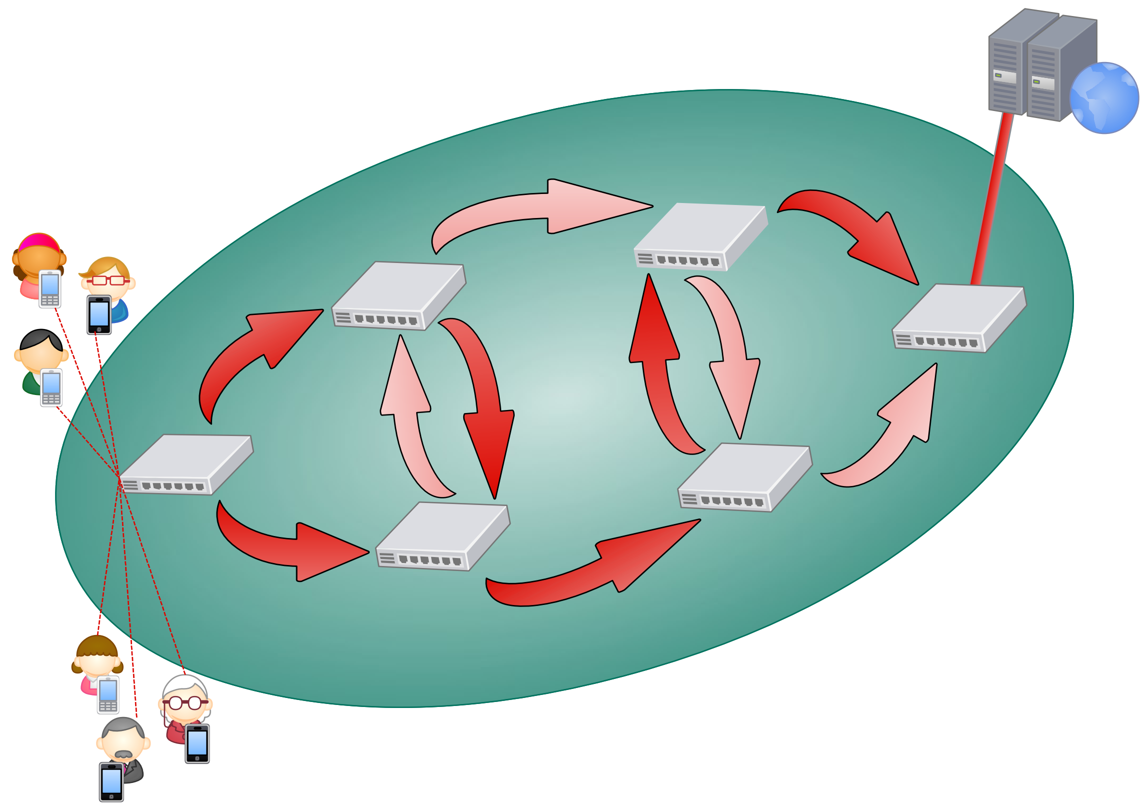

Performance Management in Software Defined Networking

Master of Science Thesis in Computer Science – Algorithms, Languages and Logic

JACOB ANDERSSON
ERIK TERMANDER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2015

Performance Management in Software Defined Networking

JACOB ANDERSSON
ERIK TERMANDER

Examiner: OLAF LANDSIEDEL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
End users connecting to an endpoint of a microwave mobile backhaul carrying their traffic to the core of the network.

Department of Computer Science and Engineering
Göteborg, Sweden, January 2015

Abstract

In computer networks, one commonly performs load-balancing assuming fixed capacities of the links in the network. In the microwave backhaul portion of a telecommunications network however, various dynamic effects such as rain and snow affect the capacities of the links. Active measurement of the performance along the available paths in the network can detect these capacity fluctuations. Combining the resulting performance knowledge with an SDN controller makes it possible to reroute traffic around microwave links having reduced capacity, with the goal of balancing the load in the mobile backhaul and improve the overall performance. This thesis simulates a microwave mobile backhaul utilizing SDN and static load-balancing to determine if the addition of active Performance Management can decrease the drop level of the network by enabling dynamic load-balancing.

The proposed design is a dynamic load-balancer which uses an iterative approximation of a congestion minimization algorithm, measuring congestion using end-to-end latencies. We evaluate the dynamic load-balancer against a "base case" static load-balancer splitting all traffic evenly across all network paths, and an optimal load-balancer assumed to have exact, real-time knowledge of the link capacities. The results show a clear improvement in drop level as compared to the static load-balancer when simulating a microwave mobile backhaul and modeling the microwave links using an arbitrary link degradation scheme.

We show that performing active PM and using the resulting end-to-end latencies as a basis for making dynamic load-balancing decisions can improve the overall performance of a simulated microwave mobile backhaul. The main contribution of this work is the establishment of active PM as a potentially viable candidate for performing dynamic load-balancing in the context of microwave mobile backhauls.

Acknowledgements

This Master's Thesis was carried out in collaboration with Ericsson at their premises on Lindholmspiren in Gothenburg. We would like to dedicate a special thanks to our supervisors Hans-Åke Lund and Olof Hagsand for sharing their expert knowledge within the domain and for their general help and support along the way.

We would also like to thank our supervisor at Chalmers, Olaf Landsiedel, for his patience in steering us in the right direction and for providing valuable input.

Jacob Andersson and Erik Termander

Göteborg, 2014

Table of Contents

List of Figures

List of Tables

1 Introduction

SDN (Software Defined Networking) is a technique to separate the forwarding elements of a network from the routing logic by using generic elements and protocols, such as OpenFlow [1], and a centralized controller managing the network elements. The technique thus allows software to control routers and switches, making it possible to e.g. deploy and test new routing algorithms in existing networks.

In recent years, SDN has been used to cope with architectural challenges in e.g. data centers. The growing popularity of SDN has also lead to research concerning new areas of application, and it has been suggested that SDN could be a good fit for microwave mobile backhauls [2]. The application of SDN in mobile backhauls could lead to a number of benefits by enabling centralized management.

When applying SDN to a mobile backhaul, it presents an opportunity to carry out Performance Management (PM). PM measures key indicators for network performance such as latency and loss. By using such indicators in combination with decision-making algorithms, it is possible to perform load-balancing in the mobile backhaul. The application of SDN in combination with PM and a load-balancing algorithm is especially interesting in the context of microwave mobile backhauls, since e.g weather conditions dynamically affect the microwave links. When microwave links degrade, it should then be possible to dynamically reroute traffic around microwave links with reduced capacity.

1.1 Problem statement

In a mobile backhaul, the network carries traffic between radio base stations at the edges to the services in the core of the network. If the backhaul contains microwave links, dynamic effects including weather conditions may lead to certain links building up packet queues causing high latencies and occasionally packet drops. Traditional routing algorithms such as OSPF and RIP are not designed to on their own handle such fluctuations in link bandwidth [3, 4]. In network topologies such as mesh and ring where multiple paths are available, this presents an opportunity for a more capable routing algorithm that incorporates PM indicators.

A PM method can detect performance fluctuations in the network, and if combined with an SDN controller, the controller can dynamically reconfigure the network when bandwidth fluctuations occur. A load-balancer can then process the gathered performance data and use it to make active routing decisions, rerouting traffic around microwave links with reduced capacities, with the goal improving the overall network performance in terms of packet loss levels and latencies.

1.2 Purpose

The purpose of this work is to simulate a microwave mobile backhaul network utilizing SDN and static load-balancing and determine if the addition of active PM can improve the overall network performance in terms of drop level by adding dynamic load-balancing capabilities based on PM indicators.

1.3 Limitations

For practical purposes, we performed the evaluation and testing of the solution in a simulated environment using software switches (Open vSwitch [5]). Network simulations involving many nodes are resource intensive, meaning that we had to limit the number of nodes and consequently the number of alternative paths in the network to allow evaluation on a single computer.

Given that there is a substantial difference between a simulation and a real-world microwave backhaul, it was not feasible to compare the performance of the solution in the simulated environment to any implementation in an actual microwave mobile backhaul. Instead, we compared the dynamic load-balancing solution based on PM metrics to static load-balancing in the same simulated environment as well as a theoretical optimal solution assuming real-time knowledge of the throughput on all links.

Furthermore, the quality of the results depends to a substantial degree on the modeling of the microwave links, their unique characteristics, and realistic link degradation scenarios. The actual modeling of microwave links was simplified due to lack of access to detailed data from actual microwave mobile backhauls and to keep the scope feasible.

Network security aspects were out of scope and did not impact the design, implementation, or evaluation of our solution.

1.4 Method

We employed the following methodology to develop and evaluate a simulation prototype representing a microwave mobile backhaul network with PM and dynamic load-balancing capabilities:

1. Background study

   Initially, we performed some reading on SDN, SDN controllers, PM measurement techniques, and microwave mobile backhaul topologies. This was necessary in order to be able to set up an appropriate simulation prototype.

2. Development of a prototype by extending an existing SDN platform with a PM controller module

   We developed the infrastructure using Mininet [6], OpenFlow, Floodlight [7], and a suitable PM implementation. Microwave links were modeled in Mininet using approximate data provided by previous research describing dynamic effects. Network traffic was generated using IMIX [8] data.

3. Extension of the prototype by developing a load-balancing algorithm for making dynamic routing decisions

We investigated different existing methods/algorithms for performing load-balancing in the context of SDN and used the research as a starting point for development of the final load-balancing method. We designed the load-balancing method to make decisions based on performance indicators given by the PM component.

4. Simulation and evaluation of the prototype in terms of performance (packet loss levels)

   We evaluated the developed prototype by comparing loss level of the developed solution with the loss level given by the static load-balancing counterpart (a default SDN controller without a PM component and dynamic load-balancing capabilities).

1.5 Outline

The report starts with Chapter *2 Background*, which provides a foundation to assist the reader in understanding the different parts of the design and implementation. The chapter covers the most relevant aspects of SDN, PM, and microwave mobile backhauls.

Next comes Chapter *3 Design* which describes the overall concept of the design and outlines the design details of each component forming the complete solution. It discusses the design choices made on a per component basis as a way of breaking down the complexity and focus on each individual subtask.

Chapter *4 Implementation* presents the actual implementation based on the design choices. This chapter is divided into different components to make a clear connection to Chapter *3 Design*.

Chapter *5 Evaluation* describes the evaluation of the final solution. It specifies the criteria and the compared load-balancers as well as the evaluation setup and procedure. The chapter ends with a combined results and discussion section, presenting and discussing the results of the evaluation.

Chapter *6 Conclusion* discusses the findings of the thesis, provides direction for future work, and concludes the report.

The very last part of the report holds a complete reference list as well as an *Appendix* containing additional and related information supporting the material presented in this report.

2 Background

In this chapter, we provide a general background introducing the concepts and techniques used throughout this thesis. The purpose of the presented material is to augment the understanding of Chapter *3 Design* and Chapter *4 Implementation*.

2.1 Software Defined Networking (SDN)

SDN is a concept of separating the control-plane from the forwarding-plane in a network device [9]. In traditional switching, these planes are commonly located in the same device (i.e. the switch or router). Separation of the control-plane and the forwarding-plane means that the decision of where to forward a packet (control) and the action of forwarding it (forward) is not bound to be made on the same device. Instead, a centralized device with more knowledge about the state of the network can make the decisions.

2.1.1 OpenFlow

OpenFlow [10] is an open, standard protocol which falls within the definition of SDN. It is an important building block when constructing SDN solutions. The terms SDN and OpenFlow are often used synonymously, but they are not identical. SDN is the overlaying idea and concept, while OpenFlow is one of several alternative protocols that enables the use of SDN in a network.

The OpenFlow protocol [11] enables an SDN controller to remotely insert rules into the switching table of a switch. In effect, this insertion capability makes it possible for a central entity to have complete and fine-grained control of all switching activities in an arbitrary L2-network.

In more detail, OpenFlow allows an SDN controller to add, update, or delete *flow entries* from the *flow table* in a switch. A flow entry (or flow rule) is a basic structure containing a set of match fields and a set of instructions describing which action to apply to a packet. When a packet arrives, the switch scans the flow table for a flow entry that matches the packet, and if found, the switch applies the action of the flow entry to the packet. An example of a common action is to output the packet to a specific physical switch port.

Matching occurs on the content of the packet header, and common matching use-cases include ethernet or IP address, TCP or UDP port, VLAN id, or physical switch port. A flow entry defines a *flow*, the set of packets in a network which shares the same values for some fields in their header, such as their destination IP address, their protocol type, or their VLAN id.

OpenFlow also allows a controller to organize flow entries in a switch into groups, stored in a separate *group table*. In the same way as with flow entries, the controller can add, update or delete groups using corresponding OpenFlow commands. A flow entry can use a group id as a valid value in its action field, and flow entries with the same group id in their action field conceptually belong to the same group. The use of groups simplifies the use-case where multiple flow entries share the same behavior.

More concretely, a group contains a list of *action buckets*, and each bucket in turn contains a set of actions. Depending on the type of group (all, select, indirect, or fast-failover), the group applies the actions of either a single bucket or multiple buckets to packets forwarded to the group. The select group, which selects the actions of only a single bucket to apply to a packet, is especially interesting since the OpenFlow specification does not specify the algorithm that selects the bucket. Implementers of the OpenFlow protocol themselves have to decide how to configure an algorithm for selecting buckets.

2.1.2 Controller

Within the OpenFlow specification, a controller is a software program that communicates with OpenFlow-enabled switches using the OpenFlow protocol. In most cases, the controller simply acts as a server application, listening for incoming switch connections over TCP. There are two principal ways in which a controller can add flow entries to the flow table of a switch [12]:

1. Proactively

   At a chosen time, the controller actively pushes out a flow entry to one or several switches. The controller thus itself initiates the decision to add a new flow entry.

2. Reactively

   When a switch receives a packet that does not match any existing flow entry, the switch has two choices: drop it; or forward the packet to the controller. If forwarded, the switch sends the packet to the controller which then makes the decision on how to handle the packet, and whether to add any flow entry to take care of future packets of the same type or not. This allows the controller to set up flow entries only when required.

There are many open-source SDN controller frameworks available for a wide range of languages, such as Nox (C++) and Pox (Python), Ryu (Python), Floodlight (Java), and OpenDaylight/Hydrogen (Java).

All of these SDN controllers offer more or less equivalent base functionality, as they all implement the OpenFlow specification. They differ mainly in which version of OpenFlow they support and the API they expose for writing controller modules. Some controllers implement more high-level services than other, such as topology discovery, link discovery, switch diagnostics, and switch monitoring.

2.2 Performance Management (PM)

Performance management (PM) in the context of computer networks is the overlaying concept of identifying and rectifying performance problems in a network. In a more relaxed terminology, PM refers to the act of measuring network performance and identifying performance issues.

We can divide PM into *active* and *passive* approaches. In passive PM, the switches and routers a network gather information on the performance of the network passively. While this method can theoretically capture any performance related information about a network, the need for direct access to the switches or routers in the network often makes it technically impractical or infeasible to perform.

An active PM technique on the other hand, gathers information by actively sending probe packets into the network. In strict terms, the definition of active PM also covers situations where the switches or routers cooperate by e.g. timestamping packets. Active PM is however more commonly used in situations where only the end-points are accessible. The main drawback of an end-to-end, active PM approach is that it introduces additional traffic overhead in the network, and that the method produces less accurate estimates than a passive approach [13].

While PM can conceptually gather many different types of path performance metrics, the metrics of primary interest in a load-balancing context are capacity and available bandwidth. Dovrolis et al [13] define capacity and available bandwidth, together with some key concepts as follows:

Capacity: The maximum IP-layer throughput a path can provide, given no cross traffic.

Available bandwidth: The maximum IP-layer throughput a path can provide, given the current cross traffic of the path.

Narrow link: The link in a path which limits the capacity of the path.

Tight link: The link in a path which limits the available bandwidth of the path.

Bottleneck link: An umbrella term which can refer to either the narrow link or the tight link in a network path.

Cross traffic: Traffic that competes with or distorts the *significant flows* in a network path.

The distinction between the related network parameters capacity and available bandwidth is important, and there is a difference in how to measure them.

The transmission or serialization delay of the bottleneck link in a path determines the capacity of the path, meaning that the capacity does not relate to the current level of cross traffic. It is the physical bandwidth limit of the link, and the hardware of the router or switch, which imposes a maximum limit on the path capacity [13]. One way of measuring capacity is to measure the dispersion of two back-to-back packets, of a known size, traveling across a path. Given a measured packet-pair dispersion of δ, and a packet size L, $C = L/\delta$ (i.e. the speed-distance-time formula) estimates the capacity of the narrow link in the path. However, the formula is only true in principle, and requires an otherwise empty path to produce consistently accurate capacity estimations [13]. The risk that a packet-pair will experience queue time and increased dispersion due to cross traffic increases with increased link utilization. Tools such as Pathrate [13] instead constructs its capacity estimate from analysis on the distribution of multiple packet dispersion values.

Available bandwidth relates to capacity, but is fundamentally different. This can intuitively be seen from the fact that unless a path suffers from congestion, a packet-pair will pass through the switches in the path unhindered, and the packet dispersion will remain the same at the receiver (a measurement of the capacity of the path). Thus, it is not possible to directly use packet dispersion to measure the available bandwidth of a path. Common approaches instead attempt to measure path congestion, such as through e.g. an adaptive probe traffic rate. By increasing the rate of the probe traffic the measurement methodology induce a congestion in the switches, at which point (in principle) the rate of the probe traffic should equal the current available bandwidth.

2.3 Microwave mobile backhauls

A mobile backhaul is the part of a telecommunications network that connects subnetworks at the edges of the network (access) to the core (or backbone) network by an intermediate aggregation network [14]. The mobile backhaul thereby makes sure that the services at the core of the network can be reached by users connecting through the access. When a mobile backhaul is built up by nodes communicating over microwave links, it is called a microwave mobile backhaul. The dynamic behavior of microwave links have implications for the topology of microwave mobile backhauls.

2.3.1 Topologies

There are a number of basic standard topologies that builds all types of networks, some of which are more common than others when designing microwave mobile backhauls. Figure 1 illustrates these basic topologies.

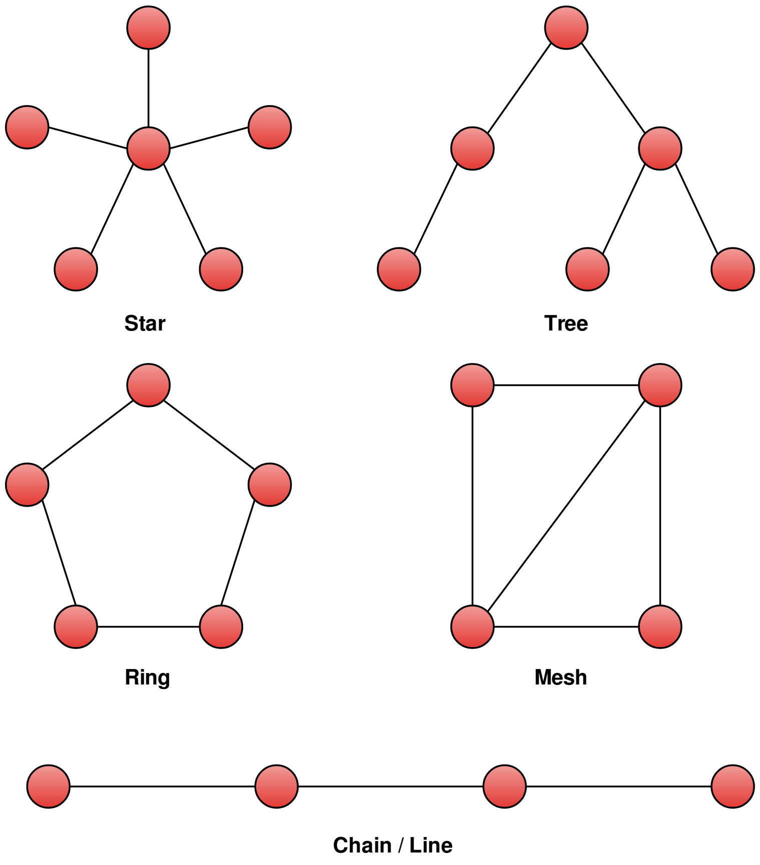

Figure 1: *Basic network topologies that can be used as building blocks for any type of network*

Microwave mobile backhaul topologies are typically built to provide some kind of redundancy, since the risk of microwave link failure (or severe degradation) is inherently more prominent than the risk of failure of for example fiber links [15]. Therefore, as a way of providing an alternative path in the case of a link failure, microwave mobile backhaul topologies commonly use ring or mesh (see example of each type in Figure 1) in the aggregation. The ring or mesh in the aggregation then often connects to trees closer to the edges of the network (the access of the backhaul). Figure 2 shows an example microwave mobile backhaul topology combining mesh and trees in.

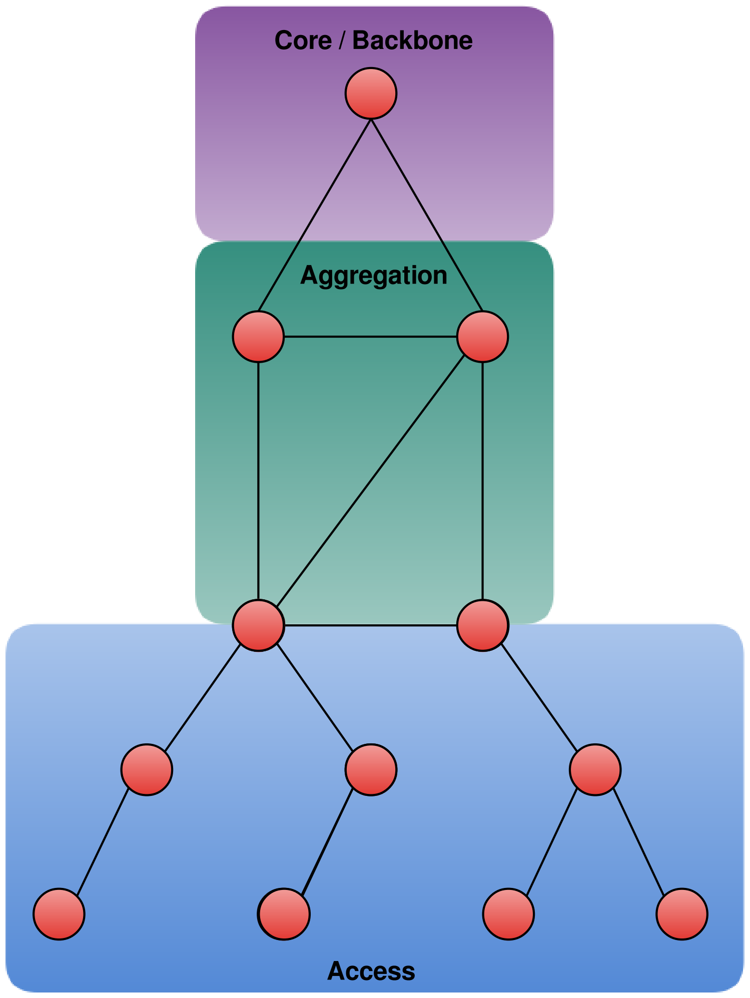

Figure 2: *A microwave mobile backhaul topology built up by trees in the access that connect to a mesh in the aggregation, which in turn connects to the core of the backhaul*

As seen in Figure 2, the aggregation of the microwave mobile backhaul is the central part responsible for carrying all traffic from access to the core of the network. It is therefore also the aggregation of the backhaul that benefits the most from providing some redundancy in the form of alternative paths.

2.3.2 Link behavior

Dynamic effects such as rain and snow affect microwave links. Such dynamic effects degrade the signal strength of microwave links, thereby increasing the Bit Error Rate (BER) and risk of loss [16]. The typical degradations are commonly split up in the categories flat fadings (also known as rain fadings or rain attenuations) and multi-path fadings. Figure 3 shows an example of a rain fading and a multi-path fading as well as their impact on the BER of a microwave link.

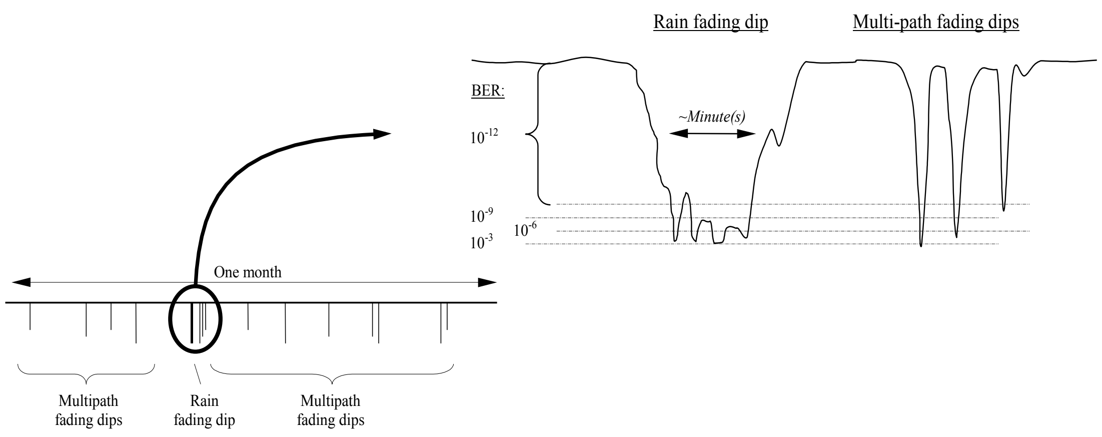

Figure 3: *A rain fading and a multi-path fading increasing the BER of a microwave link*
Source: *Ericsson*

Microwave links commonly make use of adaptive modulation to cope with signal-related degradations such as flat fadings and multi-path fadings [16]. The application of adaptive modulation means that the link adaptively adjusts the current Quadrature Amplitude Modulation (QAM), for example in the case of a rain weather causing flat fading. When a flat fading occurs, the link shifts the modulation to a lower-modulation format (lower QAM) that prevents the affected link from dropping packets, at the cost of reducing the maximum capacity [16]. Figure 4 shows an example of how the modulation can vary over time given a certain signal.

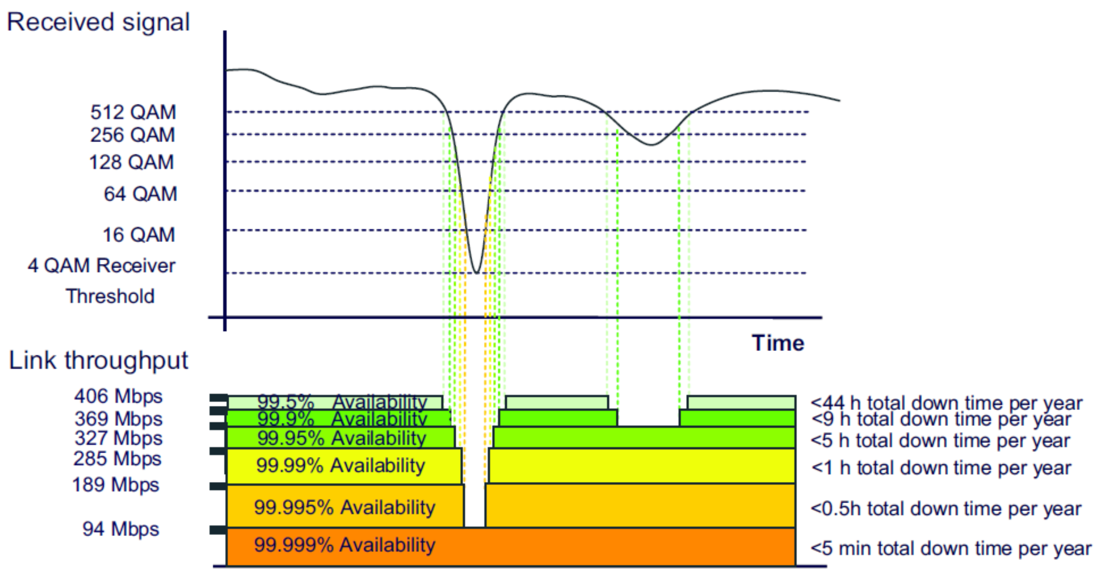

Figure 4: *The effect of adaptive modulation on link throughput given a certain microwave link signal*
Source: *Ericsson*

As seen in Figure 4, the change in modulation affects the capacity (throughput) of the link. Lower QAM results in lower capacity while increasing the availability of the link. A microwave link can thus make use of adaptive modulation to assure a high availability, by compromising on capacity.

2.4 Load-balancing

While load-balancing is a heavily overloaded concept, for the purpose of this thesis, we define it as the distribution of traffic across multiple resources such as servers or network links. The goal of the load-balancing is to optimize some property of the network, such as minimizing response time, maximizing throughput, or maximizing the utilization of the network links.

In a classic use-case, the load-balancer distributes incoming traffic evenly over multiple identical server replicas [17]. In this situation, the load-balancer performs the distribution at the gateway – the router or switch immediately closest to the servers. In this scenario, the processing capacity of each individual server is the limiting factor, and not the network infrastructure itself.

Another use-case is to load-balance traffic between two end-points in a network. This use-case is less common, since it requires a network infrastructure where there exists multiple paths between the two end-points, as well as control over the equipment in the network. The limited resource is the processing capacity of the switches or routers, and the physical capacities of the network links.

In the case of load-balancing between two given end-points, we can define the task of a load-balancer as the distribution of the flows in the network over the available paths, in such a way that we reach the overall goal (e.g. maximum throughput). A flow in this case is a logical construct which represents a sequence of related network packets, which can (or should) not be split up over multiple paths. A typical flow could be a TCP connection, which if split up over multiple paths could cause packets to arrive at the destination out of order.

2.5 Related work

Much research has been done on load-balancing in the context of SDN using OpenFlow [17, 18, 19, 20, 21, 22]. However none of this research has been done in the environment of a mobile backhaul, and there is to our knowledge no previous work on the combination of SDN, load-balancing, and active PM.

Wang et al [17] make use of OpenFlow to load-balance traffic to web servers, all of which host the same replica of a large file and where server processing capacity is the main bottleneck resource. A hash of the source IP-address acts as the basis for the load-balancing, and the last switch distributes the traffic between the servers. Wang et al also describe in detail the problem of transitioning to a new traffic partitioning, and they effectively use the TCP SYN flag to send traffic of the same TCP connection to the same server replica, avoiding packet reordering.

The main limitations of the work is that Wang et al assume that the client IP-addresses follow a uniform distribution across the whole IP space (e.g. not clustered to certain countries or ISPs), and that traffic distributes uniformly across IP-addresses.

Most research, such as Wang et al [17], study the load-balancing of a dynamic level of traffic and a dynamic number of clients over multiple web servers [17, 18, 21, 22], which makes it natural to partition traffic at the IP-address level. This partitioning does, however, present problems in a mobile backhaul where traffic may e.g. only be routed over IP between end-points within the backhaul, and not between end-to-end hosts (mobile device to destination server). Therefore, there is no guarantee that there exists a uniform distribution of source IP-addresses, or even more than a single IP-address if traffic is tunnelled.

Many of the OpenFlow-based load-balancing algorithms such as LABERIO [19] and LOBUS [18] make the assumption that the capacity and available bandwidth of all links is known. They all make this assumption since the algorithms are designed for a scenario where the capacity of the network is more or less static, while the traffic is the dynamic factor. The scenario of load-balancing in a network where the bandwidths of the links themselves fluctuate seems to be a rather neglected problem, most likely since the situation in itself is rare. Microwave link networks could be one of few environments where link bandwidth fluctuations occurs frequently enough to be worth solving.

When it comes to PM, there exists numerous work on using active PM to measure the end-to-end capacity and available bandwidth in a network [13, 23, 24, 25, 26, 27, 28].

The literature clearly shows that packet dispersion is the fundamental property to use for measuring capacity and available bandwidth, and much work focuses on finding out how different packet dispersion characteristics relate to the capacity and available bandwidth in a network. Packet-pairs, as well as the generalized form packet-trains, are common approaches to measure capacity. Jacobson [27] was the first to describe the concept of packet dispersion, which has now been a field of research for over 20 years [13].

Strauss et al [29] separate the available tools and estimation techniques of available bandwidth into two distinct approaches:

- **Probe Gap Model (PGM)**

  TThe basic packet-pair approach compares the difference between the time gap at the sender δ_{in}, and the time gap at the receiver δ_{out} between two successive packets. The measured dispersion then estimates, after varying levels of statistical analysis, the available bandwidth of the path. Tools such as Pathrate [13] and Spruce [29] make use of this approach.

- **The probe rate model (PRM)**

  This approach relies on an adaptive rate of probe traffic. The basic idea is that when probe traffic is sent at a rate below the available bandwidth, the dispersion of the packets at the sender will remain the same at the receiver. Conversely, if the rate is above the available bandwidth, the switches will queue the packets, and the receiver will observe an increase in packet dispersion. By adapting the rate of the probe traffic, a bandwidth estimation tool can find the turning point at which the dispersion starts to increase. The probe rate at this point then corresponds to the available bandwidth. Tools such as Pathload [23], pathChirp [25], ASSOLO [26], and BART [28] all use this approach.

The PRM approach has an inherently intrusive behavior, given that it often sends probe traffic at a rate higher than the available bandwidth, thus inducing a congestion in the switches (i.e. an artificial burst of traffic). Whether or not the *self-induced congestion* is large enough to cause any noticeable network performance issues remains unclear. Simulations show that e.g. Pathload does not seem to cause any persistent increase in the queue sizes of the network [23].

In the literature, there is some confusion as to what exactly packet dispersion measures. Early work assume that the dispersion of long packet trains is inversely proportional to the available bandwidth, but Dovrolis et al [13] show that this is not the case [29]. Instead, packet dispersion measures a property called Asymptotic Dispersion Rate (ADR). The property relates to, but is not the same as, available bandwidth. ADR determines the lower bound of the capacity and the upper bound of the available bandwidth [13].

Dovrolis et al further show that the statistical distribution of packet dispersion measurements is multimodal, and that the capacity of the path is often not the global mode. Thus, to accurately estimate the capacity, one has to select the packet dispersion value that occurs frequently, but not necessarily most frequently. The challenging task is therefore to identify which local mode represents the capacity of the path. Specifically, the results from Dovrolis et al suggest that earlier tools which simply select the average, median, or mode value from the distribution of packet dispersions will produce inaccurate capacity and bandwidth estimations.

3 Design

To simulate a microwave mobile backhaul and to perform active PM together with dynamic load-balancing, we need a number of different components to form a complete solution. To get an overview of the solution as a whole, we first describe the overall design on a conceptual level. This is followed by a detailed description of each of the main components, where we explain its design together with its respective limitations.

3.1 Concept

The traditional way of using static rules to decide how to balance traffic fits poorly in a microwave mobile backhaul, given the non-deterministic behavior of microwave links. This thesis therefore discusses an alternative design capable of performing dynamic load-balancing based on active performance measurements. The idea is to actively measure the performance along different paths in a microwave mobile backhaul and to use those performance indicators to trigger load-balancing decisions. To be able to apply such decisions, the switches in the microwave mobile backhaul must be continuously updated with new switching rules. We accomplish these continuous updates of switching rules by using SDN and a centralized SDN controller.

As a way of reducing complexity and to emphasize on separation of concerns, we divide the solution into a number of different components. Figure 5 shows an overview of the overall design including all the components that together form a complete solution. In Section *3.2 Components* we describe the design of each individual component.

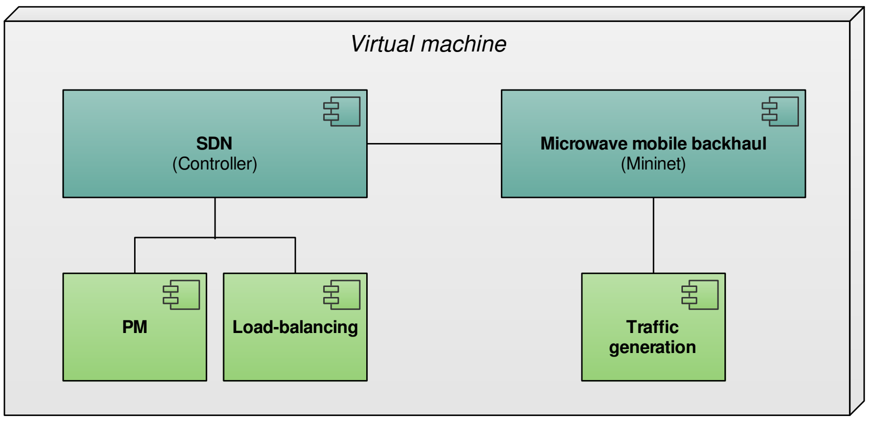

Figure 5: *Overall design described in terms of components*

In order to gain some intuition about how the components in Figure 5 should collaborate, let us first consider a minimal example microwave mobile backhaul topology (see Figure 6). Figure 6 describes how a network connects host `h1`, representing end-users, to another host `h2`, representing a core switching site. The main consideration here is that there are two alternative paths available in the microwave mobile backhaul that can carry user traffic.

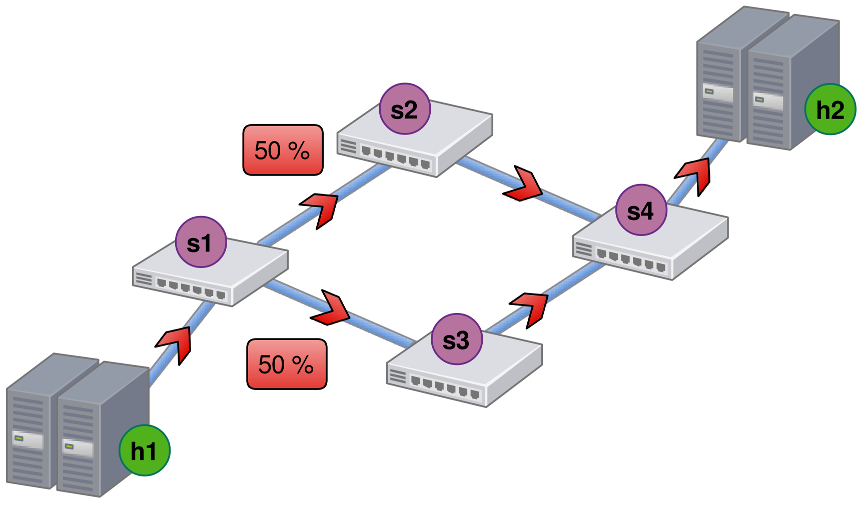

Figure 6: *Example microwave mobile backhaul topology with 50/50 traffic split*

Given that all links have the same capacity, a normal network configuration splits the traffic evenly across the paths, as seen in Figure 6. However, upon fluctuations in bandwidth on certain microwave links, it might be preferable or even necessary to balance the traffic over the different paths (as can be seen in Figure 7) in order to avoid packet queues building up at some switches, eventually causing packet drops.

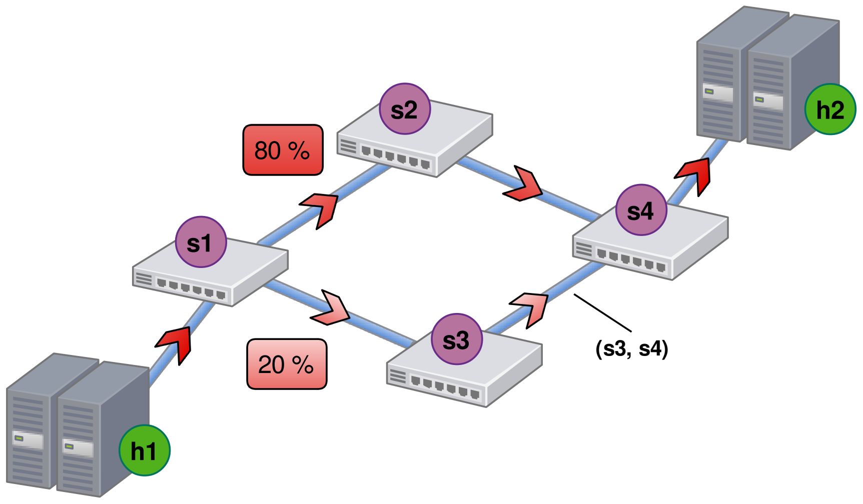

Figure 7: *Example microwave mobile backhaul topology with 80/20 traffic split, responding to a capacity degradation of link (s3, s4)*

Figure 7 describes how a bandwidth fluctuation on link `(s3, s4)` has been coped with by a new distribution of traffic, sending 80% of the traffic on one path and 20% on the other. The goal is to balance the load in the microwave mobile backhaul in a way that minimizes the overall packet loss level, thereby optimizing the utilization of the microwave mobile backhaul. One way of triggering such load-balancing decisions is by using performance indicators from active PM, which is the approach we take in this thesis. The following high-level steps provide a description of the overall design:

INITIALIZATION:

- Start simulation of a microwave mobile backhaul
- Identify all paths in the network between the start node (host representing end-users) and the end node (host representing the core switching site)
- Start an active PM flow for each path
- Generate user traffic
- Start simulation of dynamic effects on microwave links

REPEAT (in intervals):

- Compute weight of each path using metrics from active PM flows
- Split traffic over paths using computed weights

From this quite general approach, we observe that it should be possible to apply the same method of performing PM and load-balancing also in other types of networks. Since this thesis discusses the domain of microwave mobile backhauls, where we need to run a simulation to evaluate the solution, the approach does however include a few additional steps in the initialization process to start the simulation as well as the traffic generation.

The idea is that once we start a simulation and get active PM up and running on all paths, we can recalculate load-balancing weights at fixed time intervals based on the active PM measurements we receive during each time interval. In this way, when for example dynamic effects on a certain microwave link occurs, measurements on the affected paths should reflect the changed conditions and eventually lead to a redistribution of traffic after recalculation of the load-balancing weights. In this way, we can continuously balance traffic on the different paths in the microwave mobile backhaul to cope with dynamic effects on microwave links causing reduced capacities on certain paths.

3.2 Components

In this section we describe the design of each component while outlining the choices we make as well as stating relevant limitations.

3.2.1 Microwave mobile backhaul

Essential to the simulation is the modeling of the network itself, namely the microwave mobile backhaul. When deciding upon how to model the network, the two following criteria needs to be fulfilled:

1. Alternative paths

   In order to be able to perform load-balancing there must be at least some alternative paths through the network.

2. Links with fluctuating bandwidth

   Dynamic effects such as rain and snow cause bandwidth fluctuations on microwave links. These dynamic effects, called fadings, should be incorporated in the microwave link model.

The criteria outline the two main parts that we need to model a microwave mobile backhaul; topology, and link model.

3.2.1.1 Topology

We design the topology to resemble an actual microwave mobile backhaul, but resource limitations in performing the simulation impose a restriction on the number of nodes (switches) we can use in the topology. The presence of alternative paths in the topology is a prerequisite, which limits the options for choosing a topology to ring and mesh, or combinations of any topologies involving at least one of the two.

Since we need to keep the number of nodes low, we abstract away the end-points of the microwave mobile backhaul. Instead, we connect end users to the core of the microwave mobile backhaul by substituting them with a single node (host) representing all connected end users. This is a reasonable simplification to make, since the end-points of microwave mobile backhauls often connect to the core backhaul using trees, where no alternative paths are available anyway. The host is the node which should generate user traffic, which the network then routes through the microwave mobile backhaul to the other end-point, representing a core switching site connecting the microwave mobile backhaul to a fiber network. We also model the core switching site as a single host, which is responsible for receiving all traffic.

Given the representation of the end-points as two single hosts, the topology we choose for the simulation is mostly concerned with the core of a microwave mobile backhaul where ring and mesh topologies are common. We need to connect the two hosts in a way that keeps the total number of nodes low while creating multiple paths between the two end-points by adopting a ring and/or mesh topology. Therefore, one of many suitable options is a simple ring topology connecting the two hosts with extra cross-feed links to provide some redundancy, thereby creating a ring/mesh topology with several alternative paths according to Figure 8.

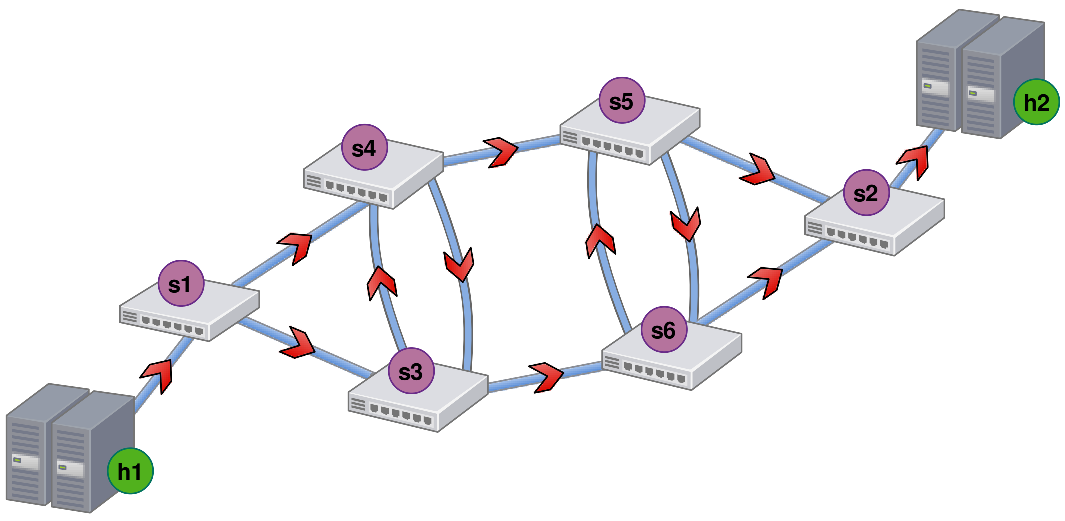

Figure 8: *The microwave mobile backhaul topology, with 8 possible paths between hosts h1 and h2*

Figure 8 shows that there are 8 possible paths that a packet can take from `h1` to `h2`. Thus, the PM component described in Section *3.2.3 PM* shoud measure the performance along these 8 paths, while the load-balancing component described in Section *3.2.4 Load-balancing* distributes the traffic across the same 8 paths.

We model the topology in mininet, since it provides a good API and solid infrastructure capable of running different software switches and enables seamless connection to an SDN Controller. Ideally, we should evaluate the solution using several different topologies. However, as a way of limiting scope this thesis only evaluates the solution using a single topology.

3.2.1.2 Link model

Different types of dynamic effects affect microwave links in microwave mobile backhauls. One type of dynamic effects that can severely degrade microwave links for considerable time intervals are weather effects, such as for example rain and snow. Two common patterns seen on microwave links when affected by such effects are flat fadings and multi-path fadings, which both lead to capacity degradations of the links. We use the physical link properties described in Appendix *A Physical properties of microwave links* as a starting point for modeling of microwave links in this thesis, but it should be noted that microwave link characteristics are heavily dependant on local factors such as climate. The limited statistics in Appendix *A* is only concerned with flat fadings and multi-path fadings. Therefore, we limit the modeling of microwave links in this work to only account for those two types of fadings, making it concerned only with the general characteristics of microwaves.

Another substantial limitation in the modeling of the microwave links in this work is the lack of relations between microwave links. It is reasonable to assume that for example rain that affects a certain microwave link significantly increases the probability for flat fadings on surrounding microwave links. However, we choose to model each microwave link independently, and the reason for this is both lack of data and the level of complexity. More sophisticated microwave modeling is a topic of its own, beyond the scope of this thesis.

Flat fadings and multi-path fadings, which essentially are dips in the signal strength of a microwave link, degrading its capacity, have different characteristics that we need to model accordingly. One approach to model fadings is to describe them as functions of capacity over time by using known values for the capacity at specific times. This approach would however require a large dataset of fadings in order to be able to capture the diversity in e.g. duration and amplitude, but such a dataset is however not available. Instead, we model the two types of fadings using mathematical functions, making it possible to vary key characteristics such as duration and magnitude of the dips etc.

The challenge is thus to find suitable mathematical functions to approximate flat fading and multi-path fading. For the purpose of this thesis, our primary concern is to capture the main characteristics, thereby allowing us to react to changes in capacity. Simplified, we can choose to view a flat fading as a single dip that can vary in duration and magnitude. In the same way, we can view a multipath fading as three consecutive flat fadings, only shorter in duration.

Therefore, we can use cosine as a starting point for the mathematical model of a dip. A flat fading maps well to a single cosine cycle, thus making cosine a suitable mathematical function for approximation of flat fadings. In the same way, we can approximate a multi-path fading using three consecutive cosine cycles, equivalent to three consecutive single dips.

A big advantage of modeling fadings using cosine is the flexibility of being able to alternate the duration of the fading by simply changing the period, and vary its amplitude by multiplying the cosine function with a constant that we assign different values. This means that we can chain together and randomize fadings to form almost any distribution.

Given the modeling of fadings, we model a microwave link as a function of capacity over time, where at each time step, we pick one of the following functions with a preset probability:

- Flat rate

  The normal capacity of the microwave link

- Flat fading

  A single cosine cycle with randomized duration and amplitude within preset intervals

- Multi-path fading

  Three consecutive and equal flat fadings with randomized duration and amplitude within preset intervals

The three events are mutually exclusive, meaning that different types of fadings cannot happen simultaneously. This means that we do not actually choose one of these three types of events (flat rate, flat fading, and multi-path fading) at each step of the simulation, but rather at each step in the simulation when the previous event has just ended.

The available data about frequencies of occurrence for the different types of fading is very limited as well as detailed information about typical duration. Multipath fadings are relatively frequent as compared to flat fadings, and the duration of a multipath fading is typically in the range of seconds while flat fadings can last for minutes (see Appendix *A Physical properties of microwave links*). Therefore, we choose the following best effort numbers for duration and frequency of occurrence:

- Multipath-fading
  - Duration: 2 - 4 s per dip, 6 - 12 s in total
  - Frequency of occurrence: 15%
- Flat fading
  - Duration: 1 - 20 min
  - Frequency of occurrence: 5%

When it comes to amplitude, we need to translate the degradation in signal strength to capacity, which depends on how the microwave links apply adaptive modulation. The highest modulation provides an upper limit for the capacity (maximum or normal capacity). Then there is a minimum capacity resulting from the use of the lowest modulation. We thus randomize the amplitude of the simulated flat fadings and multi-path fadings between those two limits. The specific values for maximum and minimum capacities that we use in the simulation is provided in Chapter *5 Evaluation.*

We model each microwave link independently using the same model. In order to get different randomized behavior for every link, we use different seeds. Since we initialize each link with a specific seed, we get the possibility to re-run simulations and get the exact same link behavior as in the previous run. This is necessary to be able to compare different load-balancing solutions using identical link modeling in subsequent simulations.

Despite the limitations and the lack of accurate parameters when it comes to fading specifics, our microwave link model still captures the essence of the dynamic behavior and fluctuation of bandwidth. The primary purpose of this thesis is not to model microwaves, but rather to find a minimum viable model which we can use for the purpose of performing PM and load-balancing in the presence of dynamically changing link capacities (the context of a microwave mobile backhaul).

3.2.2 SDN

In this thesis, we use PM and load-balancing specifically in the context of SDN, meaning that an infrastructure with a central entity capable of controlling a set of switches is necessary. We design the SDN infrastructure, here referred to as SDN component, using a SDN Controller and accompanying software switches.

3.2.2.1 SDN Controller

In this work, the SDN controller functions as a layer on top of the simulated microwave mobile backhaul, connecting to the nodes (software switches) and providing them with flow rules. The SDN Controller runs in two different modes:

1. Static load-balancing

   In this default setup, the SDN controller splits all traffic evenly on all paths, meaning that the switches distribute all incoming traffic over all outgoing links in a way that leads to all paths carrying the same proportion of traffic. We will refer to this behavior as static load-balancing from now on. Static load-balancing only requires the SDN Controller to provide the switches with flow rules in the startup phase of the simulation.

2. Dynamic load-balancing

   In this setup, which uses a PM component, the SDN Controller continuously provides the switches with updated flow rules based on load-balancing weights calculated from PM metrics for each path.

Note that we could as well achieve static load-balancing without the use of a SDN Controller. The reason for choosing to configure a SDN Controller to perform static load-balancing is to make the evaluation of the dynamic load-balancing solution more straightforward and intuitive by limiting the number of external comparison factors.

When it comes to SDN Controllers, there are several open-source projects available, as well as many commercial products. In order to minimize the risk of encountering problems when setting up an SDN infrastructure, we choose an open-source option that is well-documented and has a satisfactory user base. We should point out that the performance of the SDN Controller itself is not crucial, since we evaluate the final solution against an identical setup using the same SDN infrastructure (only without PM and dynamic load-balancing). We can therefore eliminate the performance factor of the SDN Controller itself from the equation.

Among the open-source projects, Floodlight [7], OpenDaylight [30], and NOX/POX

[31] are well-known SDN Controllers. NOX/POX has poor documentation and OpenDaylight is in active development, making it hard to choose a stable and suitable release. We consider Floodlight to be well-documented, and it has a large enough user base as well as an available release (0.90) which has been tested for a while. Therefore, we consider Floodlight to be a suitable SDN Controller for this work.

The Floodlight SDN Controller is the main component of the load-balancing solution, and in Chapter *4 Implementation* we describe how we add PM and load-balancing capabilities as Floodlight modules.

3.2.2.2 Software switch

In order to be able to balance traffic between different paths in the simulated network, the centralized SDN controller needs to be able to communicate with the nodes and provide them with flow rules. In the simulated microwave mobile backhaul, the nodes are software switches communicating with the SDN controller using the OpenFlow protocol.

Given our choice of the open source SDN controller Floodlight, the accompanying software switches needs to be OpenFlow compatible. Open vSwitch [5] is the most commonly used software switch with OpenFlow support. While there exists other alternatives, Open vSwitch has a substantial user base, good documentation and a large amount of other available resources. Another important feature is that it supports the use of different versions of the OpenFlow protocol simultaneously [32].

Although Open vSwitch only has full support for OpenFlow 1.0, it has partial support for newer versions [33]. Therefore, the ability to interchangeably use different versions of the OpenFlow protocol makes it possible to use the stable OpenFlow 1.0 protocol as a starting point while still being able to send commands introduced in later versions of the protocol to the switches. We find this possibility highly desirable, since Open vSwitch has support for groups, which are part of OpenFlow version 1.1 and later. In Section *3.2.4 Load-balancing* we further discuss how a load-balancer can use groups to partition and distribute traffic over paths in a network.

Floodlight, as well as many other comparable SDN Controllers, only has support for OpenFlow 1.0 [34]. This limits the communication between Floodlight and Open vSwitch to the OpenFlow 1.0 protocol, which excludes any use of groups. The use of groups therefore either requires modification of the SDN Controller, or some other separate way of sending group commands to the switches. The latter approach is more lightweight than the former (which would require a lot of work) and is therefore the method we choose for utilizing groups.

3.2.3 PM

In order to be able to take load-balancing decisions, a load-balancer requires information on the throughput of the network. End-to-end latency is one of the key PM metrics of interest, and the PM component can measure it using any of the two main approaches:

1. Measure on path

   In measuring traffic on path, a start node sends traffic to an end node, timestamping each packet. The packets pass by the switches in the network with no special action, and on the other end the receiver calculates the total end-to-end delay of a packet using the provided timestamp.

2. Measure on link

   Similar to measure on path but in addition each switch in the path also timestamps each packet, allowing the receiver to calculate the transmission delay for each link in the path. While this provides more information, it also requires that every switch in the network is aware of the PM stream and treats it specially. This can raise some concerns regarding how well the test traffic captures the latency characteristics of the user traffic, as it is being specially treated by the switch. Note also that this method does not necessarily capture any additional useful information, as a path end-to-end latency includes the latency of each individual link.

In the search for a suitable performance measurement method we recognize the following requirements:

1. Accuracy

   The quality of the performance estimates is the main limiting factor in the effectiveness of the overall load-balancing, and thus a high accuracy is important.

2. Non-intrusiveness

   The PM method should have no or negligible impact on the available bandwidth, end-to-end latency, or packet loss in the network. Microwave links are typically used at the edges of networks over greater distances where higher speed connections are too costly to maintain. Since active PM occupies a constant fraction of the total bandwidth, the need for a lightweight PM method is especially substantial in a mobile backhaul network.

3. Speed

   How quickly the PM method can detect changes in link capacity determines which types of link degradations we can handle.

A major obstacle is that most existing bandwidth estimation techniques are not designed for fast and continuous performance measurement. Several of the techniques such as Pathload [23], Spruce [29] and Yaz [24] are too slow, requiring up to 10 seconds to produce a bandwidth value, and Pathload in some cases up to 20 seconds [35].

Traffic overhead of active PM is a large problem. Even if we consider a conservative estimate of 500 kB per measurement [35], the total overhead of measuring 8 paths, e.g. every 2 seconds reaches up to 16 MBit/s. In a microwave mobile backhaul where the gross bit rate of a typical microwave link is < 100 Mbit/s depending on frequency band and modulation used [16], test traffic could occupy a significant fraction of the bandwidth. This also renders detection of variations in bandwidth shorter than a second practically infeasible as the test traffic required would approach 50-100% of the link capacity. In light of these challenges, we call for a simpler and much more lightweight PM technique.

We first observe that tools such as Pathload and ASSOLO [26] are by design bandwidth estimation tools, where it is important to receive an absolute value of bandwidth that can later be compared to other measurements performed at different times or locations. In contrast, an isolated system such as a load-balancer does not inherently require absolute values, as the load-balancer only compares path performance in relation to other paths in the same network, and at the same point in time.

We propose a design where a constant stream of empty packets (essentially UDP headers), at a fixed and configurable low rate, probes the level of congestion in the network by observing how end-to-end latencies change over time. The end-to-end latency estimates the congestion in the packet queues of the switches in a relatively crude but good way.

In this approach the PM component starts a test flow for each path between the start and end node in the network, effectively providing a continuous measurement of the end-to-end delay of all paths. Special flow rules are used (differentiated by UDP port) to guide each test flow through its respective path in the network. Packets leaving the start node are timestamped and the end node computes end-to-end delay of the packet. At a configurable time interval of 1.5 - 5 s, the PM component collects and analyzes all received measurements. The analysis of the end-to-end latency of a path can then be made as simple or complex as the situation requires.

One key assumption of this approach, and any other approach which compares timestamps, is the existence of a synchronized clock, which is natural in a simulated environment but less so in a physical network. The problem of clock synchronization is however limited; only the start and end node of the network require a synchronized clock.

3.2.4 Load-balancing

Within the boundaries of the OpenFlow specification and the previously described network model, we can define a load-balancing solution as the distribution of flows over the available paths in the network. A load-balancer then has three independent problems to solve:

1. Partitioning of incoming network traffic into flows
2. Finding an optimal load-balancing solution
3. Transitioning between load-balancing solutions (responding to changes in link capacities)

3.2.4.1 Partitioning of incoming network traffic into flows

Partitioning network traffic into flows is in itself a large and rather complex problem, where the approach highly depends on the specifics of the network environment. We have intentionally kept this part of the overall problem as simple as possible, while making sure it does not limit the effectiveness of the load-balancing or the validity of the conclusions.

Firstly, a high number of flows is desirable. Since the load-balancer assigns each flow to only one path, the amount of flows directly determines the granularity of a load-balancing solution. To explain this further, let us consider a network with only two flows. In such network, each path is limited to accepting 0%, 50%, or 100% of the traffic. In this situation, the amount of possible load-balancing states is very limited.

Secondly, traffic should be split evenly among flows, making sure that each flow represents roughly the same amount of traffic. If a situation arises where a single flow represents the majority of the traffic, the load-balancing may be uneven and cause packet drops. This can be extremely challenging in a dynamic network, with varying levels of traffic and varying number of transport level protocol (TCP, UDP) connections.

Finally, it is preferable to find a partitioning scheme that minimizes the amount of control traffic required to create and maintain the flow partitioning. Unless the load-balancer and switch communicate over a separate channel, the control traffic directly competes with the user traffic.

Based on the criteria of a desirable partitioning scheme, we use a partitioning scheme that we refer to as *stochastic switching* [36] throughout the rest of the thesis. In this scheme the switch randomly selects which port to output an incoming packet on, but weights the selection to achieve the traffic split set by the load balancer. This method relies on the group type *select*, which is a valid action of a flow rule and an optional feature in OpenFlow 1.1 and later versions of the protocol. The group, which is essentially a list of actions, selects which action (in our case output port) to apply to each incoming packet. The OpenFlow specification does not define how this selection is performed, or on what basis, but specifies that the selection algorithm can optionally be based on the action weights, which the SDN controller sets when creating or modifying the group [11].

Open vSwitch does not supply a way to plug-in custom behavior at this level, but through modification of the source code, we can replace the selection algorithm in order to achieve a stochastic partitioning scheme. In its most basic implementation, the switch invokes a PRNG (Pseudo-Random Number Generator) for every incoming packet, and picks an output port based on the result. This however requires that each packet passes through the software layer of the switch, which greatly limits the maximum obtainable network speed. For the purpose of this thesis however, the simulations do not require e.g. gigabit network speeds. In a real-world scenario, and for greater switching performance, one can use a weighted round-robin algorithm [37] to achieve the same effect.

The result is a packet-level traffic split of maximum granularity, where the load-balancer regards each packet as a separate flow. The granularity, together with the great simplicity, is the main advantage of this design. The design keeps flow rules to a minimum (one per input port, which forwards the packet to a select group) and eliminates all logic around creating and maintaining flows in the SDN controller.

Note however that the way the design achieves a maximum granularity partitioning has a considerable drawback, which however is acceptable for the purpose of this thesis. Since the output port, and consequently the network path, of each packet is randomized independently, the design can not guarantee the preservation of packet ordering. This is a factor that can largely influence the choice of a suitable partitioning scheme. A partitioning that e.g. splits a TCP connection into multiple flows (which the load-balancer then may assign different paths) runs the risk of causing packets to end up at the destination out of order, triggering TCP retransmissions and reducing the overall throughput of the network. However, there are algorithms such as FLARE [38] which attempt to mitigate this problem.

3.2.4.2 Finding an optimal load-balancing solution

One immediate approach is to utilize one of the many existing capacity estimation tools to estimate the capacity of each path in the network. Given the capacity of each path, we can derive the minimum capacity of all links in the network.

This makes it possible for us to treat the network as a single-source, single-sink flow network, and directly translate the problem of finding an optimal load-balancing solution can into the *maximum flow problem*. This is a problem for which there exists several algorithms guaranteed to find a solution in polynomial time, such as Edmonds-Karp algorithm [39].

Furthermore, with the stochastic traffic partitioning scheme we use (providing maximum granularity), the effectiveness of the load-balancing depends (in theory) only on the accuracy of the performance values which the PM component supplies. The overall thesis problem could thus, in some sense, be reduced to finding the best performance measurement method.

However, this approach has several weaknesses:

1. To accurately estimate path capacity from end-to-end packet dispersions is a complex problem. Although there exists a correlation between packet dispersion and path capacity, it is hard to correctly estimate the capacity in the presence of cross-traffic [13]. As we have already seen, there are many tools that attempt to solve this problem with varying degrees of success.

2. Using a max-flow algorithm as a basis of a load-balancing solution makes the load-balancer quite sensitive to errors in the link capacity estimates. Since traditional max-flow algorithms offer no degree of uncertainty in their output, any kind of error in the measurement will likely result in an unstable load-balancing state.

3. A max-flow algorithm does not necessarily distribute the residual capacity of the network evenly among links. To exemplify: given two paths, each with a capacity of 20 Mbit/s and a traffic level of 20 Mbit/s, a perfectly viable solution to a max-flow algorithm would be to send all the traffic over one path, leaving the other path at zero traffic. While correct, this also means that any increase in the traffic level or decrease in link capacity will cause congestion and trigger load-balancing actions, thus making it a highly unstable load-balancing state. A more sensible solution would be to split the traffic evently, 10 Mbit/s over each path, which would leave a buffer should something change. Traffic should generally never be sent on a path at 100% capacity of the path unless it is the bottleneck of the network.

To counter these weaknesses, another approach is to find the load-balancing state that minimizes the congestion in the network. The goal is to maximize the minimum residual capacity of the links. With the terminology we use in this work, we can describe the problem as follows:

```
Find a traffic split which maximizes the minimum residual capacity of
the links, but with the additional condition that it must support the
current level of traffic.
```

This is a NP-hard problem known as the Congestion Minimization Problem (CMB) [40]. It is however possible to approximate a solution, which has been shown to be $\Omega(log(log(n))$-hard [40].

Proposed design

The load-balancing algorithm we propose is a congestion minimization algorithm which iteratively works to minimize the congestion in the network. First, the algorithm normalizes (relative to the sum) the performance values (i.e. latencies) from the PM component, and treats them as estimates of the relative congestion of the paths. To exemplify, consider four paths with relative congestion values of 0.1, 0.1, 0.1, and 0.7. These values indicate that one of the paths is seriously congested and that traffic needs to be redistributed. Note however that these congestion values in isolation say nothing about what the overall traffic split should look like – they only tell us where the current congestion is located.

First, we can observe that the cause of the current level of congestion is the current (imbalanced) distribution of traffic. The algorithm thus calculates the new traffic distribution by multiplying the inverted congestion values with the current distribution of traffic, which it after some necessary normalization uses as the new traffic distribution. The intuition is that this traffic distribution should reduce the overall congestion when the algorithm assigns it to the network.

The algorithm then repeats the above step at a fixed interval, and each new load-balancing state slightly adjusts the traffic distribution, evening out the variations in congestion among the paths. The algorithm thus iteratively attempts to reach a load-balancing state where all paths have the same performance value – i.e. an equal level of congestion.

The result is a load-balancing algorithm that approximates a minimum congestion traffic split. Given that it maximizes the minimum residual capacity, this algorithm not only achieves a traffic split which holds all traffic, but which also is more resilient to dynamic changes in link bandwidth.

Interesting to note is that a max-flow problem can also be extended to incorporate the concept of minimum congestion. The catch however is that this extended problem is NP-complete. Finding an optimal solution is numerically intractable even for small networks, although heuristic algorithms exist [41].

On an intuitive level, a min-congestion algorithm seems like a better fit for this specific load-balancing problem. The main dynamic variable is link capacity, which a min-congestion algorithm handles best. A max-flow algorithm would be better suited in an environment with static link capacities but a dynamic level of traffic.

3.2.4.3 Transitioning between load-balancing states

While we have not given transitioning between load-balancing states a great part of this thesis, it is nevertheless an aspect to consider when designing a load-balancing algorithm.

The task of a load-balancer is not only to calculate the traffic split to use, but also to in a controlled way introduce these changes to the network. All performance measurement methods have some margin of error, and a load-balancer needs to take this into consideration. Inaccurate estimation (overestimation or underestimation) of the available bandwidth can easily cause load-balancing instability. The result can for example be that the load-balancer assigns a too high level of traffic to a link, causing congestion. The congestion then triggers a new load-balancing action to correct the first action. If this new action is also based on an overestimated bandwidth value what follows is an perpetual sequence of load-balancing actions, where traffic is lost in the transitions. This problem becomes more common as the traffic level approaches the maximum throughput of the network, where the margin of error is smaller.

A min-congestion load-balancing solution effectively reduces the risk of reaching an unstable load-balancing state. By maximizing the residual capacities in the links, the tolerance of error is higher.

3.2.5 Traffic generation

To simulate a microwave mobile backhaul we require some kind of user traffic. In this thesis we represent user traffic as generated datagrams, in varying sizes and with different frequencies of occurrence. The traffic originates from the end-point of the microwave mobile backhaul network which represents the end users, and the network then carries the packets to the other end-point, representing the core of the network.

One important criteria for the traffic generation is that it should resemble actual user traffic typically seen in microwave mobile backhauls. Another criteria is that the generation of traffic should be based on publicly available statistics, as a way of keeping the work transparent and reproducible.

In the domain of microwave mobile backhauls, operators such as cell phone carriers are typically reluctant to disclose any of their user traffic statistics, which makes basing traffic generation on such statistics inappropriate for this thesis. Instead, we generate traffic according to simple IMIX (Internet Mix) [8]. Table 1 describes Simple IMIX, which bases its statistical distribution of common packet sizes on sampling done on Internet devices.

Table 1: *Simple IMIX*

Packet size (bytes)	Proportion of total	Distribution (packets)	Distribution (bytes)
40	7 parts	58.333333 %	6.856024 %
576	4 parts	33.333333 %	56.415279 %
1500	1 parts	8.333333 %	36.728697 %

Simple IMIX is commonly used for testing various network equipment and is stated to have a correlation value of 0.892 when compared to realistic internet traffic [8]. Complete IMIX and Accurate IMIX, which have correlation values approaching 1, include one or several extra categories and randomization of packet lengths in addition to the different categories (see Table 1). For this thesis however, the accuracy of Simple IMIX is acceptable, and enables a clean and simple implementation.

The generated traffic is UDP, which is inelastic traffic with the great benefit of a simple and predictable behavior. Elastic traffic such as TCP, which responds to dynamic changes in delay and throughput, might interfere or compete with the load-balancer in optimizing throughput. While one could argue that TCP traffic is more realistic, since we are mainly interested in the effectiveness of the load-balancer, elastic traffic would at this stage only add unnecessary complexity.

4 Implementation

This chapter describes each component on a more concrete level based on its design. To keep it more general and easier to follow, we present pseudocode instead of the actual implementations. First, in order to improve the understanding of the various components and their interaction with each other, we begin by providing a general overview of the implementation.

4.1 Overview

We run all the components of the simulation inside a Ubuntu 13.04 virtual machine. This setup is only used for practical purposes since it simplifies the various configuration required. The network is simulated using Mininet [6], and a Python script configures and runs the simulation using the Mininet python API. When we run Mininet it creates a set of virtual network interfaces that simulates switches and hosts, which are managed (e.g. bandwidth throttled) through the Linux traffic control (`tc`) utility. We can then interact with these virtual network interfaces just like any hardware network interface, i.e. through the `send` Linux system call.

On the same virtual machine, we run the SDN controller which is essentially a server application listening for incoming switch connections on port 6653. The switches are configured through Mininet to connect to the controller over TCP at localhost on port 6653 and initiate the OpenFlow communication between switch and controller. The switches will begin by sending the `OFTP_HELLO` message, which prompts the controller to add the switch to its internal topology of the network.

The initialization of the simulation is relatively complicated, since order matters. The startup of the different components require some careful timing to get the simulation up and running correctly. When the virtual network starts, the switches connect to the SDN controller one-by-one. However, in this transitionary state, the network is unable to route any traffic, as the controller has not yet set up any flow rules. This does not apply to the OpenFlow control traffic, as the controller-switch communication takes place outside the virtual network. Once all switches have connected, the controller pushes out the following two flow rule configurations:

1. Static flow rules for UDP traffic on port 20000-20008 (the PM flows)
2. A 50/50 load-balancing split which will initially distribute user traffic evenly across all paths

After a set delay, the initialization procedure starts the PM flows at the source host. The PM flows follow their designated path described by the static flow rules, and the destination host collects the end-to-end latency values. Shortly after, the generation of user traffic starts, and finally the initialization procedure initiates the dynamic simulation of link degradation.

4.2 Components

In this section, we outline the implementation of each component based on its design described in Chapter *3 Design*.

4.2.1 Microwave mobile backhaul

We simulate the microwave mobile backhaul in Mininet v2.1.0, which is the first version that includes support for running Open vSwitch in user-space mode [42]. Running the switch in user-space mode is a requirement for our implementation of stochastic switching (see Section *4.2.2.2 Software switch*).

Mininet automatically sets up MAC addresses according to a predetermined scheme, starting at `00:00:00:00:00:01` (in contrast to randomized) and thus remains the same on each run of the simulation. This behavior allows us to avoid unnecessary complexity during simulation startup, such as communicating the MAC address of the start and end node to the SDN controller. For the same reason, we use static ARP tables set up at startup. Dynamic MAC address resolution would require additional static flow rules, and makes debugging of the network traffic more difficult.

4.2.1.1 Topology

We define the topology as a custom topology in Mininet using the mininet.topo API, and load it when the simulation starts. The topology contains the six switches `s1, s2, s3, s4, s5, s6`, and the two hosts `h1` and `h2`. Appendix *C Mininet topology* contains the source code for the implementation.

4.2.1.2 Link model

We implement the dynamic link model in Python together with the Mininet network simulation. This allows us to leverage the Mininet API, which provides objects for switches, hosts and links, and ready functions for interacting with the virtual network hosts.

The architecture of our link modeling design makes it possible to run a separate and independent time-stepped simulation of capacity for each individual link. This architecture allows high flexibility in the type of link behavior patterns that we can simulate, such as staggered link degradations and other interference effects. Our link model describes how the capacity of a link fluctuates over time, and it is easy to switch out, making it possible to plug-in any kind of link behavior. Essentially, the link model can use any function defined by:

$$f(t) = \texttt{bandwidth of link at time } t$$

The core of the implementation contains the three classes `Model`, `LinkSimulation` and `Simulation`:

- `Model`

  An interface that represents a link model, which describes how the capacity of a link changes over time. The Model interface defines a single method *step*, which advances the simulation by one step. The reason why the model is mutated rather than acting as a pure function is to allow the use of link models that are impossible or inefficient to reverse (such as PRNGs), or functions that depend on some external state or I/O. When the model is called (objects can be callable in Python by implementing the special method `__call__`), it returns the capacity of the link at the current step.

- `LinkSimulation`

  A simulation of a single link which runs in a separate thread – initialized by supplying a link and a model. In this context, we define a link as a pair of nodes; a switch and a switch, or a switch and a host. In addition, the two nodes must have a network interface connecting them. We then step the model at a set interval to retrieve a new capacity value. Through the use of the traffic control (`tc`) system command, we throttle the network interfaces on both ends of the link.

- `Simulation`

  Essentially a container class for managing link simulations. The class maintains a list of all existing link simulations and exposes methods to collectively start and stop them, with the possibility to set a start delay.

We simulate changes in link capacity by applying various rate limits on the virtual network interfaces using the `tc` command. The `tc` command is a complex tool providing a multitude of different parameters, but a common use case which is also applicable here is to limit the throughput of a network interface. Consider the following example commands which limits the throughput between switches `s1` and `s2`, which are connected through the interfaces `s1-eth1` and `s2-eth1`:

```
$ tc qdisc replace dev s1-eth1 root handle 1: tbf rate 40mbit
burst 500kB latency 100ms

$ tc qdisc replace dev s2-eth1 root handle 1: tbf rate 40mbit
burst 500kB latency 100ms
```

In our basic use case, the only parameters of interest (which affects the rate limiting) are; the queue algorithm Token Bucket Filter (TBF), the size of the bucket (as specified through the burst parameter), and the maximum time a packet can wait in the bucket before being dropped (specified through the latency parameter).

Note that the above rate command only limits the *average* throughput rate (a consequence of using a TBF algorithm). Bursts are allowed through and traffic will thus occasionally exceed the set rate, but the average should stay close to its configured value. If this is not desirable there is an additional command called peak rate, which as its name suggests limits the maximum possible throughput. Peak rate limits the rate at which the bucket can be emptied, and packets that have not entered the bucket (received a token) in the allotted time (as set by latency) will be dropped.

There is however problems with using peak rate in addition to rate in order to achieve a more exact rate throttling:

1. It is hard to determine which value to use for the peak rate when used in addition to an average rate. If both rates are set to e.g. 40 Mbit/s, there will be a conflict between maintaining an average rate of 40 Mbit/s while never exceeding 40 Mbit/s. It is reasonable to suspect that some traffic is dropped when attempting to fulfill both these conditions. If the peak rate then instead must be set higher than the average rate, how much higher should it be set? The configuration of the rate and peak rate is highly dependent on the purpose of the rate limiting. The stated goal of a rate limiting that matches the degradation characteristics of a microwave link is hard to translate to rate and peak rate parameters.

2. Rudimentary tests show that setting a peak rate does not work particularly well with virtual network adapters, since a virtual network adapter is bound by the OS scheduler. While the CPU is context-switching and dedicating time to other processes, packets are being buffered. When execution continues, packets are released in a burst which, if a peak rate is set, is highly susceptible to packet drop.

4.2.2 SDN

The SDN component provides the infrastructure around which we build the implementation. The SDN infrastructure includes an OpenFlow-enabled software switch (Open vSwitch) and an SDN controller (Floodlight) for communicating with the switches.

4.2.2.1 SDN Controller

We implement the overall architecture of the SDN controller as modules in Floodlight using the Java programming language. Floodlight's API is built around modules, which either consume or expose one or more services. Floodlight uses a dependency-injection system to connect these modules at runtime. Changes to the switching tables of the OpenFlow-switches are performed either as a response to packet-in messages from a switch or by actively pushing out OpenFlow commands to connected switches.

A central object in the implementation is the `Route` object, which is a representation of a directed path in the network graph between two nodes. In the context of this thesis, a `Route` is always a directed and unique path between the designated start and end node. The `Route` is internally represented by a list of `NodePortTuples` (see Figure 9), which are pairs of a node (switch or host) and a port (inport or outport, depending on if it represents traffic entering or leaving the node). The first `NodePortTuple` in the list is always an inport and the final `NodePortTuple` is always an outport. An edge can in this context be described as two `NodePortTuples` that are connected with network interfaces.

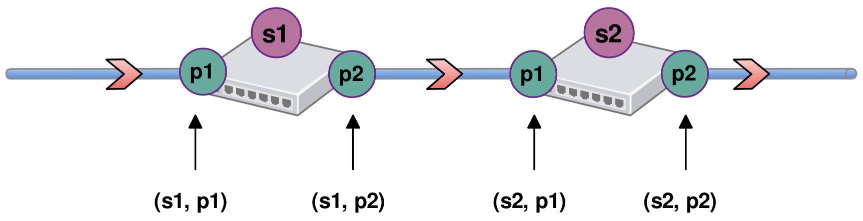

Figure 9: *Example* `Route` *formed by 4* `NodePortTuples`

Figure 9 provides a visual overview of how a `Route` involving the two switches `s1` and `s2` is built up using `NodePortTuples`. `Route` objects are used throughout the execution flow in the SDN Controller, which we extend with the following custom Floodlight modules:

- `RouteManager`

  We implement this module to listen to Link Layer Discovery Protocol (LLDP) messages, thereby enabling it to construct and maintain a map of the OpenFlow-enabled switches in the network. It can detect changes to the topology that introduces additional paths in the network and communicate the updated list of paths to other modules.

- `TestFlowManager`

  This module is responsible for the PM flows. When it receives a *routesChanged* event from the `RouteManager`, is sets up static flow rules guiding the PM flows along their designated paths. The `TestFlowManager` also maintains a TCP socket from which it receives end-to-end latencies for each PM flow (given by the end node). It maps each received latency to its path using the received port number and communicate the results to other modules.

- `LatencyAnalyzer`

  This module consumes and analyzes latencies provided by the `TestFlowManager`.

4.2.2.2 Software switch

We use Open vSwitch v2.1.0, which we manually compile from source with a modification to the group select algorithm. The modification enables us to use stochastic switching with the select OpenFlow group. See Appendix *B Open vSwitch modification* for the source code of the modification.

The modification requires the switch to run in user-space mode in Linux. In Linux, the network stack runs in kernel-space and to achieve a high networking performance Open vSwitch attempts to avoid crossing the boundary between kernel and user-space as much as possible, since crossing into user-space is relatively costly in terms of performance. In its current implementation, the stochastic switching modification invokes the *rand()* function for each individual packet. However, this function and `stdlib.h` as a whole is not available in kernel-space, and to run stochastic switching we therefore have to run Open vSwitch in user-space mode. While this significantly slows down the switching speed, the speeds achievable in user-space are sufficient for the work in this thesis. The other work on stochastic switching [36] runs the simulations on OF13SoftSwitch [43], which runs solely in user-space.

In a non-simulated environment, stochastic switching can be implemented using a Weighted Round-Robin (WRR) or a Deficit Round-Robin (DRR) algorithm. These two algorithms do not have any of the above restrictions or performance implications, since they are simple enough to be implemented in hardware [44, 45].

4.2.3 PM

We split up the logical construction of the PM component implementation into two independent parts:

1. A Python script that starts the sender and receiver of the PM flows on the two hosts.
2. Two modules in Floodlight which receive and analyze latencies – continuously computing a performance value for each path in the network.

4.2.3.1 Python script

We implement the Python script as part of the script that initializes the Mininet simulation, since it relies on the Mininet API to invoke system commands on the virtual hosts created by Mininet. This integration allows us to, from the Python script, perform network operations using the virtual network adapters of the hosts.

Furthermore, we implement the PM flows as two small programs written in C; a sender which generates UDP packets at a given rate containing a timestamp of when the packet was sent, and a receiver which receives the packets and outputs the end-to-end latency of the packet to `stdout`. We assign each sender-receiver pair a unique destination UDP port to differentiate the path that is being measured. PM traffic is sent at a configurable rate of between 5 - 50 packets per second, and the size of a PM packet is 48 bytes. The PM packet consists of a 28 bytes UDP header, a 4 bytes sequence number integer, and a 16 bytes timestamp (timeval struct from `time.h`). We therefore get a traffic overhead of 0.24 - 2.4 kB/s per path by using active PM. Over 8 paths, this adds up to a total overhead of less than 0.2 MB/s, which is very lightweight compared to many other measurement methods.

We continuously parse the latency output of the packet receivers and messages are echoed over a TCP connection to the PM component in the SDN controller using an ad-hoc text format. The message contains a port number (which identifies the path), a packet sequence number (to detect packet drops), and a latency value in seconds. In this case, the communication is passed through a loopback TCP socket and is thus outside of the Mininet virtual network.

4.2.3.2 Modules in Floodlight

In the SDN controller, we split the PM component into the two Floodlight modules `TestFlowManager` and `LatencyAnalyzer`.

First, we push out static flow rules for each PM flow using a method called *pushRoute*, which can be described by the following psuedocode:

```
path := array of n NodePortTuples
for $i = 0, i < n, i \leftarrow i + 2$ do
    $sourceNode, sourcePort \leftarrow path[i]$
    $destNode, destPort \leftarrow path[i + 1]$
    $match \leftarrow createFlowMatch(sourcePort)$
    $action \leftarrow creatFlowAction(destPort)$
    writeFlowMessage(destNode, match, action)
end for
```

One interesting property about the above pseudocode is that the `sourceNode` and `destNode` variables always refer to the same node. Each pair of `sourceNode` and `destNode` thus represents the *internal* link between inport and outport inside the switch. This internal link is natural, since it essentially represents what a flow rule is – a mapping between a in-and-out port on a switch.

As soon as we have started the PM traffic, we receive messages about each path over the TCP connection. We then parse the messages and use the UDP port to lookup the `Route` object corresponding to the PM flow. Then we send the route, latency, and packet sequence number to the rest of the Floodlight modules.

The `LatencyAnalyzer` receives the latency values for each path and we buffer the latencies per path in an array. We keep a counter of the last received packet number for each `Route`, and detect packet drop through jumps in sequence numbers. When a packet is dropped, we add a penalty value, which is a configurable but high latency value. At a set rate, we sort the array of all received latencies since the last update in ascending order, and pick the 90th percentile as the performance value of the path. We therefore use a high, but not the highest, latency to represent the current performance of the path.

This is a simple and very lightweight method that captures some part of the relationship between latency and link congestion. The main weakness of this approach is that it assumes a linear relationship between latency and congestion, which is most likely not the case.

4.2.4 Load-balancing

Most of the work performed by the load-balancer is to convert the relative congestion value for each path to the relative congestion value of each link, and to compute output probabilities for all switch inports.

For each switch inport, we create an OpenFlow select group, with one bucket per applicable outport (excluding the inport itself which if included would result in a loop). Each bucket contains one action, which is an output action outputting packets on the respective outport. We assign each bucket of each group a probability of 1/ `number of buckets in the group`, which means that the initial load-balancing state is an even split of traffic. For each group, we then add a flow rule directing incoming packets on the respective inport to the group, which selects an outport to output the packet on. In the topology we use, this procedure results in a total of 11 groups and 11 flow rules.

In contrast to all other controller-switch communication, we invoke the `OFTP_GROUP_MOD` and `OFTP_GROUP_ADD` commands directly through system commands using Open vSwitch's CLI interface. This is possible since we are running the SDN controller on the same machine as the Mininet network simulation. The reason why we have to issue commands using Open vSwitch's CLI interface is because Floodlight only implements OpenFlow 1.0. Even though the other main controller alternative OpenDaylight supports OpenFlow 1.0, 1.1, 1.2, and 1.3, the optional select group is still not implemented in any of the protocol versions.

To get a better intuition of how we convert a load-balancing solution (see Table 2) to output probabilities (see Table 3), we provide a few command line examples.

Table 2: *An Example traffic split across paths*

Path	Fraction of total traffic (%)
s1->s3->s4->s5->s2	10
s1->s4->s5->s6->s2	10
s1->s4->s5->s2	15
s1->s4->s3->s6->s5->s2	5
s1->s3->s6->s2	30
s1->s3->s6->s5->s2	15
s1->s4->s3->s6->s2	10
s1->s3->s4->s5->s6->s2	5

Table 3: *Example switch outport distributions*

Switch	Inport	Outports and distribution
s1	3	1 (40%), 2 (60%)
s2	1	3 (100%)
s2	2	3 (100%)
s3	2	1 (100%)
s3	3	1 (75%), 2 (25%)
s4	2	1 (100%)
s4	3	1 (62.5%), 2 (37.5%)
s5	1	2 (35%), 3 (65%)
s5	2	3 (100%)
s6	1	2 (35%), 3 (65%)
s6	2	3 (100%)

Initial group creation and 50/50 traffic split (example switch 1):

```
$ ovs-ofctl -O OpenFlow13 add-group s1 group_id=3, type=select,
bucket=output:1, weight:50, bucket=output:2, weight:50
```

```
$ ovs-ofctl -O OpenFlow13 add-flow s1 in_port=3, priority=10,
actions=group:3
```

`OFTP_GROUP_MOD` command issued to update switch 1:s output probabilities to match those in Table 3 (which has been compiled using the assigned splits in Table 2):

```
$ ovs-ofctl -O OpenFlow13 mod-group s1 group_id=3, type=select,
bucket=output:1, weight:40, bucket=output:2, weight:60
```

Important to note is that since Floodlight runs OpenFlow 1.0, all OpenFlow commands sent by Floodlight have the `OFP_VERSION` field set to `0x01`. In contrast to this behavior, we send the above commands with the version set to `0x04`. In its normal mode of operation however, Open vSwitch would not allow this. To enable Open vSwitch to accept multiple versions of OpenFlow simultaneously, we configure the switches at startup using the following command:

```
$ ovs-vsctl set bridge [switch-name] protocols=OpenFlow10, OpenFlow11,
OpenFlow12, OpenFlow13
```

where `[switch-name]` is `s1, s2, s3, s4, s5,` or `s6`.

At a high-level the load-balancing algorithm has the follow characteristics:

- Input:
  A congestion value or factor (i.e. some percentile of the latency) for each path in the network supplied by the PM component.
- Output:
  Appropriate updates to the group table of each OpenFlow switch, with calculated output probabilities on all outgoing ports for each switch inport.

We run the algorithm at a fixed interval (1.5 - 5s) and divide it into two separate steps with the tasks of:

1. Calculating a new traffic distribution, a value between 0 and 1 for each path representing the fraction of the total traffic that should pass through the path.
2. Translating the traffic distribution to output port probabilities for each switch inport.

We describe the two steps of the algorithm in detail:

1. Calculating a new traffic distribution

   The algorithm normalizes (relative to the sum) and inverts the congestion values from the PM component for each path to obtain the relative congestion (between 0 and 1) of each path in the network. Since we have a fixed amount of traffic that should be distributed over all paths in the network, there is no need to relate the congestion values to any absolute property such as bandwidth. By only comparing the paths relative to each other, we can avoid much of the complexity and source of inaccuracy from attempting to translate relative congestions to absolute properties.

As a next step the algorithm transforms the normalized congestion values, since these values can not directly be used as the traffic split to assign to the network. This can be emphasized by the fact that observed congestion measurements of 0.25, 0.25, 0.25, 0.25 (given four paths) does not mean that the traffic should be split equally between these paths, but rather that all paths are equally congested and that the currently assigned traffic split is optimal at the current level of traffic. Therefore, the algorithm instead calculates the weight of a path at update tick n, given the relative congestion c by:

$$w_n = max(w_{min}, w_{n-1}) \cdot c$$

where

$$w_0 = 1/\texttt{number of paths}$$

since we always start with an equal distribution of traffic, and w_{min} is the minimum level of traffic possible to assign to a path (e.g. 0.05 = 5%).

These weights (when normalized relative to the sum) translate directly to the distribution of the traffic in the network. A weight of e.g. 0.3 means that 30% of the total traffic will pass through that specific path.

To prevent the assigned fraction of traffic for a path to reach zero we put a minimum limit w_{min} on the amount of traffic to be sent through a path. We choose this limit arbitrarily, but it should be higher than 0 and less than 1/`number of paths`, since a higher value would mean that the load balancing would be fixed at an equal distribution of traffic.

Although we in theory should allow the assigned traffic level of a path to reach zero (e.g. if the link is down), the use of latency values to measure link congestion means that as the amount of traffic assigned to a path approaches zero the amount of information obtainable from the latency value also approaches zero. A low latency value (= low congestion) on a path with no traffic provides no information at all, since it essentially states that "Given no traffic the congestion of this path is very low". This breakdown of the formula when traffic reaches zero is seen by the fact that if the weight of a path reaches zero it will remain at zero regardless of the congestion value of that path.

2. Translating the traffic distribution to output port probabilities

   Given the weights of all the paths in the network, the algorithm calculates the weight of each edge. Note that this only holds since every path traverses each edge at most once. Using the edge weights, the algorithm constructs the set of updated OpenFlow select groups by the following:

```
For each incoming edge (which maps exactly to one unique switch
inport) of each node
                    let $e_1, ..., e_m$
be the outgoing edges from that node, $p_i$ the switch outport
connected to $e_i$ and $w_{e_i}$ the weight of $e_i$.

It is then possible to construct the list of tuples

                 $(w_{e_1}, p_1), ..., (w_{e_n}, p_n)$

which corresponds to the buckets of the OpenFlow group matched
by a flow rule to the incoming edge (inport).
```

The algorithm normalizes the weights (to the sum of the weights) to represent probabilities and pushes them to the switches using the `OFTP_GROUP_MOD` command.

4.2.5 Traffic generation

A separate component in the Python Mininet simulation script generates the user traffic, and uses the same sender utility program as the script that generates the test traffic. We run the sender program as three instances, one for each packet size of the Simple IMIX traffic distribution. The script accepts a single parameter, the amount of traffic to send in Mbit/s. The 7:4:1 distribution of Simple IMIX is achieved by computing the rate of packet generation (frequency) required of each packet size to reach the given traffic level:

$$\begin{aligned} f_{1500} &= \frac{10^6 \cdot \frac{1}{8} \cdot \texttt{Traffic in Mbit/s}}{(1500 \cdot 1 + 576 \cdot 4 + 40 \cdot 7)} \\ f_{576} &= 4 \cdot f_{1500} \\ f_{40} &= 7 \cdot f_{1500} \end{aligned}$$

The traffic that reaches the receiving end is logged and compiled into a report detailing statistics such as traffic received and sent, network performance metrics such as mean, max and min latencies, and total packet drop.

5 Evaluation

In this Chapter, we describe how the evaluation of the implemented solution is performed and present the results. We define the criteria of the evaluation and specify the compared load-balancers in more detail. Furthermore, we outline the setup specifics and describe the simulation procedure. Finally, we present and discuss the results of the evaluation.

5.1 Criteria

The responsibility of a microwave mobile backhaul is to carry user traffic from one end to another, making sure as much traffic as possible reaches its destination. Therefore, the main objective is to minimize the packet drop level. Another important factor is latency; packets should not only be delivered, they should also be delivered with as little delay as possible.

Since the idea of performing dynamic load-balancing in a microwave mobile backhaul is to avoid drops by rerouting traffic around degraded microwave links and thereby maximize the utilization of the microwave mobile backhaul, the key criterion here is packet drop level.

With that being said, it is still important that all delivered packets are delivered within a reasonable time frame. Such threshold is determined by the size of the switch queues (buffers) that a packet travels through. Packets that remain too long in a queue is dropped. This means that making sure that latencies are tolerable (i.e. packets are delivered in time) is a matter of building/configuring the network itself, and not a job of the load-balancer. The goal of the load-balancer is thus limited to getting as many packets through the network as possibly, i.e minimize the packet drop level.

We can express drop level by any of the following quotients:

$$\frac{\texttt{number of received packets}}{\texttt{number of sent packets}} \tag{1}$$

$$\frac{\texttt{received data amount}}{\texttt{sent data amount}} \tag{2}$$

(1) is the common way of measuring packet drop, and is arguably the best measurement for drop depending on context. For example when using TCP, a dropped packet triggers retransmission independently of the packet size, meaning that a high number of drops might be worse than dropping fewer packets of greater size.

In this evaluation however, we primarily measure drop level by using (2), which can be described as data drop level. The reason why we choose data drop level is to enable performance comparisons with other load-balancers using mathematical integration as described in Section *5.4 Procedure*. In the case of an optimal load-balancer, where the throughput is theoretically calculated, there are no actual packets to count. Thus, (1) is rendered inapplicable for the purpose of comparison.

Although (2) describes a highly relevant drop measurement, it is still possible to achieve a reasonably low percentage while (1) says otherwise. It could for example be the case that all small packets are dropped in favour of larger packets, causing the drop level according to (1) to be high while not severely impacting the drop level described by (2).

Therefore, as a way of assuring that the implemented load-balancer is not flawed, we also calculate the drop level according to (1) as a complement to the comparison of drop level given by (2).

5.2 Compared load-balancers

In the evaluation of the implementation, we compare three different load-balancers. Firstly, the base case where no dynamic load-balancing is performed; secondly, the load-balancer described in this thesis; and finally an optimal load-balancer, to establish the maximum potential of a load-balancer in the simulated environment.

5.2.1 Static load-balancer

To be able to evaluate the implementation, we require a reference load-balancer. However, within the scope of this thesis, it is not feasible to compare the performance of the dynamic load-balancer in the simulated environment to any implementation in an actual microwave mobile backhaul.

One option in this situation is to treat the case where no load-balancing at all is being performed as the base case. However, given that the topology contains alternative paths, this would not be realistic or especially fair. If redundant links exist, they should to some degree be utilized.

In a network topology where there exists redundant links, the arguably simplest and most effective way to utilize them is to balance the traffic in a way that achieves the maximum throughput. Given the symmetry in our chosen topology (see Figure 10), we can achieve maximum throughput by evenly distribute all incoming traffic on a switch port over all outgoing switch ports.

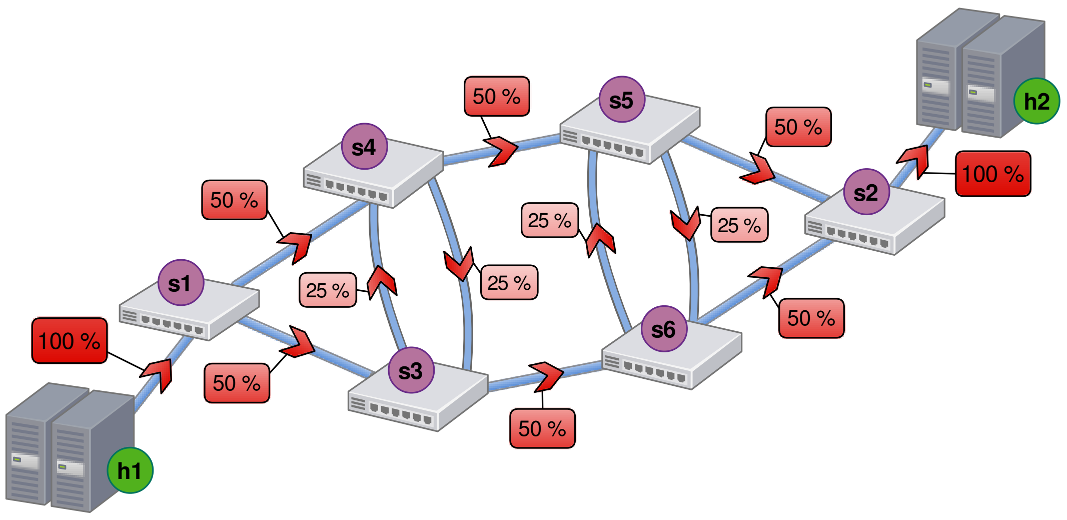

Figure 10: *Static load-balancer traffic split, all links explicitly marked with the fraction of total traffic passing over them*

Figure 10 describes in detail how the traffic generated from `h1` is split evenly and eventually received by `h2`. In this configuration, which we refer to as the static load-balancer, 50% of the traffic passes each link in the outer ring, while 25% of the traffic passes through each cross link (in both directions) according to Figure 10. Note that there is only one available outgoing switch port for traffic that enters a switch through a cross-link, meaning that all the traffic passing through a cross link is sent to that outport.

5.2.2 Dynamic load-balancer

The second case is the dynamic load-balancer, which is the solution described in Chapter *3 Design* and Chapter *4 Implementation*. This is the load-balancer that is used when running the simulation of the modeled microwave mobile backhaul.

5.2.3 Optimal load-balamcer

In addition to a "base case" static load-balancer, we also desire some kind of optimal load-balancer. The purpose of such reference load-balancer is to act as a benchmark, providing an upper performance limit of a load-balancing solution, thereby putting the performance of the dynamic load-balancer into perspective.

We realize the optimal load-balancer as a variant of the dynamic load-balancer, with the difference that the optimal load-balancer is assumed to have full knowledge about the state of the microwave links. It is assumed to have access to perfect information about the capacities of the links at each point in time, and it makes instant transitions between load-balancing states.

Given that this optimal load-balancer is independent from any PM method, and has a maximum granularity load-balancing, it establishes the maximum potential of any load-balancer in the given network topology. This also settles the minimum, theoretically achievable, level of packet drop for the simulation. Note that traffic throughput exceeding the current maximum throughput of the network, dependant on the current state of the link degradation scheme, is always dropped regardless of how well the traffic is load-balanced.

As is also the case with the static load-balancer, we do not actually run the optimal load-balancer in the simulation to gather drop statistics. Instead, we make a theoretical calculation of the data drop level. Since the link simulation model is repeatable, we can retrieve the capacity of each link in each step of the simulation. Using a max-flow algorithm, we can then calculate the best (allowing the maximum throughput) possible load-balancing weights. The load-balancer is assumed to transition to a new state instantly, thus achieving a perfect load-balancing state in each step of the simulation.

When we compare the different load-balancers, it is important to note that the maximum throughput calculation does not take switch buffers into account, which means that the minimum drop level may be higher than the actual drop level recorded in the simulation. The amount of drop the buffers can protect against is highly dependent on how the traffic exceeding the maximum throughput is distributed.

5.3 Setup

In order to be able to perform the evaluation, we need to establish two things; which link simulation scenario to run; and which parameters to use for the dynamic load-balancer and link simulation.

5.3.1 Simulation

In the evaluation, we use a single link simulation scenario for all test runs (see Figure 11). The scenario follows the model of flat and multi-path fadings as described in Section *3.2.1.2 Link modeling*.

We choose an arbitrarily seed for each independent link simulation, giving each link a unique bandwidth degradation scheme. Figure 11 illustrates the simulation of the links, showing the capacity of each link over the duration of the simulation.

The base capacity of the links is set low enough to produce drops with the static load-balancing solution (leaving room for improvement), but high enough for the network to be able to handle all traffic without drops given the optimal load-balancing solution. Depending on the link degradation scenario however, even the optimal load-balancer should produce some drop.

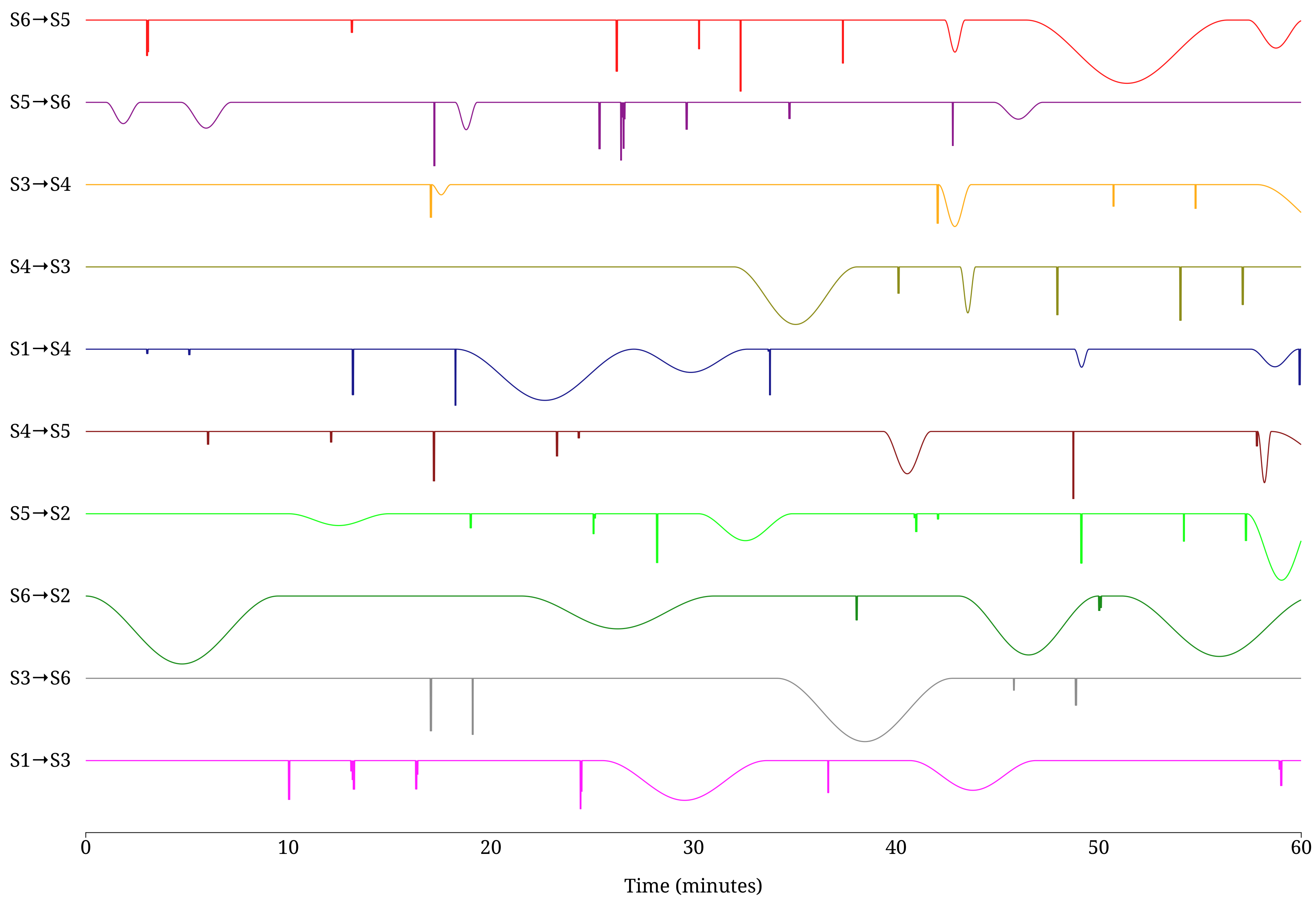

Figure 11: *The link degradation scheme for the simulation*

5.3.2 Configuration

There are some parameters that we need to set in order to be able to conduct a simulation run. We first present the parameters that we use in the evaluation simulation runs and then we describe each parameter in more detail.

IMIX rate: 60 Mbit/s

Link rate (high/low): 40/5 Mbit/s

Simulation steps: 7200

Time resolution: 0.5 s

Simulation time: $7200 \cdot 0.5s = 1h$

PM packet size: 48 bytes (i.e. empty UDP packets)

PM packet rate: 10 packets/s

Load-balancer update interval: 2.5 s

IMIX rate
The fixed rate at which user traffic is generated from the start host (`h1`) in the network. The traffic is generated according to the Simple IMIX packet size distribution. We choose to set this traffic rate to a level that roughly corresponds to the capacity often seen on microwave links, while also low enough to enable simulation without performance issues in the virtual network.

Link rate (high/low)
These are parameters of the link modeling simulation, determining the base rate (high) and lowest possible rate (low) of the links. The base rate is the initial and maximum bandwidth capacity of a link, and is the rate at which it normally operates given no external impacts. We keep the dynamic changes and dips in capacity within the set high/low rate. All links are set to the same high/low rates.

Together, the IMIX and link rate parameters determine the maximum throughput of the network when all links are operating at their maximum capacity, which in this specific configuration is 80 Mbit/s. Using the chosen network topology, the maximum throughput of the network can be reached by a 50/50 split of traffic, which is the traffic split used by the static load-balancing test case.

Simulation steps and time resolution
Determines the length of the simulation. In each simulation step, we update the bandwidth capacities of all links. A lower time resolution thus results in a finer granularity link simulation. However, since each update requires us to issue a new set of tc set rate commands, a very low time resolution could result in performance problems.

PM packet size
The size of the UDP PM packet. In this configuration, we set the packet to be as small as possible, which at 48 bytes is only slightly larger than a UDP header. In addition to the header, the packet also contains a timestamp and a sequence number.

PM packet rate
The rate at which we send test packets. The intuition is that more packets should provide higher accuracy congestion estimates. The tradeoff is the increase in PM traffic and overhead of active PM. At the current PM packet rate, we get a total PM overhead of 48 bytes $\cdot$ 10 packets/s $\cdot$ 8 = 3.84 kB/s, which is less than 1% of the total throughput with links at their maximum capacities.

Load-balancer update interval
The interval between updates of the load-balancing state. Together with PM packet rate, this setting determines how many PM packets we use for each load-balancing decision. A higher value provides more packets on which we can base a load-balancing decision, which should increase the quality of the decision. The downside is that higher values increase the time it takes to detect and respond to changes in link capacities. Using the current parameter values, we us $10 \cdot 2.5 = 25$ PM packets to determine the performance value of each path.

5.4 Procedure

The overall procedure of the evaluation is as follows:

1. We select arbitrary seeds for the link models, and simulate the links according the implementation described in Section *4.2.1.2 Link modeling.*

2. We run the dynamic load-balancer and log its sequence of load-balancing states.

3. We calculate the packet drop from the log and compare to the precalculated drop levels of the static and optimal load-balancing test cases.

During a simulation run, we generate two main logs. Firstly, a log from the load-balancer detailing the sequence of load-balancing states and their respective traffic splits. Secondly, a log of the link modeling scheme, documenting for each step in the simulation the capacity of each link.

We timestamp all log entries, which makes it possible to reconstruct the capacities of each link together with the traffic split set by the load-balancer for any point in time. By comparing the maximum throughput with the available throughput from the traffic split, we can then detect if packets were lost due to an inferior load-balancing decision.

5.4.1 Packet drop calculation

In the analysis of a simulation, we describe each load-balancer by its sequence of load-balancing states (one state for each step in the simulation). In this context, we represent a load-balancing state as the fraction of total traffic assigned to each path in the network (see Table 4).

Table 4: *Even traffic split across all paths, each path accepting 1/8 of the total traffic*

Path	Fraction of total traffic (%)
s1->s3->s4->s5->s2	12.5
s1->s4->s5->s6->s2	12.5
s1->s4->s5->s2	12.5
s1->s4->s3->s6->s5->s2	12.5
s1->s3->s6->s2	12.5
s1->s3->s6->s5->s2	12.5
s1->s4->s3->s6->s2	12.5
s1->s3->s4->s5->s6->s2	12.5

As an example, the static load-balancer operates using only a single load-balancing state with evenly distributed fraction of traffic assigned to each path according to Table 4.

Given the knowledge of the load-balancing state, the amount of traffic, and the link capacities in all steps of the simulation, we can determine the maximum throughput at step i for all three load-balancers by a function $throughput(i)$. The throughput is the result of the traffic split assigned by the load-balancer. More formally, given the following:

(1) $state(i) = f_1(i), f_2(i), \ldots, f_j(i), \ldots, f_n(i)$

The load-balancing state at simulation step i, where $f_j(i)$ denotes the fraction of total traffic (0-1) that passes through link j at step i. Note that the load-balancer assigns traffic per *path* and not per link, but the fraction of traffic per link can be derived from that information.

(2) $capacity(i) = c_1(i), c_2(i), \ldots, c_j(i), ..., c_n(i)$

The capacity (e.g. Mbit/s) of all n links at simulation step i, which is known from the link model simulation scheme. $c_j(i)$ is the capacity of link j at step i.

(3) $traffic(i) = C$

The constant function for the level of generated traffic (Mbit/s).

(4) $throughput(i)$

The maximum throughput of the network at step i, calculated using a max-flow algorithm, where the link capacity is set to $min(c_j(i)), traffic(i) \cdot f_j(i))$. $f_j(i)$ is the fraction of total traffic passing through link j at step i, which is implicitly set by the load-balancer. Using the static load-balancer, $f_j(i)$ will always be a static value of 0.5 (outer ring) or 0.25 (cross links). In the optimal load-balancer, the link capacity will always be $c_j(i)$, since the optimal load-balancer has perfect information about the links' capacities.

we can obtain the total packet drop of each load-balancer through integration of their discrete $throughput(i)$ function:

$$packetDrop = \int traffic(i) - \int min(throughput(i), traffic(i))$$

This corresponds to the fact that traffic is dropped where $throughput(i) < traffic(i)$.

5.5 Results and discussion

The outcome of the evaluation is two-fold:

1. We show that the implemented PM method and load-balancing algorithm achieves a clear reduction in packet drop as compared to the static load-balancing case using the described setup.
2. Variations within reasonable ranges in the amount of PM traffic (4-19 kB/s) and the frequency of load-balancing updates (1.5-5s) has little or no impact on the overall effectiveness on the dynamic load-balancing.

5.5.1 Load-balancer comparison

The simulation run and the analysis of its resulting logs when sending a total of 27000 MB traffic (60 Mbit/s · 3600 s) resulted in the drop levels given in Table 5.

Table 5: *Calculated data drop per load-balancer*

Load-balancer	Dropped data (MB)	Dropped data (%)
Optimal	1314	4.9
Dynamic	2118	7.8
Static	3435	12.7

Table 6 shows the actual drop levels (packet drop and data drop) measured during the simulation run using the dynamic load-balancer. The difference between the calculated result given in Table 5 and the measured result shown in Table 6 is explained by the switch buffers used in the actual simulation.

Table 6: *Measured drop in simulation using the dynamic load-balancer*

Dropped packets (%)	Dropped data (%)
1.8	2.4

The distribution of the packet drop level shown in Table 6 is detailed in Table 7, showing the packet drop level per Simple IMIX packet size.

Table 7: *Measured drop per Simple IMIX packet size*

Packet size (bytes)	Dropped packets (%)
1500	2.9
576	2.1
40	1.2

Table 7 shows that the packet drop level is more or less evenly distributed among packet sizes, suggesting that the dynamic load-balancer is fully capable of handling different sized packets. The result of the simulation run is further illustrated by Figure 12, which describes the maximum possible throughput achievable by each load-balancer at each point in time. The time corresponds to the duration of the simulation and maps to the timeline of the link degradation scheme (see Figure 11).

The green curve in Figure 12 represents the optimal load-balancer, which consequently shows the maximum possible throughput of the network at any given time. At times when this curve dips below the static traffic level of 60 Mbit/s, sent data is inevitably dropped as there are no configuration of the network that can carry all traffic. The network is at this point operating at its maximum capacity.

The shape of the optimal throughput curve does not only depend on each individual link capacity, but also on the interference of the individual link degradations. Outer ring links has a more direct impact on the total throughput of the network than the cross-links. At the initial state of the network, the cross-links are essentially superfluous – since they do not provide any additional throughput unless one of the outer-ring links are congested. At the point of congestion however, the cross-links are periodically used to transfer traffic over to the other non-congested side of the topology. This can be exemplified by the fact that the severe degradation of link `(s6,s2)` between minute ∼2 and minute ∼10 (see Figure 11) results in a severe drop in network throughput (see Figure 12), while e.g. the equally severe degradation of link `(s6,s5)` at its lowest point at ∼52 minutes has little impact on the overall throughput.

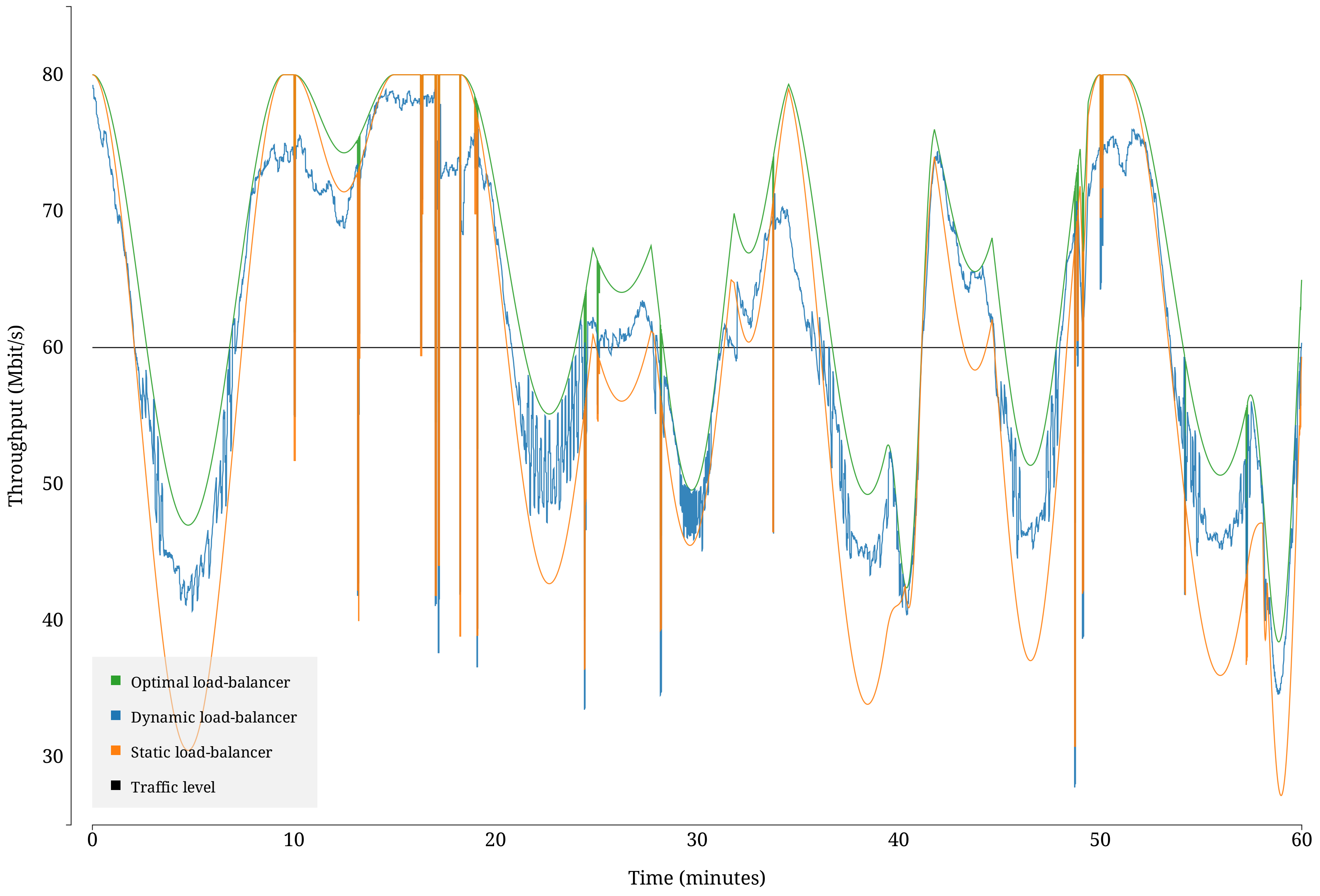

Figure 12: *Plot of the throughput result of each load-balancer*

The blue and the orange curve in Figure 12 represents the dynamic and static load-balancers respectively. As anticipated, we can see that the static load-balancer matches the optimal load-balancer in the case where links are not degraded, but falls short when links starts to get degraded.

Note that the behavior of the dynamic load-balancer (blue) above the level of generated traffic is less meaningful, as at this point no traffic is being dropped. Given our design of the dynamic load-balancer, the use of congestion as a basis for the load-balancing makes it hard to maximize the throughput once it is above the current level of generated traffic, at which point there are no congestion to measure.

Above the level of generated traffic, the load-balancing state will simply remain at the state it had immediately before reaching a no congestion state. When the congestion disappears (i.e. the congestion value for each path is equal), as a consequence of the algorithm we use (see Section *3.2.4.2 Finding an optimal load-balancing solution*), there will be no further changes to the load-balancing state. The goal of the algorithm is simply to reach a load-balancing state where the congestion is zero.

As an intuitive example, let us note that a 10 Mbit/s, a 500 Mbit/s, or a 1 GB/s link may all show the same latency value when not congested. There is no direct relationship between latency and bandwidth, only the indirect relationship through congestion. To guarantee that a maximum throughput load-balancing state is reached, some other kind of measurement technique is thus necessary, such as the more intrusive methods (ASSOLO, pathChirp, Pathload) where a congestion is artificially induced.

Another interesting phenomenon shown in Figure 13 is the dip that occurs below both the static load-balancer and the traffic level at $\sim$32 minutes, even though the total throughput is steadily increasing. One possible explanation is that once the congestion is gone, the load-balancing state ceases to change, resting at a level only slightly above the level of traffic. Small variations in the latency values makes the load-balancer drift below the level of traffic. At this point, there is suddenly a spike in congestion and the load-balancer responds with a relatively large change to the traffic distribution.

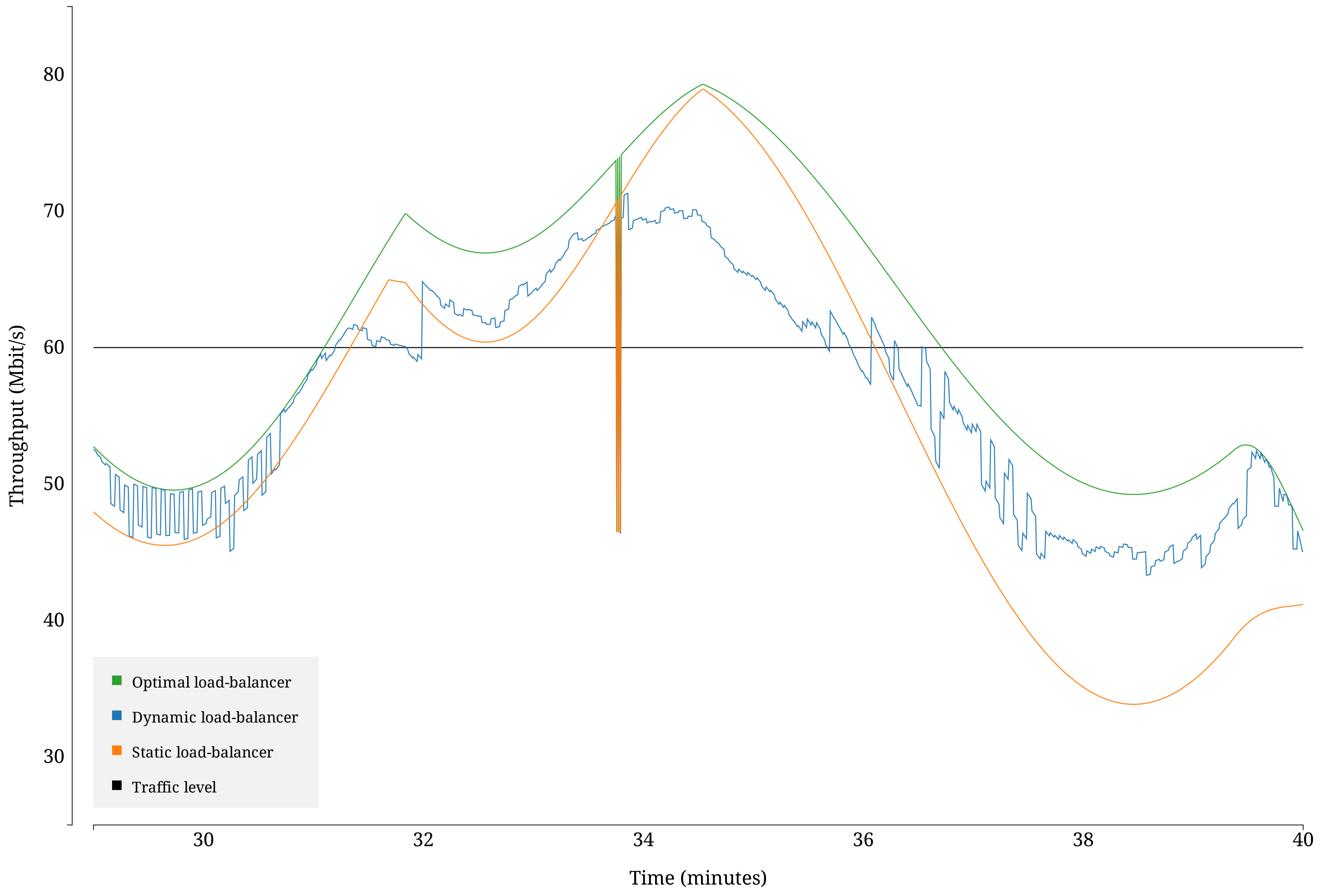

Figure 13: *Subset of the throughput plot*

The jaggedness of the curve implies a slight imbalance in the load-balancing, which is not unexpected given our limitation of not focusing on load-balancing transitioning in this thesis. The most likely cause for the imbalance is our tendency of making too drastic changes when we transition to new load-balancing states. The continual over- and under compensation in the correction of the load-balancing appears to lead to perpetual fluctuating states. One solution to this problem could be to introduce some kind of adaptive inertia factor, e.g. setting an upper limit on how much the fraction of total traffic assigned to a path can change within a single load-balancing state. If this limit is hit within a load-balancing state it would subsequently be slightly increased (by some constant value), or conversely decreased if it is not hit. This would prevent a sudden spike in congestion from completely reshaping the load-balancing state, while still allowing the load-balancer to respond to more persisting trends in network congestion.

Note however that the fluctuations are not particularly large – in the mentioned example they are only between ∼47 and ∼50 Mbit/s. On a larger scale, despite the fluctuations, the dynamic load-balancer stays very close to the optimal load-balancing state in the areas of interest (that is, where the throughput is below the generated traffic).

Even though the results suggest that end-to-end latencies can be successfully used as a basis for congestion-based load-balancing decisions, we note that there are limitations in using latency as a direct approximation of congestion. Our implementation of the PM component assumes a linear relationship between latency and congestion, which is most likely a false assumption. The relationship may well be linear in some latency range (e.g. around the median latency), but it is reasonable to believe that very low or very high latencies do not follow the same approximate linear relationship. For example, while an increase from 20 ms to 40 ms latency might conceivably indicate double the congestion, an increase from 0.1 ms to 0.2 ms does most likely not. The implications of this false assumption could be part of the explanation for the slight but notable transitioning fluctuations we observe.

5.5.2 Different load-balancer parameter values

In addition to the main simulation run, we performed a series of additional simulation runs with different values for the load-balancer update interval and PM packet rate, to determine their impact on the quality of the dynamic load-balancing.

In total, we performed 12 simulation runs, covering different combinations of PM packet rates and load-balancer update frequencies. We describe the results of these runs in Table 8, which contains values for the calculated data drop levels followed by the actual measured packet drop levels within parentheses.

Table 8: *Resulting drop values when running the simulation using different key parameters*

PM update interval (s) / PM packet rate (pkts/s)	1.5	2.5	5
5	7.9 (1.8)	8.0 (2.4)	8.1 (1.7)
10	7.8 (1.8)	7.8 (1.8)	7.9 (1.6)
25	7.8 (2.3)	7.9 (2.0)	8.1 (1.4)
50	7.8 (1.6)	7.9 (1.6)	8.0 (2.1)

Surprisingly, Table 8 shows no significant differences in the overall results when using different parameters for PM packet rate and load-balancer update interval. The intuition is that an increase in PM packet rate coupled with a relatively short update interval should increase the accuracy of load-balancing decisions by providing larger sample sizes to base the decisions on. However, there is also a tradeoff with adding the additional overhead of more PM traffic, which could be especially sensitive in cases when the congestion is already high. In a high congestion situation, a high PM rate (as compared to a lower PM rate) increases the risk of packet drops, since there are then more PM packets competing against user traffic.

The results in Table 8 suggests that a lower PM packet rate might provide a good enough estimate of the congestion, meaning that we could consider a higher PM packet rate excessive in this specific simulation using the given link degradation scheme. Note however that other topologies and link degradation schemes could result in other interpretations.

When it comes to load-balancer update interval, we barely notice a trend favoring a shorter update interval when we analyze the calculated drop levels in Table 8. However, we do not notice the same trend when we inspect the measured drop level within parentheses in Table 8. A trend favoring shorter update interval is what makes sense in theory (if coupled with a high enough PM packet rate), since the resolution of load-balancing decision is increased. In practice however, there are several complicating factors such as transitioning time and switch buffers.

Since new load-balancing states needs to be calculated and sent to the switches after each measurement interval, transitioning is not instant. In practice, this means that there might be a certain offset between the calculated load-balancing states and the desired ones. Such offset is likely to vary depending on technical aspects such as available CPU, OS scheduling, and the quality of the connection between the SDN controller and the software switches etc. Also, the specific link degradation scheme used could influence the result depending on how it maps to the timing offset of load-balancing states.

We also observe that given the typical length of multi-path fadings, we might yield better results by trying to ignore them in the load-balancer and instead let switch buffers handle them. The difficulty is then to specifically single out multi-path fadings using only a limited sample size of latencies. One way of doing this could be to look for substantial differences between latencies within the same sample, indicating sudden and volatile fluctuations.

Given the degree of uncertainty connected to finding the optimal PM packet rate and load-balancer update interval, we can not make any indisputable statement regarding optimal parameter settings without performing repeated simulation runs using different link degradation schemes.

6 Conclusion

In this thesis, we have shown that the addition of active PM to a simulated microwave mobile backhaul network using SDN and static load-balancing can in fact notably improve the overall network performance in terms of drop level by introducing dynamic load-balancing decisions based on the resulting PM metrics.

Our proposed design is a load-balancer which uses an iterative approximation of a congestion minimization algorithm, measuring congestion using end-to-end latencies. The load-balancer makes use of previous work on stochastic switching and OpenFlow to achieve a lightweight but effective load-balancing solution suitable in the context of microwave mobile backhauls where links are frequently affected by various dynamic effects.

We evaluated the dynamic load-balancer in a simulation of a microwave mobile backhaul, against a "base case" static load-balancer as well as an optimal load-balancer using an arbitrary microwave link degradation scenario. The results of the evaluation show a clear improvement in drop level over using a static load-balancing scheme, where traffic is split evenly across all paths. The network topology used in the evaluation, although relatively small, is realistic and has several alternative paths, making it a suitable representation of a microwave mobile backhaul. The microwave link model is simple, but captures the main characteristics of microwave links.

Since we performed the evaluation using a single link degradation scenario and one specific topology, we can not draw any conclusions about the general performance of the dynamic load-balancer when using an arbitrary network topology and other link simulation scenarios. To be able to draw such general conclusions about the dynamic load-balancer, it would be necessary to run simulations using a wide range of different link simulations and different topologies.

Although the evaluation of the dynamic load-balancer was limited in scope, the load-balancer and PM methodology have been designed for use in an indistinct virtual network making no assumptions on the network configuration or topology (with the exception that it should have alternative paths). Also, the link model is designed to be easily adapted to model many types of bandwidth degradation patterns and scenarios. Our proposed design is therefore not in any way limited to the specific topology used in the evaluation, nor the specific link degradation scenario.

To summarize, this thesis shows that performing active PM and using the resulting end-to-end latencies as a basis for making dynamic load-balancing decisions can be used to improve the performance of a simulated microwave mobile backhaul. The main contribution of this work is the establishment of active PM as a potentially viable candidate for performing dynamic load-balancing in the context of microwave mobile backhauls.

6.1 Future work

Ultimately, a desired continuation of the work presented in this thesis is to investigate how well the proposed dynamic load-balancer maps to a real-world microwave mobile backhaul. We do however suggest that further evaluation is carried out before such investigation, as well as research on compatibility with TCP flows. We therefore suggest the following direction for future work:

1. Evaluate the proposed load-balancer using other topologies and link model scenarios

   The evaluation of the work carried out in this thesis was only done using one specific topology and one specific link degradation scenario. Further evaluation should be carried out to determine if the performance improvement over static load-balancing seen in the evaluation in this thesis extends to arbitrary topologies and link degradation scenarios.

   Furthermore, the method of performing dynamic load-balancing using active PM is not inherently restricted to microwave mobile backhauls, although it is a good fit given the dynamic link behavior. The generality of the proposed design makes the dynamic load-balancer a candidate for serving as a basis for further work on more advanced and general-purpose SDN load-balancers also in other contexts.

2. Investigate the compatibility of the proposed load-balancer with TCP flows

   A limitation in the scope of this thesis is that the dynamic load-balancer only deals with UDP traffic, thereby making it possible to use stochastic switching and avoid partitioning of user traffic. When dealing with TCP traffic, a new set of challenges arises. First of all, there is the issue of keeping TCP flows intact to avoid packet reordering and subsequent retransmissions. This issue alone does however not require any further research, since it is solely a matter of partitioning incoming traffic into flows and performing load-balancing on flow level. It could however negatively affect the effectiveness of the dynamic load-balancer since the granularity level needs to be decreased.

   A more specific challenge that could arise when introducing TCP traffic to the dynamic load-balancer is a potential conflict between TCP's congestion control and the attempt of the dynamic load-balancer to minimize congestion. Suddenly, there are two "agents" trying to tamper with congestion, which could potentially cause undesirable oscillations between load-balancing states. However, it could also be the case that the net effect of a large number of independent TCP connections is indistinguishable from a constant UDP stream. In a network such as a microwave mobile backhaul, where both endpoints are controlled, it is also possible to tunnel traffic in such a way that it maximizes the effectiveness of the dynamic load-balancer.

Nevertheless, further research is needed to establish the compatibility of the dynamic load-balancer with TCP flows.

References

[1] "Openflow," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow, Open Networking Foundation.

[2] D. Venmani, D. Zeghlache, and Y. Gourhant, "Demystifying link congestion in 4g-lte backhaul using openflow," in *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*, May 2012, pp. 1–8.

[3] J. Moy, "Ospf version 2," http://tools.ietf.org/html/rfc2328, 1998.

[4] G. Malkin, "Rip version 2," http://tools.ietf.org/html/rfc2453, 1998.

[5] "Production quality, multilayer open virtual switch," http://openvswitch.org/, Open vSwitch.

[6] "An instant virtual network on your laptop (or other pc)," http://mininet.org/, Mininet.

[7] "Floodlight is an open sdn controller," http://www.projectfloodlight.org/floodlight/, Project Floodlight.

[8] "The journal of internet test methodologies," http://s3.amazonaws.com/zanran_storage/www.ixiacom.com/ContentPages/109218067.pdf, Spirent Communications, 2007.

[9] "Software-defined networking: The new norm for networks," Open Networking Foundation, Apr. 2012.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[11] "Openflow switch specification: Version 1.3.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, Open Networking Foundation, 2012.

[12] M. P. Fernandez, "Comparing openflow controller paradigms scalability: Reactive and proactive," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on.* IEEE, 2013, pp. 1009–1016.

[13] C. Dovrolis, P. Ramanathan, and D. Moore, "Packet-dispersion techniques and a capacity-estimation methodology," *IEEE/ACM Trans. Netw.*, vol. 12, no. 6, pp. 963–977, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1109/TNET.2004.838606

[14] J. Salmelin and E. Metsälä, *Mobile Backhaul.* Wiley, 2012.

[15] H. Lehpamer, *Microwave Transmission Networks, Second Edition*, ser. Communication engineering. McGraw-Hill Education, 2010.

[16] J. HANSRYD and J. EDSTAM, "Microwave capacity evolution," *The data boom: opportunities and challenges*, p. 22, 2011.

[17] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," in *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972422.1972438

[18] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM SIGCOMM Demo*, 2009.

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

[20] C. Macapuna, C. Rothenberg, and M. Magalhãs, "In-packet bloom filter based data center networking with distributed openflow controllers," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, Dec 2010, pp. 584–588.

[21] N. H, S. Seetharaman, M. Flajslik, A. Gember, N. Mckeown, G. Parulkar, A. Akella, N. Feamster, R. Clark, A. Krishnamurthy, V. Brajkovic, T. Anderson, D. T. R, and D. L. Usa, "Aster*x: Load-balancing web traffic over wide-area networks."

[22] M. Koerner and O. Kao, "Multiple service load-balancing with openflow," in *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*, June 2012, pp. 210–214.

[23] M. Jain and C. Dovrolis, "Pathload: A measurement tool for end-to-end available bandwidth," http://eeweb.poly.edu/el933/papers/pathload.pdf, 2002.

[24] J. Sommers, P. Barford, and W. Willinger, "A proposed framework for calibration of available bandwidth estimation tools," in *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*, June 2006, pp. 709–718.

[25] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell, "pathchirp: Efficient available bandwidth estimation for network paths," 2003.

[26] E. Goldoni, G. Rossi, and A. Torelli, "Assolo, a new method for available bandwidth estimation," in *Internet Monitoring and Protection, 2009. ICIMP '09. Fourth International Conference on*, May 2009, pp. 130–136.

[27] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329. [Online]. Available: http://doi.acm.org/10.1145/52324.52356

[28] A. Johnsson and M. Bjorkman, “On measuring available bandwidth in wireless networks,” in *Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on*, Oct 2008, pp. 861–868.

[29] J. Strauss, D. Katabi, and F. Kaashoek, “A measurement study of available bandwidth estimation tools,” in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’03. New York, NY, USA: ACM, 2003, pp. 39–44. [Online]. Available: http://doi.acm.org/10.1145/948205.948211

[30] “Opendaylight,” http://www.opendaylight.org/, Linux Foundation.

[31] “Noxrepo,” http://www.noxrepo.org/, NOXRepo.org.

[32] “Frequently asked questions,” https://github.com/openvswitch/ovs/blob/master/FAQ.md, Open vSwitch.

[33] “Open vswitch,” https://github.com/openvswitch/ovs/blob/master/README.md, Open vSwitch.

[34] “Faq floodlight openflow controller,” http://docs.projectfloodlight.org/display/floodlightcontroller/FAQ+Floodlight+OpenFlow+Controller, Project Floodlight.

[35] E. Goldoni and M. Schivi, “End-to-end available bandwidth estimation tools, an experimental comparison,” in *Proceedings of the Second International Conference on Traffic Monitoring and Analysis*, ser. TMA’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 171–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12365-8_13

[36] K. Shahmir Shourmasti, “Stochastic switching using openflow,” Master’s thesis, Norwegian University of Science and Technology, Department of Telematics, 2013.

[37] M. Shreedhar and G. Varghese, “Efficient fair queueing using deficit round robin,” *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 4, pp. 231–242, Oct. 1995. [Online]. Available: http://doi.acm.org/10.1145/217391.217453

[38] S. Kandula, D. Katabi, S. Sinha, and A. Berger, “Dynamic load balancing without packet reordering,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 51–62, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1232919.1232925

[39] J. Edmonds and R. M. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems,” *J. ACM*, vol. 19, no. 2, pp. 248–264, Apr. 1972. [Online]. Available: http://doi.acm.org/10.1145/321694.321699

[40] J. Chuzhoy and J. S. Naor, “New hardness results for congestion minimization and machine scheduling,” *J. ACM*, vol. 53, no. 5, pp. 707–721, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1183907.1183908

[41] K. Walkowiak, “Maximizing residual capacity in connection-oriented networks.” *Journal of Applied Mathematics and Decision Sciences*, vol. 2006, no. 3, pp. Article ID 72 547, 18 p.–Article ID 72 547, 18 p., 2006. [Online]. Available: http://eudml.org/doc/129490

[42] “Announcing mininet 2.1.0 !” http://mininet.org/blog/2013/09/20/announcing-mininet-2-1-0/, Mininet.

[43] “Openflow 1.3 software switch,” https://github.com/CPqD/ofsoftswitch13, ofsoftswitch13.

[44] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, “Weighted round-robin cell multiplexing in a general-purpose atm switch chip,” *Selected Areas in Communications, IEEE Journal on*, vol. 9, no. 8, pp. 1265–1279, Oct 1991.

[45] M. Shreedhar and G. Varghese, “Efficient fair queueing using deficit round robin,” *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 4, pp. 231–242, Oct. 1995. [Online]. Available: http://doi.acm.org/10.1145/217391.217453

Appendix

A Physical properties of microwave links

Microwave hop characteristics is the underlying physical mechanisms why the adaptive bandwidth varies on the microwave link. Some measurements have been done in this area and these measurements will be translated into packet transport characteristics like bandwidth, packet drop, delay variation and also the statistical behavior over time.

Adaptive bandwidth characteristics are based on standard dimensioning and fading properties.

Multipath fading

- Outage is doubled on each level (rule of thumb)
- Approx. 3db between modulations
- Rule of thumb: 3db less doubles outage
- Relatively frequent, but very short outages (<10s, one study states 1s)
- Small/no risk of correlated outages

Rain fading (in comparison to multipath fading)

- Less frequent, but longer outages
- 10s of seconds to few minutes
- In few seconds we can go down to severe degradations (rate of change 1db/s)
- The deeper the degradation the shorter it is
- Correlated outages are likely to happen (within the given availability range)

B Open vSwitch modification

Changes applied to the Open vSwitch source code to achieve stochastic switching:

```
diff --git a/build-aux/thread-safety-blacklist b/build-aux/thread-safety-
blacklist
index 42560df..4db5a84 100644
--- a/build-aux/thread-safety-blacklist
+++ b/build-aux/thread-safety-blacklist
@@ -70,7 +70,6 @@
 \bputchar_unlocked(
 \bputenv(
 \bpututxline(
-\brand(
 \bsetenv(
 \bsetgrent(
 \bsetkey(
diff --git a/ofproto/ofproto-dpif-xlate.c b/ofproto/ofproto-dpif-xlate.c
index 14e8fe2..f0384ba 100644
--- a/ofproto/ofproto-dpif-xlate.c
+++ b/ofproto/ofproto-dpif-xlate.c
@@ -17,6 +17,8 @@
 #include "ofproto/ofproto-dpif-xlate.h"

 #include <errno.h>
+#include <stdlib.h>
+#include <time.h>

 #include "bfd.h"
 #include "bitmap.h"
@@ -240,6 +242,8 @@ static void clear_skb_priorities(struct xport *);
 static bool dscp_from_skb_priority(const struct xport *,
  uint32_t skb_priority,
                                   uint8_t *dscp);

+static bool is_srand_initialized = false;
+
 void
 xlate_ofproto_set(struct ofproto_dpif *ofproto, const char *name,
                   struct dpif *dpif, struct rule_dpif *miss_rule,
@@ -819,26 +823,31 @@ group_best_live_bucket(const struct xlate_ctx *ctx,
                       const struct group_dpif *group,
                       uint32_t basis)
 {
-    const struct ofputil_bucket *best_bucket = NULL;
-    uint32_t best_score = 0;
-    int i = 0;
-
```

```
-    const struct ofputil_bucket *bucket;
+    uint32_t rand_num = 0, sum = 0;
+    const struct ofputil_bucket *bucket = NULL;
     const struct list *buckets;

+    // initialize random seed once
+    if (!is_srand_initialized) {
+        srand(time(NULL));
+        is_srand_initialized = true;
+    }
+
+    // generate a random number in [1, 100]
+    rand_num = (rand() % 100) + 1;
+
+    // Note that weights are not probabilities, but partitions...
     group_dpif_get_buckets(group, &buckets);
     LIST_FOR_EACH (bucket, list_node, buckets) {
         if (bucket_is_alive(ctx, bucket, 0)) {
-            uint32_t score =
(hash_int(i, basis) & 0xffff) * bucket->weight;
-            if (score >= best_score) {
-                best_bucket = bucket;
-                best_score = score;
+            sum += bucket->weight;
+            if (rand_num <= sum) {
+                return bucket;
             }
         }
-        i++;
     }

-    return best_bucket;
+    return bucket;
 }

 static bool
@@ -1990,6 +1999,10 @@ xlate_select_group(struct xlate_ctx *ctx,
struct group_dpif *group)
     const struct ofputil_bucket *bucket;
     uint32_t basis;

+    // The following tells the caching code that every packet in
+    // the flow in question must go to the userspace "slow path".
+    ctx->xout->slow |= SLOW_CONTROLLER;
+
     basis = hash_bytes(ctx->xin->flow.dl_dst,
      sizeof ctx->xin->flow.dl_dst,
      0);
```

```
bucket = group_best_live_bucket(ctx, group, basis);
if (bucket) {
```

C Mininet topology

The microwave mobile backhaul topology implemented as a custom mininet topology:

```
from mininet.topo import Topo

class Backhaul (Topo):

  def __init__(self):
    Topo.__init__(self)

    # Create edge switches
    edge1 = self.addSwitch('s1')
    edge2 = self.addSwitch('s2')

    # Create ring
    ringBottomRight = self.addSwitch('s3')
    ringBottomLeft = self.addSwitch('s4')
    ringTopLeft = self.addSwitch('s5')
    ringTopRight = self.addSwitch('s6')

    self.addLink(ringBottomLeft, ringTopLeft)
    self.addLink(ringTopRight, ringBottomRight)

    # Cross links
    self.addLink(ringBottomRight, ringBottomLeft)
    self.addLink(ringTopLeft, ringTopRight)

    self.addLink(ringBottomLeft, edge1)
    self.addLink(ringBottomRight, edge1)
    self.addLink(ringTopLeft, edge2)
    self.addLink(ringTopRight, edge2)

    # Hosts
    h1 = self.addHost('h1')
    h2 = self.addHost('h2')

    self.addLink(h1, edge1)
    self.addLink(h2, edge2)
```