_count']) * 8  # want everything in bits
            entry.recent = (bc - entry.total) / self.period
            entry.total = bc
            if entry.recent <= 20000:
                self.flows.remove(entry)


    def get_flows(self):
        """
        Combine flows from all switches into single list of (flow,size) pairs.
        """
        self.flows.sort(key=lambda e: e.recent, reverse=True)
        flows = [e.flow for e in self.flows if e.recent]
        overall = []
        seen = []
        for f in flows:
            g = lib.Flow(f.nw_proto, f.nw_dst, f.nw_src, f.tp_dst, f.tp_src)
            if f in seen or g in seen:
                continue
            s = [e.recent for e in self.flows if e.flow == f]
            overall.append((f,max(s)))
            seen.append(f)
        log.info(overall)
        return overall


def launch(period=5, length=1):
    core.registerNew(Statistics, period=period, length=length)
```

B.1.3 Routing Control

```python
#!/usr/bin/python

"""
Multicommodity flow module: route flows according to an objective function
"""

from pox.core import core
from pox.lib.recoco import Timer
import pox.openflow.libopenflow_01 as of
from pox.openflow.of_json import *
from pox.lib.addresses import IPAddr

from mcfpox.controller import topology
from mcfpox.controller.lib import Flow, Hop, Entry
from mcfpox.objectives import shortest_path

from collections import namedtuple
import time

log = core.getLogger()


class Multicommodity:
    _core_name = "thesis_mcf"


    def __init__(self, objective, preinstall):
        """
        Initialise multicommodity module.
        objective: objective function for calculating routes
        preinstall: list of flow rules to preinstall on switches
        """
        Timer(30, self._update_flows)

        self.flows = {}
        self.net = core.thesis_topo
        self.stats = core.thesis_stats
        self.objective = objective

        self.preinstall = preinstall
        self.log_rules = False
        self.done = False

        core.openflow.addListeners(self)
        core.addListeners(self)


    def _handle_PacketIn(self, event):
        """
        Handle arrival of new flow.
        Install routes in both directions using the shortest path metric.
        """
        try:
            if self.done:
                return
            self.net.refresh_network()
            packet = event.parsed
            if packet.find('tcp'):
                ip = packet.next
                tcp = ip.next
                if str(ip.srcip) != '0.0.0.0':
                    log.info("packetin {1}:{3} to {2}:{4} at {0}".format(
                            time.clock(), str(ip.srcip), str(ip.dstip),
                            tcp.srcport, tcp.dstport))
                    there = Flow(6, str(ip.srcip), str(ip.dstip),
                            tcp.srcport, tcp.dstport)
                    back = Flow(6, str(ip.dstip), str(ip.srcip),
                            tcp.dstport, tcp.srcport)
                    rules = shortest_path.objective(self.net.graph,
                            [(back,0), (there,0)])
                    self._install_rule_list(rules)
        except Exception as e:
            print str(e)


    def _install_forward_rule(self, msg, hops):
        """
        Install forwarding rule for all ports listed in hops.
        """
        for switch in hops:
            msg.actions = []
            msg.actions.append(of.ofp_action_output(port = switch.port))
            if self.log_rules:
                print "{0}.{1}".format(switch.dpid, switch.port),
            core.thesis_base.switches[switch.dpid].connection.send(msg)
        if self.log_rules:
            print


    def _install_rule_list(self, rules):
        """
        Install forwarding rules for all flows listed in rules.
        """
        for flow,hops in rules.items():
            msg = of.ofp_flow_mod()
            msg.command = of.OFPFC_MODIFY
            msg.match.dl_type = 0x800
```

```python
            msg.match.nw_proto = 6
            msg.match.nw_src = flow.nw_src
            msg.match.nw_dst = flow.nw_dst
            msg.match.tp_src = int(flow.tp_src)
            msg.match.tp_dst = int(flow.tp_dst)
            if self.log_rules:
                print "Installing rule for {0}:".format(flow, time.clock()),
            self._install_forward_rule(msg, hops)


    def _preinstall_rules(self):
        """
        Log and then install preinstall rule list.
        """
        log.info("Preinstalling rules...")
        log.info("Rules are:")
        log.info(self.preinstall)
        self._install_rule_list(self.preinstall)


    def _solve_mcf(self):
        """
        Recalculate routes using the provided objective, then install them.
        """
        self.log_rules = True
        log.info("Flows are " + str(self.flows))
        print "Starting objective with flows:"
        for flow, demand in self.flows:
            print "{0}: {1:.2f} Mbps".format(flow, demand/1e6)
        rules = self.objective(self.net.graph, self.flows)
        log.info("Rules are " + str(rules))
        print
        self._install_rule_list(rules)
        self.done = True


    def _update_flows(self):
        """
        Update view of network and statistics, then begin recalculating routes.
        """
        self.net.refresh_network()
        self.flows = self.stats.get_flows()
        self._solve_mcf()



def default_objective(net, flows):
    print "Default objective function, given:"
    print "net:", net
    print "flows:", flows
```

```python



def launch(objective=default_objective, pre={}):
    core.registerNew(Multicommodity, objective=objective, preinstall=pre)
```

B.1.4 Utility Library

```python
"""
Library of data structures shared across the framework
"""


class EqualityMixin(object):
    """
    Mixin to add equality comparators based on class fields.
    """
    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return False

    def __ne__(self, other):
        return not self.__eq__(other)


class Hop(EqualityMixin):
    """
    A network hop, represented by the source switch and port to exit from.
    """
    def __init__(self, dpid, port):
        self.dpid = dpid
        self.port = port

    def __repr__(self):
        return "{0}.{1}".format(self.dpid, self.port)


class Flow(EqualityMixin):
    """
    A network flow, represented by the 5-tuple of information.
    """
    def __init__(self, nw_proto, nw_src, nw_dst, tp_src, tp_dst):
        self.nw_proto = nw_proto
        self.nw_src = nw_src
        self.nw_dst = nw_dst
        self.tp_src = tp_src
        self.tp_dst = tp_dst

    def __repr__(self):
        return "(proto {0}) {1}:{2} -> {3}:{4}".format(self.nw_proto,
                self.nw_src, self.tp_src, self.nw_dst, self.tp_dst)


class Entry(EqualityMixin):
    """
    A flow statistics entry, summarising the recent size of a flow.
    """
    def __init__(self, switch, flow):
        self.switch = switch
        self.flow = flow
        self.recent = 0
        self.total = 0

    def __repr__(self):
        return "flow {0} on switch {1}: {2} bytes recently, {3} total".format(
                self.flow, self.switch, self.recent, self.total)


def match_to_flow(match):
    """
    Convert POX representation of statistics match to Flow class, if possible.
    """
    d = match if type(match) == dict else match_to_dict(match)
    try:
        f = { k:d[k] for k in ["nw_proto", "nw_src", "nw_dst", "tp_src", "tp_dst"]}
        f['nw_src'] = str(f['nw_src']).split('/')[0]
        f['nw_dst'] = str(f['nw_dst']).split('/')[0]
        flow = Flow(**f)
        return flow
    except KeyError:
        return None
```

B.2 Experiment Module

```python
"""
Experiment framework module: boilerplate code for network experiments
"""

import re
import os
import sys
import json
import time
import sched
import subprocess
from multiprocessing import Process
from termcolor import colored

from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.node import RemoteController


def start_log(log_dir):
    """
    Initialise logging directory with either suggested name or timestamped name.
    """
    if not log_dir:
        log_dir = 'log.{0}'.format(int(time.time()))
    subprocess.call(['mkdir', '-p', log_dir])
    return {'log_dir': log_dir}


def start_net(net, logs):
    """
    Start a previously-created network. Logging is currently disabled.
    """
    print "Starting mininet: no logs"
    net.start()
    return net


def start_pox(logs, level={}, module='mcfpox.controller.base',
              objective=None, rules=None, poxdesk=False):
    """
    Launch the POX controller.
    Note: relies on the github.com/krman fork of POX.
    """

    log_level = {
        'INFO': True,
        'packet': 'WARN',
    }
    log_level.update(level)

    logs['pox'] = 'pox.log'
    print "Starting POX: log files in {0}".format(logs['pox'])

    args = {
        module: [{
            'objective': objective
        }],
        'log.level': [log_level]
    }

    if poxdesk:
        args.update({
            #'samples.pretty_log': [{}],
            'web': [{}],
            'messenger': [{}],
            'messenger.log_service': [{}],
            'messenger.ajax_transport': [{}],
            'openflow.of_service': [{}],
            'poxdesk': [{}],
            'poxdesk.tinytopo': [{}],
        })

    def start(components):
        from pox.boot import boot
        out_log = os.path.join(logs['log_dir'], 'pox.out')
        err_log = os.path.join(logs['log_dir'], 'pox.err')
        #sys.stdout = open(out_log, 'w')
        sys.stderr = open(err_log, 'w')
        boot({'components':components})

    process = Process(target=start, args=(args,))
    process.start()

    return process


def start_iperf(src, dst, port, bw, server_log, client_log):
    """
    Start iperf with given parameters and log both server and client.
    """
    server_cmd = "iperf3 -s -p {0} &> {1} &".format(
            port, server_log)

    blbw = str(int(bw[:-1]) * 1000.0/1070) + 'm'
    client_cmd = "iperf3 -c {0} -p {1} -b {2} -t 10 -O 10 -J &> {3}&".format(
            dst.IP(), port, blbw, client_log)
```

```python

    print "Flow: {0} from {1} ({2}) to {3} ({4})".format(
            bw, src, src.IP(), dst, dst.IP())

    dst.cmd(server_cmd)
    src.cmd(client_cmd)


def start_flows(flow_schedule, net, logs):
    """
    Schedule iperf flows in network net according to given flow schedule.
    """
    s = sched.scheduler(time.time, time.sleep)
    port = 5101
    flow_logs = []
    log_dir = logs['log_dir']

    longest_delay = 0
    for delay, flows in flow_schedule.iteritems():
        for flow in flows:
            src = net.get(flow[0])
            dst = net.get(flow[1])
            bw = flow[2]

            server_log = 'server.{0}.{1}'.format(dst, port)
            client_log = 'client.{0}.server.{1}.{2}'.format(src, dst, port)
            flow_logs.append((server_log, client_log))

            s.enter(delay, 1, start_iperf, (src, dst, port, bw,
                    os.path.join(log_dir, server_log),
                    os.path.join(log_dir, client_log),))
            port += 1
        longest_delay = max(longest_delay, delay)

    logs['flows'] = flow_logs
    time.sleep(1)

    print "\nStarting scheduled flows: iperf output in server/client logs"
    s.run()
    print

    time.sleep(longest_delay+15)


def start(scenario, config, log_dir=None):
    """
    Start POX and Mininet with the given configuration.
    """
    logs = start_log(log_dir)
    print "Starting experiment: log files in {0}".format(logs['log_dir'])

    try:
        net = start_net(scenario['net'].create_net(), logs)
        controller = start_pox(logs, **config)

        start_flows(scenario['flows'], net, logs)

        net.stop()
        controller.terminate()

    except KeyboardInterrupt as e:
        print "Exiting on user command"
        raise
    finally:
        print colored("End of experiment:", "green", attrs=['bold']),

    return results(scenario, config, logs)


def results(scenario, config, logs):
    """
    Process iperf results and write summary to file.
    Return measured throughput array for all flows.
    """
    log_dir = logs['log_dir']
    recv = []

    src_pattern = re.compile("client.(h[0-9]+).server.(h[0-9]+).([0-9]+)")

    print

    for slog, clog in logs['flows']:
        server_log = os.path.join(log_dir, slog)
        client_log = os.path.join(log_dir, clog)

        src_host, dst_host, dst_port = src_pattern.match(clog).groups()

        with open(client_log, 'r') as f:
            try:
                j = json.load(f)
                r = j['end']['sum_received']['bits_per_second']
                recv.append(r)
            except (ValueError, KeyError):
                r = 0.0
                recv.append(0)

        string = "{0} - {1}:{2}: {3} bps".format(src_host, dst_host, dst_port, r)
        print colored(string, "green")

    pox_log = os.path.join(log_dir, "pox.err")
    process = subprocess.Popen(["grep", "Rules", pox_log],
                    stdout=subprocess.PIPE)
    output, err = process.communicate()

    process.wait()
    rules = output.split("Rules are ")[-1]

    scenario["net"] = scenario["net"].__name__
    config["objective"] = config["objective"].__module__

    summary = {
        "scenario": scenario,
        "config": config,
        "results": recv,
        "rules": rules,
        "logs": log_dir
    }

    string = "Total throughput {0} Mbps".format(sum(recv)/1e6)
    print colored(string, "green", attrs=['bold'])

    with open("results.json", 'a') as f:
        f.write(json.dumps(summary, sort_keys=True,
                indent=4, separators=(',', ': ')))
        f.write("\n")

    return recv
```

References

[1] GNU Linear Programming Kit. Available: http://gnu.org/software/glpk/.

[2] iperf3. Available: http://software.es.net/iperf/.

[3] NetworkX graph library. Available: http://networkx.github.io/.

[4] POX controller framework. Available: http://www.noxrepo.org/.

[5] OpenFlow switch specification version 1.0.0 (protocol version 0x01). Technical report, Palo Alto, CA, USA, 2009.

[6] Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation, Palo Alto, CA, USA, 2012.

[7] OpenFlow switch specification version 1.4.0 (protocol version 0x05). Technical report, Palo Alto, CA, USA, 2013.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data centre network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 63–74, 2008.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, page 19, 2010.

[10] A. Anagnostopoulos, F. Grandoni, S. Leonardi, and A. Wiese. A Mazing 2+ε approximation for unsplittable flow on a path. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 26–41, 2014.

[11] P. Bonsma, J. Schulz, and A. Wiese. A constant factor approximation algorithm for unsplittable flow on paths. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science*, pages 47–56, 2011.

[12] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar. Approximation algorithms for the unsplittable flow problem. In *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 51–66, 2002.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[14] L. Dai, Y. Xue, B. Chang, Y. Cao, and Y. Cui. Optimal routing for wireless mesh networks with dynamic traffic demand. *Mobile Networks and Applications*, 13:97–116, 2008.

[15] P. Dely, A. Kassler, and N. Bayer. Openflow for wireless mesh networks. In *Proceedings of the 20th International Conference on Computer Communications and Networks*, pages 1–6, 2011.

[16] N. Handigol, M. Flajslik, S. Seetharaman, N. McKeown, and R. Johari. Aster*x: Load-balancing as a network primitive. In *Proceedings of ACLD '10: Architectural Concerns in Large Datacenters*, 2010.

[17] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Mininet performance fidelity benchmarks. Technical report, Stanford University, Stanford, CA, USA, 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, 2012.

[19] N. McKeown. How SDN will shape networking. Open Networking Summit, Stanford, CA, USA, 2011.

[20] 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*, 38:69–74, 2008.

[21] D. Medhi and K. Ramasamy. *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann, 1st edition, 2010.

[22] K. Walkowiak. Maximising residual capacity in connection-oriented networks. *Journal of Applied Mathematics and Decision Sciences*, 2006. Article ID 72547.

[23] W. Wang, X. Liu, and D. Krishnaswamy. Robust routing and scheduling in wireless mesh networks under dynamic traffic conditions. *IEEE Transactions on Mobile Computing*, 7:1705–1717, 2009.

[24] J. Wellons, L. Dai, Y. Xue, and Y. Cui. Augmenting predictive with oblivious routing for wireless mesh networks under traffic uncertainty. *Computer Networks*, 54:178–195, 2010.

[25] J. Wellons and Y. Xue. Oblivious routing for wireless mesh networks. In *Proceedings of the 2008 IEEE International Conference Communications*, pages 2969–2973, 2008.