ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Adrian Friedli

Policy-based Injection of Private Traffic into a Public SDN Testbed

Master Thesis MA-2013-12

Advisors: Dr. Bernhard Ager, Vasileios Kotronis
Supervisor: Prof. Dr. Bernhard Plattner

Acknowledgements

I would like to thank my advisors Dr. Bernhard Ager and Vasileios Kotronis for their help and great support during carrying out my thesis and Prof. Dr. Bernhard Plattner for giving me the opportunity to write this thesis in the Communication Systems Group. Also I would like to thank Hildur Ólafsdóttir for helping me with some debugging.

Special thanks go to my parents, my roommate and her boyfriend for always giving me good advice and support.

Contents

List of Figures

Chapter 1

Introduction

Software Defined Networking (SDN) is an emerging technology. New exciting networking protocols can easily be be experimented with using this technology. Also new routing algorithms may be deployed using SDN enabled switches without having to replace the switch's hard- or software. SDN enabled switches also find application in large virtualization environments. Some large corporations e.g. Google already started deploying SDN enabled switches in their data centers.
For this upcoming technology, we see the need for testing environments. In general, many testing environments exist for SDN applications. But none of these testbeds allow testing with real user traffic. Testing with real user traffic is needed, because modelling real user traffic means just making a simplification of reality. Feedback is only available, when real traffic is involved and the user experiences additional delays and outages of his Internet connection due to the insertion of experimental SDN applications.
But why is it difficult to use real user traffic? User traffic is sensitive, using it in a testbed may cause privacy issues, furthermore the user may have some requirements on availability of his Internet connection. Thus we need to guarantee privacy and availability. Even when privacy and availability are guaranteed, the user still has no incentive to donate traffic to the testbed. Therefore a marketplace is needed. As introduced by Kotronis et al [3] a PAL proxy is available with an important part missing, a traffic Arbitrator, this Arbitrator we wanted to build.
The task of this thesis is to design, implement and evaluate the Arbitrator. Tasks of the Arbitrator are to forward user traffic to the testbed, arbitrate between experiments and shortcut traffic on policy violations. Users should be able to state guarantees for their traffic and experimenters should be able to make requirements to the traffic.
Evaluation consists of a performance analysis, including a comparison with the nature of traffic of a real network and a correctness evaluation.
We have seen that our implemented Arbitrator would be able to handle traffic coming from a typical /23 network such as the TIK network. Moreover we learned some insights about the OpenFlow protocol and that it is a sub-optimal choice to be used in a privacy layer. We had to reimplement some of FlowVisor's behaviour inside the PAL proxy such as it is feasible now to build the PAL proxy inside or around FlowVisor.
Chapter 2 describes technologies used in the thesis, Chapter 3 states the goals and challenges. The design of the implemented Arbitrator is explained in Chapter 4, performance and correctness of the Arbitrator get evaluated in Chapter 5. Chapter 6 presents possible future work and Chapter 7 concludes the thesis with a summary.

Chapter 2

Environment

In this chapter we describe technologies used in this thesis. The OpenFlow protocol, which controllers use to communicate with the switches, is described in Section 2.1. A testing facility for OpenFlow is OFELIA which we describe in Section 2.2. In Section 2.3 we present a extension to OFELIA, the PAL. A different test suite for SDN applications is described in Section 2.4.

2.1 OpenFlow

Openflow is a protocol to control hardware switches. The OpenFlow standard is managed by the Open Networking Foundation and it is called being an enabler for Software Defined Networks. We used OpenFlow version 1.0 [2], but our techniques should be adaptable to newer versions of the protocol.
OpenFlow switches are simple and fast packet forwarding devices, the logic has been moved to an external controller. The switches have a flow table with rules matching flows and each rule may apply a list of actions to the packets of the flow. A rule can match eleven header fields with either an explicit value or a wildcard. A subnet mask can also be used to match the two header fields source and destination IP address. The actions can rewrite header fields and output packets on a specified port.
The switch is connected with the OpenFlow protocol to an OpenFlow controller. If no rule matches a packet the switch forwards the packet to the OpenFlow controller. The controller may then decide what to do with the packet, apply actions to the packet and output it on a specified port, it also may install rules in the switch for matching further packets of the flow.

2.2 OFELIA

OFELIA[1] is an experimental facility based on OpenFlow. There exist several interconnected OFELIA islands at European universities. These islands typically consist of several OpenFlow enabled switches and hosts providing virtual machines.
The physical network provided by the switches is split into slices by FlowVisor. FlowVisor is a tool speaking OpenFlow and can be put between OpenFlow enabled switches and controllers. Thus allowing different controllers to control the switches, each having its own slice. OFELIA uses VLAN-ID for slicing, each experiment has its own VLAN-ID.

2.3 PAL

The goal of the Privacy and Availability Layer (PAL) is to enable experimenters to test SDN applications with real user traffic. And guaranteeing the users privacy and availability for the traffic.
The PAL has been introduced by Kotronis et al [3] and consists of several parts. We have FlowVisor as part of the OFELIA island. Then there is the PAL proxy which we will explain in

[1] OFELIA website: `http://www.fp7-ofelia.eu/`

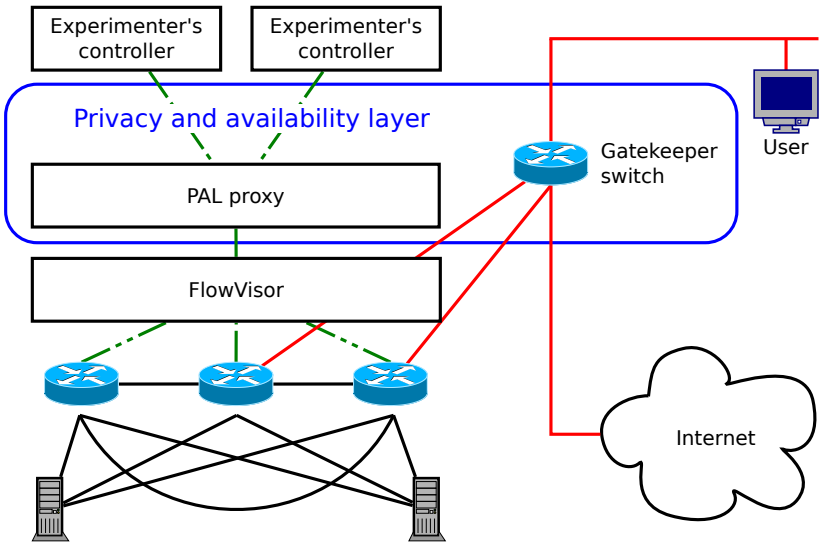

Figure 2.1: Privacy and Availability Layer before introduction of the Arbitrator. As presented by Kotronis et al [3].

Section 2.3.1. And there is a gatekeeper switch[2]. Figure 2.1 shows an example testbed with the PAL, the PAL proxy and a gatekeeper switch.

2.3.1 PAL proxy

The PAL proxy is the entity between FlowVisor and the experimenters. All OpenFlow commands sent by the experimenters to hardware switches get checked by the PAL proxy. The PAL proxy maintains internal flow tables, replicating the flow tables of the switches. The PAL proxy reserves user related flows traversing the testbed. After a successful reservation, the PAL proxy will disallow commands from experimenters, which violate a flow's privacy guarantees. The PAL proxy uses Lookup Tables (LUTs) for the reservation of user flows. We will further describe the LUTs in Section 4.2.1[3].

2.3.2 FlowVisor

In order to separate each experiment from each other and to have multiple controller connections to each switch, we installed FlowVisor between the PAL proxy and the testbed switches. Each experiment has its own controller connection per switch, its own slice and thus its own VLAN-ID assigned.

2.4 Mininet

Mininet[4] is a framework for testing SDN applications. It emulates a network and lets you define a custom network topology consisting of virtual hosts, switches and links.
Mininet isolates hosts through network namespaces provided by the Linux kernel. Whereas all processes of the different hosts run on the same operating system, only with just different virtual network interfaces attached to them and with them having assigned different network addresses. This lightweight virtualization allows Mininet to scale to run thousands of hosts on a modest computer.

[2]Depending on the setup there may be a single gatekeeper switch or two separate switches, one for the user side and one for the Internet side. Our implemented solution supports both setups.
[3]A more detailed description of the PAL proxy is available in [4]
[4]Mininet website: `http://mininet.org/`

Switches are set up using Open vSwitch. Open vSwitch is a production ready virtual switch for Linux and supports among others the OpenFlow protocol. This kernel module based switch allows fast switching with a rate of more than $2\ \mathrm{GB/s}$. Although switching performance does not matter for our correctness evaluation tests.
Mininet allows the user to launch processes on its hosts, which allows easy testing of the topology and the attached controllers. Mininet provides an interactive console which allows a user to control the Mininet environment and launch programs on its virtual hosts. With this mechanism a user can easily run ping, netcat or iperf to send traffic from a virtual host through the virtual network to another virtual host and use tcpdump at any virtual host or switch to inspect the communication path.

Chapter 3

Goals and Challenges

In this chapter we will describe the different goals of the thesis. The main goal is to design and implement a traffic Arbitrator (Section 3.2). In order to do that we first had a look at the goals of the whole PAL (Section 3.1) and at last decided how to integrate it in the PAL. Furthermore the Arbitrator needed some testing. We tested two different aspects, which are a performance analysis with its goals described in Section 3.3 and a correctness evaluation with its goals described in Section 3.4.

3.1 Goals of the PAL

The goal of PAL as a whole is to enable experimenters to experiment with real user traffic without being able to violate policies which each user states for one's traffic. The PAL consists of several parts, one part of it is the PAL proxy. The PAL proxy was already available before the start of this thesis, but it needed some improvements. The PAL proxy's job is to check the OpenFlow commands sent by experimenters to the real hardware switches in the testbed for policy violations. The PAL proxy is just one part of the PAL and we needed an entity forwarding user traffic to the OpenFlow enabled switches in the testbed. For this job we have a gatekeeper switch, which has access to real user traffic. This gatekeeper switch is also OpenFlow enabled and needs a controller. This controller we wanted to build.

3.2 Design and Build of the Arbitrator

The Arbitrator is the controller for the gatekeeper switches. Figure 3.1 shows how the PAL should be extended with the Arbitrator. It has the job to instruct the gatekeeper to forward user flows to the testbed. Each user may define some required guarantees for one's flows. And each experiment may define some requirements on the traffic. The Arbitrator must then chose an experiment to forward the flow to regarding the guarantee and the requirements. It may also chose to shortcut the flow if no other option is available.
The Arbitrator needs to be integrated into PAL. It needs to control the Gatekeeper switch using the OpenFlow protocol and it needs to communicate with the PAL proxy about flows, their policies and possible violations.

3.3 Performance Measurements

High delays makes websites load sluggishly and users notice quickly, especially at university networks where users are used to an Internet connection with round-trip times to a lot of servers with only a few milliseconds.
Google showed, that increasing the delay of loading a website by 500 ms causes them 20% traffic loss and amazon showed that increasing the delay of loading their website caused them

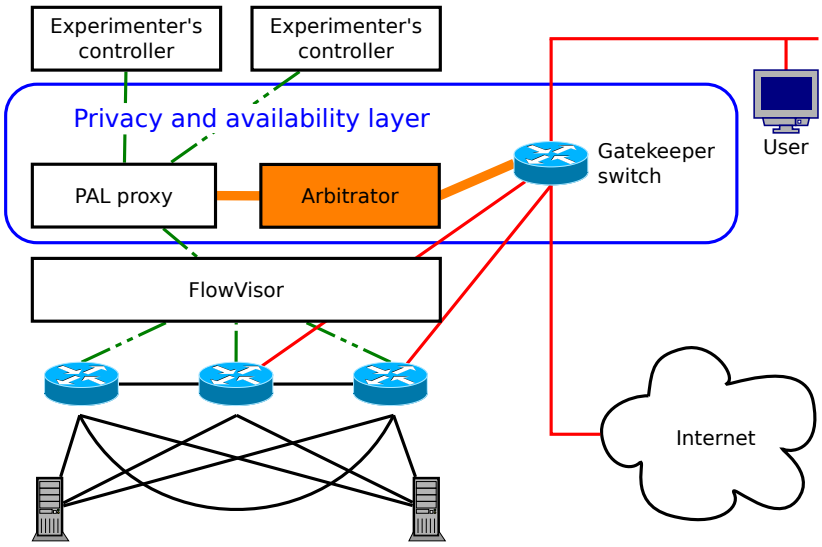

Figure 3.1: Privacy and Availability Layer after introduction of the Arbitrator.

a loss in 1% of sales[1,2].
With todays ubiquitous presence of the world wide web, a lot of effort is made by many Internet companies to improve the experience for users surfing the web. With large content distribution networks these companies try to bring the content as “near” to the user as possible, all in order to make the web “faster”. By reducing the geographical distance and the number of hops between the servers and the user’s PC one is able to lower the latency. Not only many content distribution networks get deployed to reduce latency, but also new protocols are researched on with that goal. For example SPDY [1] by Google. Delay is important, for a regular web browsing session delay will have a large impact on the user experience.
Therefore it is an important goal of this thesis to evaluate how the insertion of an OpenFlow testbed into the user’s Internet connection would impact the user experience, due to the additional delay created by the Arbitrator.
Delay is important for each single user, independent of how many users are participating and thus donating traffic to the testbed. However it does not tell how many users can be handled simultaneously. To better understand how many users can be handled we further study the flow arrival rate as well as the total amount of flows the Arbitrator can handle.

3.4 Correctness Evaluation

There are many threats in the Internet. Computer crime is long not anymore just committed by individuals for personal interests. Organized groups perform criminal actions in the Internet due to commercial interest. Be it to steal confidential information from unencrypted communication for industrial espionage or personal data for committing identity thefts or be it to install malicious software on a user’s computer to gather confidential data from the computers storage, gather access to one’s online banking account or misuse one’s computer to act as a source for further attacks or just for sending unsolicited commercial e-mail.
If an attacker is able to observe or even control a user’s Internet connection, he would be able to perform several attacks. For instance, if an attacker is able to change content of a user’s Internet connection, i.e. if one can inject traffic, an attacker can redirect users to malicious websites and trick them into downloading malicious software, which eventually gets executed by the users computer to install a back-door. If an attacker is able to just observe the user’s Internet traffic, one may be able to steal confidential information from unencrypted communication. Also some form of traffic injection is still possible from other places, even when the attacker can’t manipulate

[1] http://glinden.blogspot.ch/2006/11/marissa-mayer-at-web-20.html
[2] http://www.scribd.com/doc/4970486/Make-Data-Useful-by-Greg-Linden-Amazoncom

traffic at the observing location. Even when an attacker only has access to connection data but not to content, one may still be able to misuse gatherings from this data to perform criminal actions. For example, with such data an attacker is able to learn which websites a user visits, this knowledge may be used for blackmailing, if a user consumes inappropriate content.
In general, researchers are just nosy and we would not expect them doing any harm with user traffic they experiment with. But since we only have limited control over who might run experiments in our testbed, we have to assume anyone in the Internet can register and run potentially malicious experiments.
Because of the presence of such threats in the Internet, it is important for the Arbitrator to work as specified and not to compromise a users privacy and security, when untrusted third-party experimenters should be allowed to perform their experiments in a public testbed. Therefore it is a goal of this thesis to evaluate correctness of the Arbitrator.

Chapter 4

Software Design

Because the gatekeeper switch needs an entity controlling them, we needed to build the Arbitrator. We will discuss the design of the Arbitrator in Section 4.1.
The Arbitrator could either be built into the PAL proxy, or as a separate application. We chose to build it as a separate application (we will motivate that in Section 4.1.1). Therefore the PAL proxy needed to be aware of an external Arbitrator and needed to be extended in that way (we will discuss that in Section 4.2). Furthermore we needed to design an interface between the Arbitrator and the PAL proxy (this will be explained in Section 4.3).
Furthermore we discuss some insights especially about the OpenFlow protocol learned during design in Section 4.4.

4.1 Design of the Arbitrator

In this section we explain the design of the Arbitrator and discuss the decisions taken.

4.1.1 Integration with PAL

There was a design decision we had do to at first. We had two options, should the Arbitrator be integrated into the PAL proxy or should we build it as a separate application? Integrating the Arbitrator into the PAL proxy would certainly be easier, because then there wouldn't be the need for a communication interface. Building it as a separate application on the other hand would simplify testing. Also future refactoring of the PAL proxy would be made easier with a separate application. Therefore we decided to build the Arbitrator as a separate application.

4.1.2 Flows

We define a flow as all packets having the 5-tuple (source IP address, destination IP address, transport protocol, source port, destination port) in common. This definition corresponds to the packets in one direction of a TCP connection. Also UDP streams are caught by this requirement. We also stated a requirement for the time between two sequent packets, which we'll discuss later.
Flows are unidirectional, but most communication patterns are bi-directional. When there is a flow in the backwards direction of another flow and it belongs in some means of their protocol to that flow, we call it the reverse-flow of that flow. In case of TCP or UDP the flow and its corresponding reverse-flow have the same value in the transport protocol header field and the two values in the IP address header fields and the two values in the port header fields are swapped, respectively. The ICMP protocol may be used by routers to inform the sender of a flow about errors, care should be taken to handle these error messages under the same policy as the flow they belong to.
Both flows, a flow and its corresponding reverse-flow, together constitute a so-called bi-flow, in the case of TCP this is equivalent to a connection. Because we must keep information about a bi-flow in the Arbitrator as long as the last rule of a flow belonging to the bi-flow has expired and removed from the gatekeeper switch, we state as an additional requirement for a flow, that in

a bi-flow the maximum time interval between the arrival of two sequent packets must be lower than a specified value.

4.1.3 Policies and Guarantees

A user donating traffic to the testbed may require guarantees for his traffic or parts thereof. Currently we support the following guarantees: “direct”, “no-sniff” and “allow”. The “direct” guarantee means not to deliver this part of the traffic to the testbed at all, a shortcut rule will be installed as flows appear in the testbed. The “allow” guarantee does not warrant for anything and the “no-sniff” guarantee protects the content of the traffic. Another requirement of the user may be, not to allow the testbed to do header space rewriting. A policy includes a statement for matching a flow, a guarantee the PAL proxy must warrant for and whether the testbed is allowed to rewrite the header space.

4.1.4 Forwarding flows to experiments

When the Arbitrator installs a flow mod rule to the gatekeeper switch for forwarding a flow to the testbed, it adds two actions to that rule, the first action is to add a VLAN-ID to the packet, the second action is to output the packet on the port facing the testbed. The Arbitrator installs an opposite rule for flows leaving the testbed, also with two actions, the first is to remove the VLAN-ID and the second is to output the packet to the outside port.

4.1.5 Handling of new flows

The Arbitrator works purely event based. All its actions are triggered from events from the outside. Events it will react upon may either come from the gatekeeper switch or from the PAL proxy. The interface to the PAL proxy with the events that the Arbitrator will handle will be described in Section 4.3.
There are two possible events from the gatekeeper switch, the Arbitrator has to react to. First, there is a packet-in event, whenever a new flow traverses the gatekeeper switch, and second there is a flow removed event when an installed flow rule was removed due to its idle timeout.
A new flow has to be handled, whenever the Arbitrator receives an OpenFlow packet-in event packet from the gatekeeper switch.
When the flow is leaving the testbed, a request (requests will be explained in Section 4.3) is sent to the PAL proxy to get to know the original header space of the flow before header space rewriting might have changed the flow’s header space in the testbed.
Then the Arbitrator uses the header space of the flow to look it up in its internal flow table. If no entry was found a new one is created and for flows entering the testbed, an experiment has to be chosen to send the traffic to (arbitrating between experiments is described in Section 4.1.7). For flows leaving the testbed, the experiment is already known from the VLAN-ID in the packet header.
For flows entering the testbed the Arbitrator looks up the policy of that flow and asks the PAL proxy if forwarding the flow to the testbed would violate the given policy. If its conforming to policy, the action for that flow will be forwarding to the testbed, else the action would be to shortcut the given flow. Flows leaving the testbed are always forwarded to the outside.
The Arbitrator then outputs the packet received by the packet-in event according to the chosen action and installs a flow mod rule in the switch, for forwarding future packets of that flow. In every case, the Arbitrator updates its internal flow table, with header space information, chosen experiment, action taken, and which rule is already installed in the switch. Figure 4.1 and Figure 4.2 show the sequence of operation in handling new flows with more details.

4.1.6 Flow table

When a flow is forwarded to an experiment in the testbed, the corresponding reverse flow should also be forwarded to the same experiment. The Arbitrator needs to be able to find an existing flow, when its reverse flow traverses the gatekeeper switch. In order to do that, we introduced

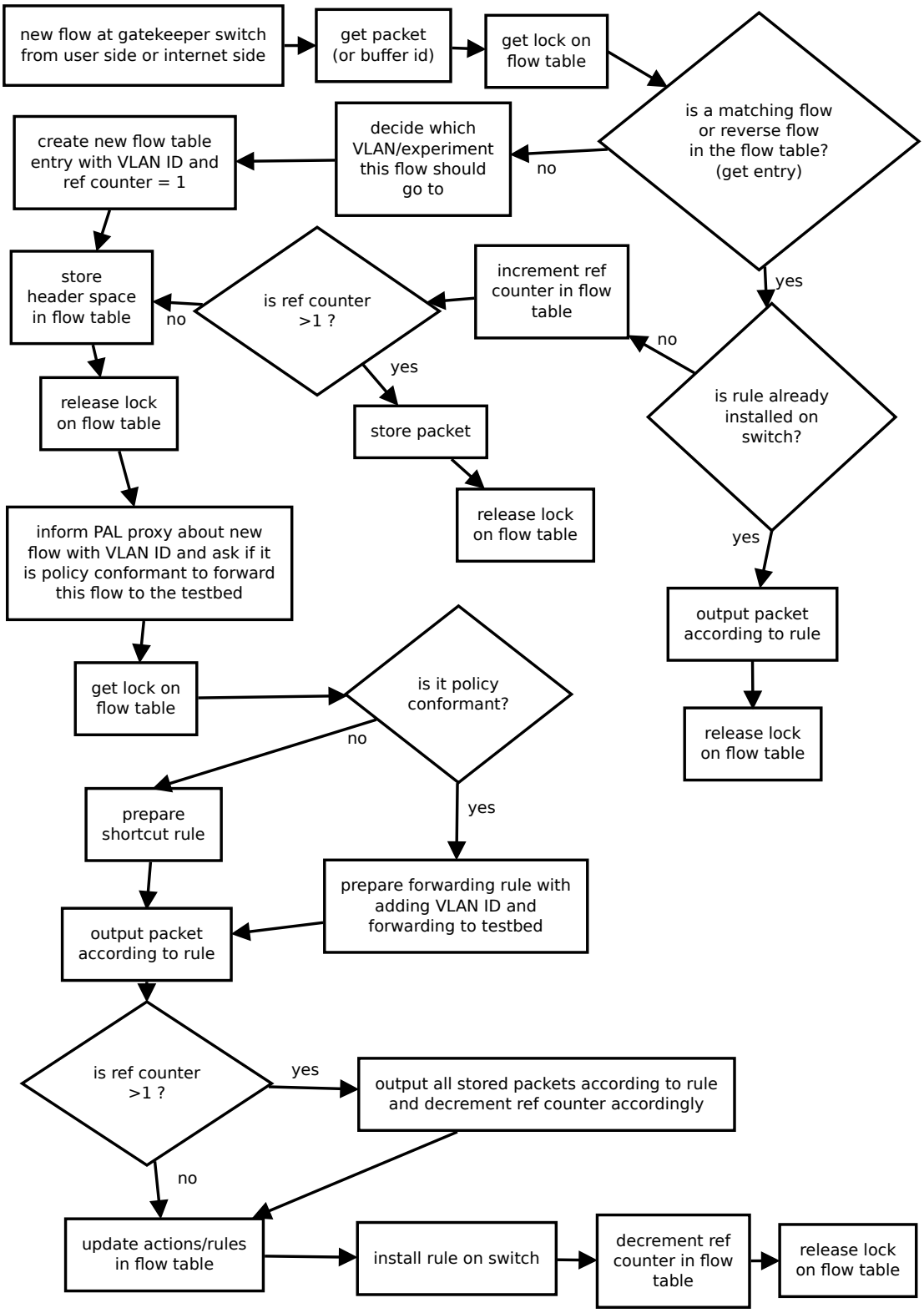

Figure 4.1: Flow chart for handling new flows entering the testbed.

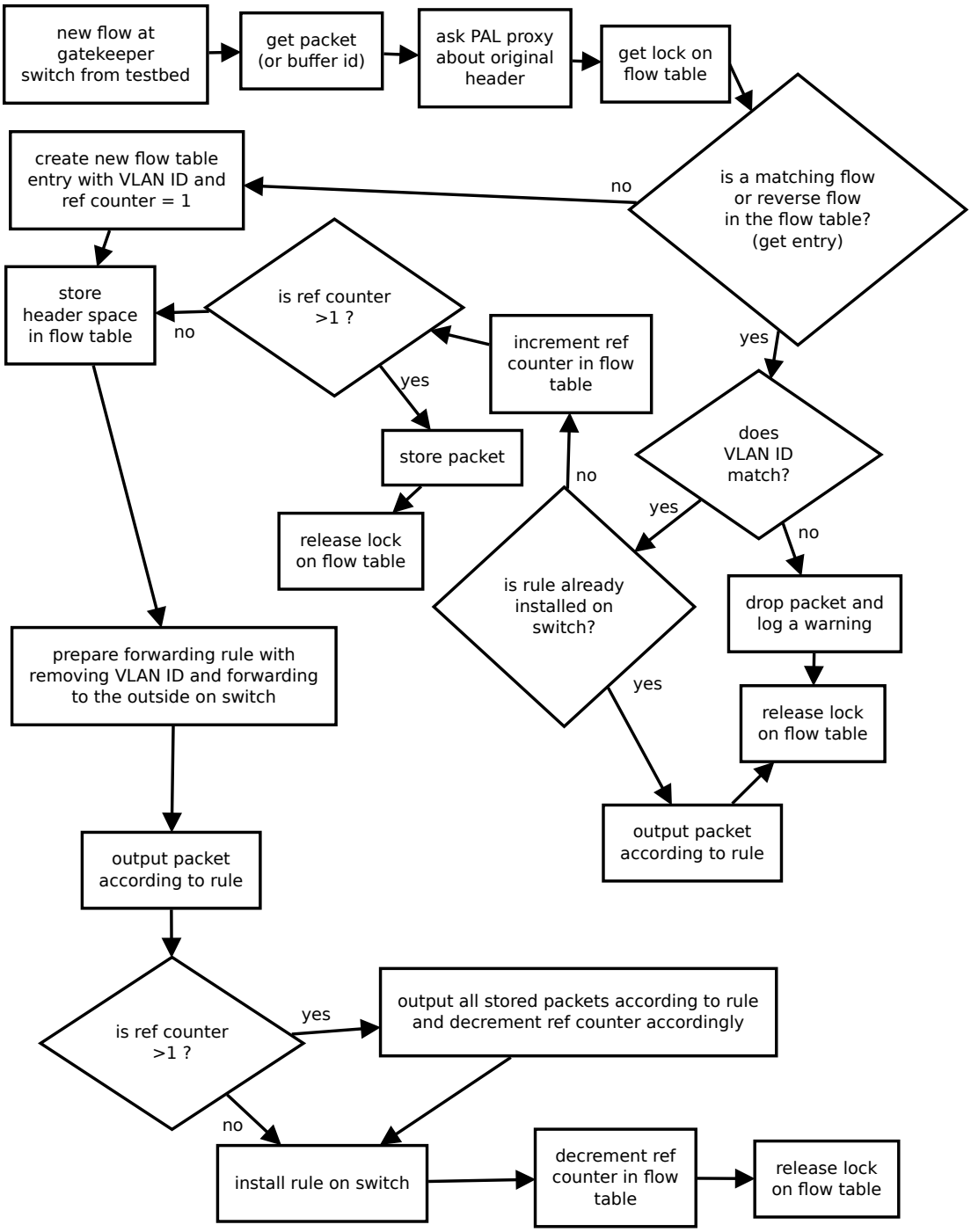

Figure 4.2: Flow chart for handling new flows leaving the testbed.

experiment	guarantee	hs rewriting	information for flow ⋆	information for reverse flow ⋆

(a) Information stored for each bi-flow.

⋆	action for flow	information for part entering testbed •	information for part leaving testbed •

(b) Information stored for each flow, twice for each bi-flow.

•	hs of flow part	rule is installed	refcount	packet store

(c) Information stored for each part of a flow, four times for each bi-flow.

Table 4.1: Information stored in a flow table entry.

an internal flow table. The flow table consists of flow table entries, for each bi-flow traversing the gatekeeper switch, a flow table entry is created.
Table 4.1 shows the information stored in a flow table entry. It consists of the experiment (VLAN-ID), the guarantee and if header space rewriting is allowed, actions taken for the flows in each direction, header space when entering the testbed and leaving the testbed for each flow. For each part of a flow, it is stored in the flow-table if a corresponding rule is already installed in the gatekeeper switch. More information stored for each part of a flow are a reference count for handling concurrency issues, and a reference to a list of stored packets also for handling concurrency issues.

4.1.7 Arbitrating between experiments

Each experiment has requirements to the policies of flows sent to it. One may require a maximum guarantee, be it to be able to access all the payload of each packet of the flow or the header may be enough. Or one may need to rewrite header space. Whenever an experiment has to be chosen to send the flow to, we create a list of experiments the guarantee and the ability to rewrite header space is usable to and take a random experiment from that list.

4.2 Changes made in the PAL proxy

This section explains the changes needed in the PAL proxy to integrate the Arbitrator into the PAL and gives an overview over important bugs found and fixed or worked around in the PAL proxy.

4.2.1 Reserve flow space in LUTs and request original header

For implementing the requests to reserve flow space in the PAL proxy and to get the original header space of a flow, not much had to be done in the PAL proxy, most of it was already there.
The PAL proxy has a topology of nodes. Every node in the topology has a corresponding switch and contains a flow table and a LUT. The flow tables in the PAL proxy's nodes resemble the flow tables in the corresponding switches. The LUT entries track user related flows as they traverse the testbed and reserve their flow spaces. They contain references to previous and next LUT entries, current header space of the flow when it enters the node and the original header space of the flow.
In order to reserve a flow, we create a LUT entry in the first node and perform a path validation. The flow space is then propagated through the virtual topology and linked LUT entries are created accordingly.
Because the last switch's header space rewriting was not recorded in any LUT, we had to extend the topology and include the gatekeeper switch. The gatekeeper node doesn't need a flow table. Looking up a flows original header space now involved just looking up the flow in the gatekeeper node's LUT.

4.2.2 Intersecting flow space workaround

We installed FlowVisor between the PAL proxy and the testbed switches, thus rules sent by an experimenter first pass the PAL proxy and then pass FlowVisor until they reach the switch. FlowVisor enforces each experimenter to use its own header space, which is called a slice. OpenFlow rules sent to the switch contain a match and a list of actions. FlowVisor intersects the header space of the match and the slice, if the result is the empty header space, the rule is rejected, otherwise the result is used as the new match. If an action would rewrite header space to be not part of the slice, the rule is rejected[1].
There was a design error in the PAL proxy, FlowVisor changes matches before it passes them to the switches, this was not considered in the PAL proxy. We used the VLAN-ID field for slicing rules from the experimenter. When an experimenter sends a rule with a wildcard in the VLAN-ID field in the match, the PAL proxy would accept it, update its internal flow table accordingly and forward it to FlowVisor. FlowVisor will then accept it and change the VLAN-ID field in the match to the experimenter's VLAN-ID. If another experimenter sends a rule with the same match with the wildcard in the VLAN-ID field, the PAL proxy will update its internal flow table and modify the first experimenter's entry and passes it to FlowVisor. FlowVisor will then accept this rule too and change the VLAN-ID field in the match to the second experimenter's VLAN-ID.
This gives us two problems. First the internal flow table of the PAL proxy is not in sync with the flow table in the switch, because FlowVisor changes rules and these changes are not considered in the PAL proxy. Second, entries in the internal flow table of the PAL don't hold associations to the experimenter who created them, thus allowing an experimenter to change another experimenter's flow table entries.
To work around these problems we enforced VLAN-IDs in rules from the experimenters in the PAL proxy. Matches in rules containing a wildcarded VLAN-ID were changed to contain the experimenter's VLAN-ID, rules with matches already containing the experimenter's VLAN-ID were accepted as such and rules with matches containing another VLAN-ID were rejected. This workaround solves both problems, rules wont get changed anymore in the FlowVisor and each flow table entry is assigned to an experiment due to the value in its VLAN-ID field.
This workaround limits the possibilities of experiments to run. We are only able to assign one VLAN-ID per experiment and experiments can't share VLAN-IDs. Both are possible in OFELIA but rarely used.

4.2.3 Handle error messages and rejected rules

The OpenFlow protocol specifies, that a switch should send no response, when a flow mod message succeeded, only in the case of an error it should send an error message. Flow mod messages from experimenters cause the PAL proxy to immediately update its internal flow table. Later error messages generated by unsuccessful flow mod messages get ignored by the PAL proxy itself, and just forwarded to the experimenter. Such that the flow table in the switch and the one in the PAL proxy get out of sync.
An OpenFlow barrier request sent to a switch, makes the switch to process all so far unprocessed messages and then respond with a barrier reply. Thus making all error messages caused by unsuccessful flow mod messages sent before the barrier request being sent before the barrier reply.
We used OpenFlow barrier request to fix this problem. After each flow mod message, we send a barrier request and wait for the barrier reply. In between, the change to the flow table is in a transaction state, it gets rolled back, when an error message is received, and it gets committed, when the barrier reply is received. Because other messages should not operate on inconsistent flow table entries, all other requests get queued.

4.2.4 Input port is part of flow space

In order to do policy violation checking, the PAL proxy emulates an OpenFlow enabled switch and maintains a copy of the switch's flow table. OpenFlow matches, as part of a switch's flow table, are designed to match twelve fields[2]. Eleven of those twelve fields match fields in the

[1] At least with FlowVisor version 1.0 and newer.
[2] OpenFlow Switch Specification 1.0 [2] in Table 2 on page 3.

Ethernet frame, as sent through the network. The other field is to match the physical input port a packet enters the switch. Flow space in the PAL proxy is also represented by these twelve fields.
The PAL proxy maintains LUTs to reserve user flow spaces, it does this by inserting a LUT entry for a switch, then checks flow table entries of that switch to find the next switch in the path of that flow. This is done recursively until a gatekeeper switch is reached.
The PAL proxy didn't consider to update the input port in the LUT's flow space as the flow travels through the testbed. This caused matches in the flow tables without wildcards in the input port field to behave incorrectly.
We fixed this by updating the input port field in the LUT entries on each hop.

4.2.5 Timeout scheduler

Because rules in the PAL proxy may expire due to an idle timeout and may cause a policy violation when they get removed, the PAL proxy needs to watch the timeout of the rules. OpenFlow switches collect statistics about flows, these flow statistics contain a time value to idle before expiration.
The PAL proxy schedules an action to get statistics of a rule some seconds before its idle timeout, and reschedules if needed. If the rule would expire soon, it does a path validation, if that fails, it tells the Arbitrator to install a shortcut rule matching the flow[3].
In the original version of the PAL proxy this was planned, but the implementation was not finished. Scheduling was implemented, but retrieving statistics from the switch was missing. One did not consider the slicing of FlowVisor, each experimenter can only get statistics about rules in the slice he has access to. In order to not have to find out the correct experiment for each rule and omit the need to filter out statistic responses to the experimenters we introduced an admin slice in FlowVisor. The admin slice consists of the all wildcarded flow space, thus it has access to all rules in the switches. The PAL proxy then uses the admin slice's connection to retrieve statistics about the rules.

4.3 Interface between PAL proxy and Arbitrator

Because we chose the Arbitrator to be a separate application which is not included in the PAL proxy (we did discuss that in Section 4.1.1), but they still depend on each other and need to exchange information at some point, we needed to design a communication interface between the two.
We want to be able to handle new events from the gatekeeper switch while having requests from the Arbitrator to the PAL proxy from earlier events still pending. In the case when some requests are pending, new events not needing communication with the PAL proxy should get handled immediately by the Arbitrator, i.e. the Arbitrator should not block while waiting for requests to complete. Therefore we chose an asynchronous request and response communication model for the communication between the Arbitrator and the PAL proxy.
In order not to let the PAL proxy idle wait for a new request after a response has been sent and therefore enable it to quickly work off requests, we chose to use pipelining. Even when there are pending requests from the Arbitrator to the PAL proxy, new events needing communication with the PAL proxy will cause the Arbitrator to send requests, which then need to get queued at the PAL proxy.
We wanted a future PAL proxy with a multi-threaded design to be able to handle requests out of order. Thus we had to make sure, the Arbitrator will handle responses correctly, when they arrive at a different order as their corresponding requests. On the requesting part, each request is assigned an incrementing request id. This request id is then included in the request data which is sent to the responding part. On the requesting part the request id is then used as a key for storing the request data in a pending requests lookup table. Together with the request data itself a handler function is stored in this table. When the requested action has been performed and there is a response, it is sent to the requesting part and always includes the request id. When a response is received by the requesting part it takes the request id and looks up the

[3] More on that can be found in [3].

entry in the pending requests lookup table and calls the handler function included in the entry with the response payload as an argument.

There are three different types of requests performed by the Arbitrator to the PAL proxy. Request to inject a flow into the testbed, requesting the original header of a flow leaving the testbed, and informing the PAL proxy about removal of a flow. In addition one type of request is performed by the PAL proxy to the Arbitrator, namely requesting the Arbitrator to shortcut a flow.

The rest of this section describes the details of the four types of requests and the choices we made for the implementation.

4.3.1 Request flow injection

Whenever there is a new flow to be injected to the testbed, the Arbitrator needs to ask the PAL proxy if injecting the flow into the chosen experiment causes a policy violation. “Flow injection” we named this request type. The response to such a request states if it’s either policy violating or not.

The payload of this request type contains the VLAN-ID of the experiment we want this flow to be injected, the 5-tuple header, the source where this flow is coming from, either “user” or “Internet”, and the guarantee, we want this flow to be warranted for.

The response to such a request is either the “forward” statement or the “shortcut” statement.

The “forward” statement tells the Arbitrator that the PAL proxy successfully reserved this flow space in its lookup tables and it is now safe to forward this flow to the testbed, the PAL proxy now also expects to be informed about removal of this flow to be able to release system resources.

The “shortcut” statement tells the Arbitrator that the PAL proxy was not able to reserve this flow space in its lookup tables, meaning that forwarding this flow would cause a policy violation and that the Arbitrator should shortcut this flow instead of forwarding it to the testbed.

4.3.2 Request original header

Whenever a new flow is leaving the testbed, the Arbitrator needs to look it up in its flow table. If the experiment handling this flow is allowed to do header space rewriting, the Arbitrator needs to know the header space of the flow before it entered the testbed. The PAL proxy reserves each user flow and keeps track of header space rewriting actions within its LUTs. All the information about original header spaces is stored in the PAL proxy. The Arbitrator can use the request type named “original header” to retrieve this information from the PAL proxy.

The payload of this request type contains the VLAN-ID, the 5-tuple header and the destination, where the flow is headed to, either “user” or “Internet”.

The response to such a request contains the VLAN-ID and the 5-tuple header of the flow when it entered the testbed.

4.3.3 Inform PAL proxy about deletion of a flow

The PAL proxy uses system resources for reserving flow space in its LUTs. Each LUT entry uses some memory, additionally policy checks demand computation time depending on the number of LUT entries installed. In order to not exhaust system memory we want to release these system resources as soon as possible. The gatekeeper switch informs the Arbitrator as soon as a flow rule expired due to the idle timeout and was removed from the switch’s table. The Arbitrator then updates its internal flow table. If a rule just removed belongs to a flow traversing the testbed the Arbitrator sends a request of the type “flow remove” to the PAL proxy to inform it to give up the flow space reservation of that flow and free used system resources.

Because no response is needed by the Arbitrator for this type of request, the PAL proxy does not send responses to such requests. Therefore these requests are not put in the pending requests lookup table.

The payload of the request contains the VLAN-ID, the 5-tuple header and the source where the flow originally was coming from, either “user” or “Internet”.

4.3.4 Request Arbitrator to shortcut flow

When the PAL proxy encounters a policy violation for an existing flow space reservation, it has to be able to inform the Arbitrator that it is not safe anymore to forward this flow to the testbed. This can happen when a rule times out on the switch and gets caught by the timeout scheduler. The PAL proxy then sends a request type named “shortcut” to the Arbitrator to inform it about such a policy violation.
Because no response is needed by the PAL proxy for this type of request, the Arbitrator does not send responses to such requests. Therefore these requests are not put in the pending requests lookup table. And because this is the single request type performed by the PAL proxy, we could omit the implementation of a pending requests table on the PAL proxy side.
The payload of the request contains the VLAN-ID, the 5-tuple header and the source, where the flow is originating from, either “user” or “Internet”.

4.3.5 Implementation choices

Because we were already using the POX library at both ends, we chose to use POX’ messenger component. The messenger component provides bi-directional channels. Because of our request and response communication model, we chose to use two channels, a request and a response channel. Requests are sent to the request channel and if there is a response to a request, the other part sends the response to the response channel.
The POX’ messenger component is built using a TCP connection and sends JSON encoded data. For the time being only the server part was implemented. Therefore on the PAL proxy side we had to extend the messenger component to act as a client and connect to a server.

4.4 Insights

As we have seen in the previous Sections, the OpenFlow protocol is a sub-optimal choice for the purpose of a privacy layer, its design makes it difficult to handle every case correctly. Furthermore we had to rebuild parts of FlowVisor in the PAL proxy. In this Section we discuss more insights learned while designing the Arbitrator and debugging the PAL proxy.

4.4.1 Insights in OpenFlow

As we have seen in Section 4.2.3, the OpenFlow protocol has no support to acknowledgement flow mod messages a controller sends to the switch. In order to be able to keep the flow tables in the switches and in the PAL proxy in sync, a mechanism for acknowledging these flow mod messages was needed. We had to use barrier requests for that purpose, which certainly has an impact on performance.
As discussed in Section 4.2.5, handling of idle timeouts as defined in the OpenFlow protocol is very limiting for our purpose. There is some kind of asymmetry in handling of idle timeouts of rules in the OpenFlow protocol. Rules are installed by the controller, but get removed by the switch as soon a timeout occurs and the controller is notified after the removal of the rule. This issue was already known from the previous version of the PAL proxy, but the proposed solution did not consider slicing of the FlowVisor.

4.4.2 Insights in FlowVisor

As seen in Section 4.2.2 FlowVisor does make changes to rules sent from the experimenters’ controllers to the switches. We had to resemble these changes in the PAL proxy and therefore duplicating behaviour of FlowVisor inside the PAL proxy. The main use of FlowVisor how it is used currently in the PAL, is to multiply a controller connection to different experimenters.

Chapter 5

Evaluation

Latency is an important factor in the user experience of one's Internet connection, as we have discussed in Section 3.3. One goal of our performance analysis is to measure the delay added by inserting the Arbitrator to one's Internet connection, so we will be able to estimate the impact on the user experience.

In order to minimize measurement artifacts, we needed a controlled environment for measuring performance. It needs to be adjustable and it needs to be able to create a specific load. Deploying the PAL setup in ETHZ's OFELIA testbed would have been possible but would have created too much effort to generate load and measure performance.

Additionally there is no standard tool readily available for our purpose to generate a specific load on an OpenFlow controller and to measure its performance. Therefore we decided to build such a tool by our own. The tool generates OpenFlow packet-in events and handles responses accordingly. This Arbitrator benchmarking tool is fast enough not to interfere with the measuring results. Thus testing the tool itself for its performance was required.

Additionally we evaluated the correctness of the implemented Arbitrator, which is discussed in Section 5.5.

All the Performance tests were done on a Lenovo Thinkpad T410 with an Intel Core i7 CPU running at 2.67 GHz and 4 GB of Memory. Furthermore to get a stable and predictable environment we disabled hyper-threading in the system's BIOS to make sure not to encounter scheduling interferences. We also did configure the CPU not to enter lower C-States and used the performance CPU governor in the Linux kernel. Both techniques would reduce power usage of the system when idling, but could fudge our test results. The Operating System used was Ubuntu Linux 12.04 LTS. The Linux kernel was version 3.2.0 and the installed Python version was 2.7.3.

5.1 Environment

We wanted to measure the performance of the Arbitrator. Therefore we had to figure out, how we could take the Arbitrator out of its intended environment and put it into a benchmark environment. Since the Arbitrator is a standalone application, which was designed as a modular part of the PAL from the beginning, with only a few connections to other parts of the PAL. These connections are the best points to measure performance at because measuring at these points includes measuring communication overhead and it is not required to instrument the Arbitrator.

We had a closer look at the outside connections of the Arbitrator and discovered, the only relevant in- and outputs of the Arbitrator is the connection to the gatekeeper switch (Figure 5.1 shows a simplified version of the Arbitrator and the gatekeeper switch in their intended environment). There is also a connection from the Arbitrator to the PAL proxy (not depicted in the Figure) and we will discuss its impact in Section 5.3. Because of this single relevant connection, where OpenFlow is used, it is sufficient to replace the gatekeeper switch by a benchmarking tool which we call Arbiterbench.

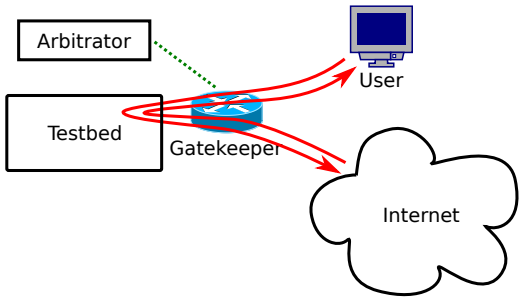

Figure 5.1: Schematic depiction of the flows traversing the testbed.

5.2 Arbiterbench

We had to figure out, which OpenFlow packets the gatekeeper switch sends to the Arbitrator, therefore we looked at the flows traversing the testbed. When a bi-flow is sent through the testbed it traverses the gatekeeper switch four times in total. First a flow initiated by the user traverses the gatekeeper switch when it enters the testbed and traverses the gatekeeper switch a second time as it leaves the testbed. When the destination host in the Internet responds, it will generate a second flow, a so called reverse-flow with IP source and destination addresses swapped and, if applicable, source and destination ports swapped. This reverse-flow traverses the gatekeeper switch as it enters the testbed and again as it leaves the testbed. The red arrows in Figure 5.1) represent these flows.
In the Arbitrator's intended environment the arrival of the first packet of each flow at the gatekeeper switch causes the switch to send an OpenFlow packet-in event packet, which makes it two for each flow and four for each bi-flow. All future packets of a flow will not cause new packet-in events, as flow-mod rules will then be installed in the gatekeeper switches and all packets of that flow will be forwarded accordingly.
The four packet-in events are a worst-case scenario. This can actually be reduced by half, if we not only install the flow-mod rule matching the flow just encountered but also we would pre-install a flow-mod rule for the corresponding reverse-flow on the gatekeeper switch, because the headers for matching the reverse-flow are already known at this point. It can even be reduced to a quarter, in the case where no header space rewriting is done inside the testbed[1] where we will be able to install all four required flow-mod rules at once, but one may not forget to request insertion of the reverse-flow to the testbed at the PAL proxy if applicable.
Our Arbiterbench simulates this behaviour and generates an OpenFlow packet-in event packet, as a new flow initiated by the user entering the testbed at the gatekeeper switch would do, and sends it to the Arbitrator. The Arbitrator then will handle this simulated new flow event and will respond to the benchmarking tool accordingly with a OpenFlow packet-out and a flow-mod command packet. The benchmarking tool will then measure the time delay between the packet-in event and the packet-out command. For each packet-out command received, the Arbiterbench will generate a new OpenFlow packet-in event packet representing the bi-flow's next traversal of the gatekeeper. This is done for every packet-out except for the one representing the fourth traversal of a bi-flow.
The Arbiterbench generates the first OpenFlow packet-in event packets with a constant rate. Additionally it will generate packet-in events after receiving a packet-out command from the Arbitrator, if required to. So for each packet it generates with a constant rate, it will generate three others in addition, sequentially.
The Arbiterbench sends an OpenFlow packet-in event packet, calculates the time interval between sending time of this and the next such packet and sleeps this amount of time. In the meantime another thread handles incoming packet-out and flow-mod commands, measures time delays and generates new packet-in events accordingly.
It is designed as a standalone application. Because we had experience with the POX library and estimated it is fast enough for that purpose we used the same patched version of the POX library as the PAL proxy was using to build the Arbiterbench. In a later point we showed it is indeed

[1] The experiment's traffic requirements tell if header space rewriting is performed in the testbed.

fast enough for our purpose. In addition we re-used some code of the PAL proxy, especially for doing the northbound communication, the part which is responsible for the communication with an OpenFlow controller.
In the hope to reduce CPU usage and make it possible to test higher flow rates, we also tested sending in batches of ten packets, meaning sending ten packets and then wait an equivalent longer time interval until the next ten packet-in events should be sent. Testing with batches showed artifacts, the delay showed a mode at about 65 ms either produced by the benchmarking tool itself or by the Arbitrator. Thus, it was for our measurement purposes not usable and we deactivated it.
We have verified that the performance of the Arbiterbench is sufficient and it's measurement results are not influenced by some co-processing issues on the computer. We have seen that our benchmarking tool generates packets fast enough, because it always caused considerably less CPU usage than the Arbitrator. The Arbiterbench and Arbitrator are only able to use one CPU at a time and the setup was running on a dual-core system.

5.3 Connection to the PAL proxy

We wanted to test both the delays inside the Arbitrator and it's communication delays, the ones between the gatekeeper switch and the Arbitrator as well as the ones between the Arbitrator and the PAL proxy. We didn't require the PAL proxy to be fully functional for the tests. Having a running PAL proxy would have required us to create a model for testing the PAL proxy's internal structures and test results would strongly depend on that. Therefore we built a dummy PAL proxy for the performance analysis task. This dummy PAL proxy responds to every flow insertion request with the "forward" statement, which means every flow should be forwarded to the testbed. And it responds to every original header space request with the header space the request was associated with, such as no flow is subject to header space rewriting in the testbed.

5.4 Performance Evaluation Results

5.4.1 Time delay

Figure 5.2 shows the time between the generation of the first packet-in event and the reception of the corresponding packet-out event for different flow rates. The peaks in time delay which are visible in all the curves stem from the internal data structure of the Arbitrator, because we used a Python dictionary, which gets resized every now and then. This could be avoided with "warming up" the data structure before use.
The curve at the top shows that $500\ \frac{\text{flows}}{\text{s}}$ create an overload on the Arbitrator. The Arbitrator is not fast enough to handle this rate and the backlog is increasing and with it the total delay. Even in the time intervals between higher delays due to dictionary resizing the delay is rising, from this we conclude even a "warmed up" data structure would create an overload with this flow rate.
The plots of the other three packet-in/packet-out delays of each bi-flow look similar, the first packet-in/packet-out delay is only a bit higher than the other, thus we omitted them for brevity.
Because we wanted to know the total delay added by the Arbitrator to a bi-flow, we summed up the 4 measured delays and plotted it in Figure 5.3. Because the peaks are shifted in time for the different packet-in events of a bi-flow, the top curve showing the overload case looks much flatter here.
Figure 5.4 shows a zoomed version of Figure 5.2 which shows the time delay of handling the first packet-in in a bi-flow, where we only include the lower rates and look only at the first 5000 flows. The cyan line with a rate of $50\ \frac{\text{flows}}{\text{s}}$ shows the most delay, and the green one with a higher rate of $200\ \frac{\text{flows}}{\text{s}}$ shows the lowest delay. We expected these to be about equal. We turned off power saving features of the CPU as described in Section 5.2, therefore we assume it has to do with either scheduling of the Linux kernel or other power saving features, which can't be turned off.
Figure 5.5 shows a zoomed version of Figure 5.3 which shows the total time delay of handling a bi-flow, also here we omit the high flow rate and only look at the first 5000 flows.

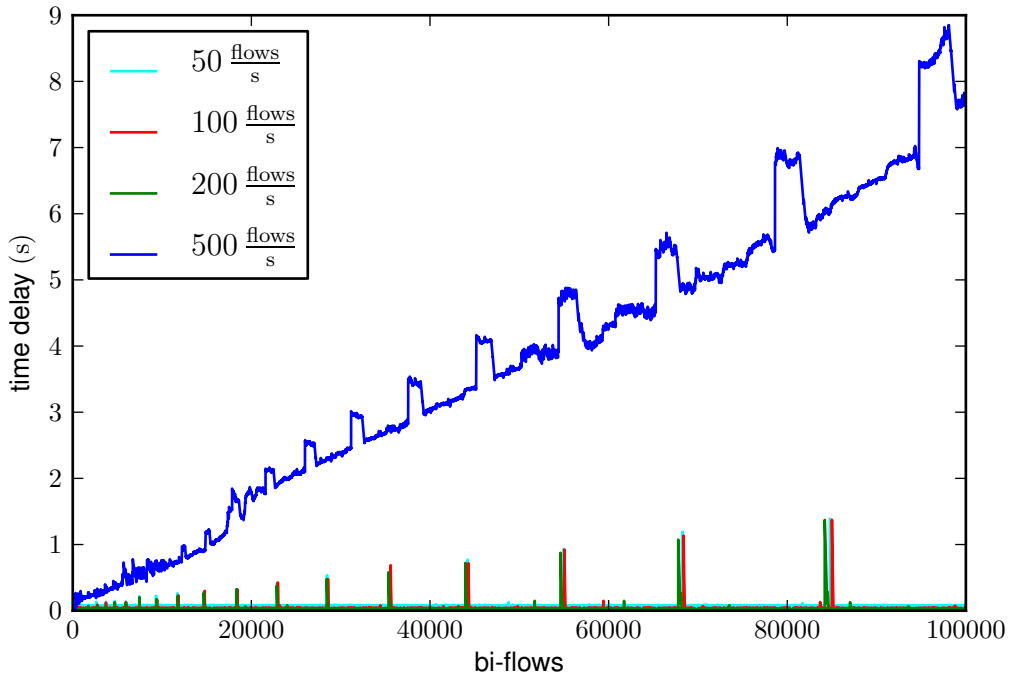

Figure 5.2: Time delay between packet-in and packet-out for the first event of each bi-flow traversing the testbed. The blue curve shows an overload of the Arbitrator. The peaks result from time needed reorganizing the internal data structure.

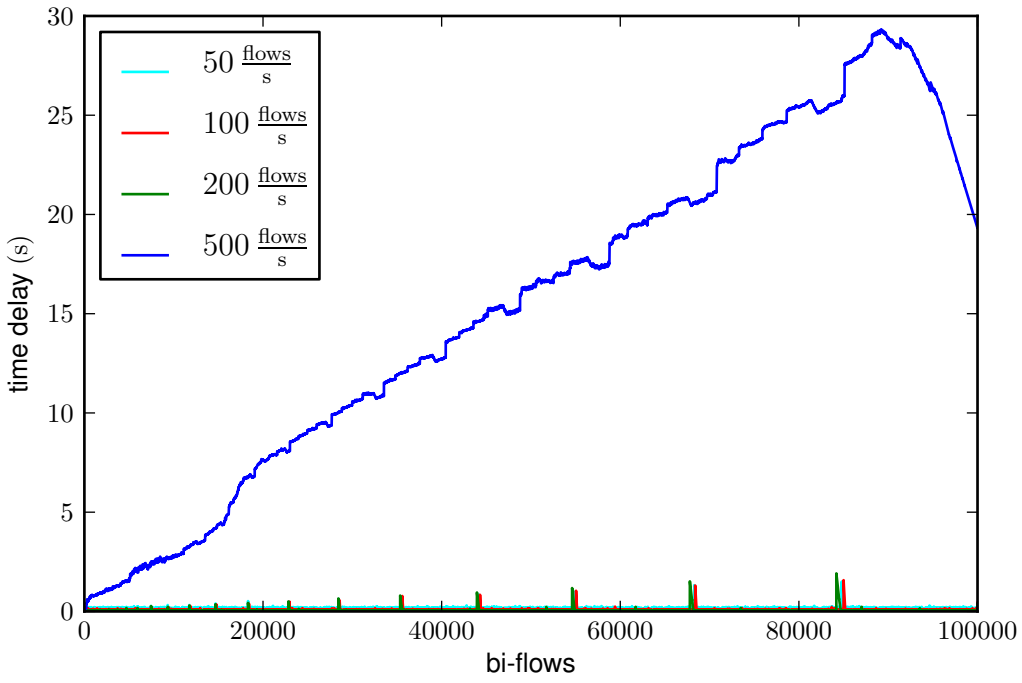

Figure 5.3: Time delay between packet-in and packet-out summed up on all events of a bi-flow traversing the testbed. Again, the blue curve shows the overload of the Arbitrator and the peaks from reorganizing the internal data structure are visible.

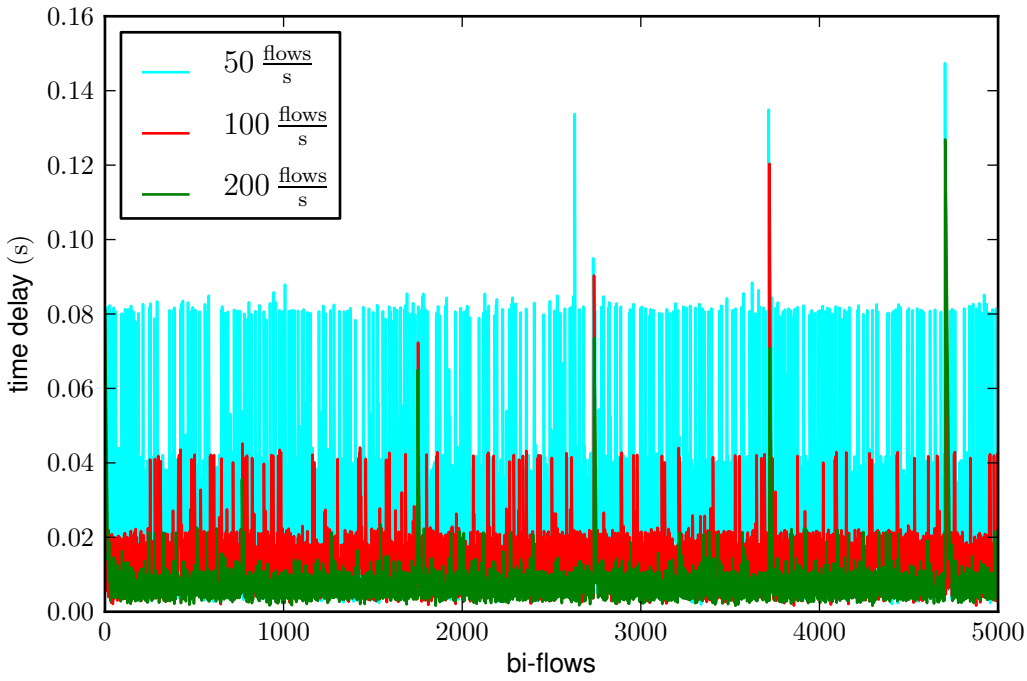

Figure 5.4: Time delay between packet-in and packet-out for the first event of each bi-flow traversing the testbed. We look at fewer flows and the overload-creating flow rate has been omitted in this plot. We see higher delays with lower flow rates, we ruled out the most obvious causes for them and didn't further evaluate it in order to not waste more time.

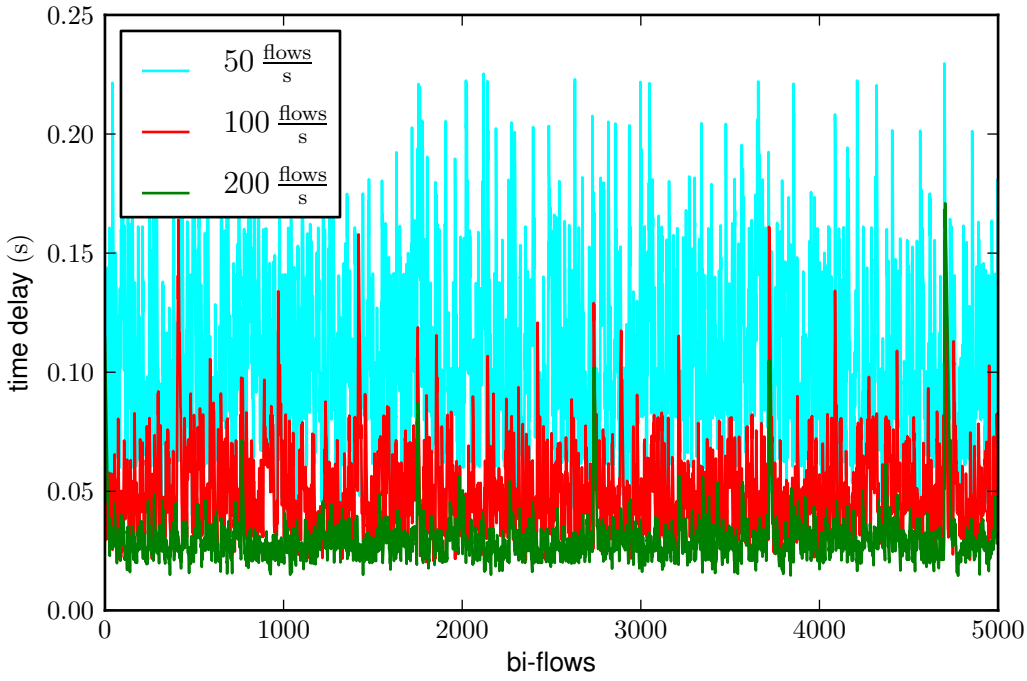

Figure 5.5: Time delay between packet-in and packet-out summed up on all events of a bi-flow traversing the testbed. Also here we look at fewer flows and we omitted the overload-creating flow rate. The higher delays with the lower flow rates are also visible here.

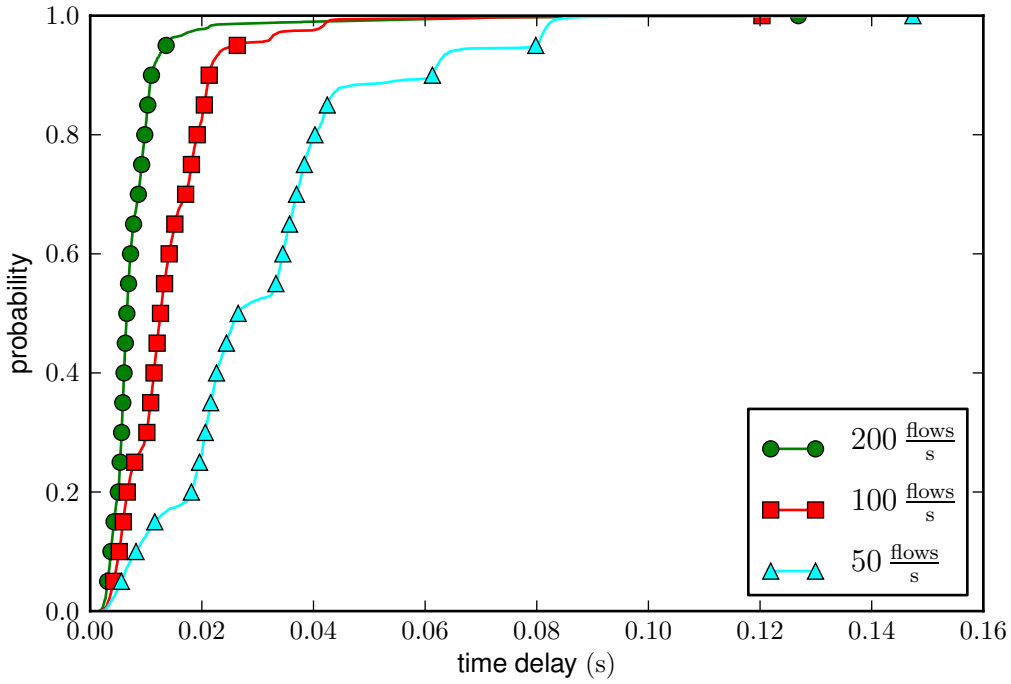

Figure 5.6: ECDFs for the time delays between packet-in and packet-out for the first event of each bi-flow traversing the testbed. For the high flow rate 90% of the delays are below 10 ms. Some modes are visible for the lower flow rates.

5.4.2 Distribution of the time delays

Figure 5.6 shows ECDFs for the three lower rates of the first 5000 flows each for the delay of the first packet-in event. The highest rate with $200\ \frac{\text{flows}}{\text{s}}$ shows about 90% of the delays are below 10 ms. The rate with $100\ \frac{\text{flows}}{\text{s}}$ shows modes at 10 ms, 20 ms and 35 ms. The rate with $50\ \frac{\text{flows}}{\text{s}}$ shows modes at 20 ms, 35 ms, 65 ms and 85 ms.
Figure 5.7 shows the ECDFs for the three lower rates of the first 5000 flows each for the summed up delays of all the packet-in events in a bi-flow. At a flow rate of $200\ \frac{\text{flows}}{\text{s}}$ about 90% of the delays are below 25 ms. There are no clear modes visible for all the rates.

5.4.3 Maximum handleable flow rate

In order to find the maximum flow rate, the Arbitrator is able to handle, we measured time delays with different flow rates around the expected maximum. Figure 5.8 shows again the time delay of handling the first packet-in event in a bi-flow for different rates between $400\ \frac{\text{flows}}{\text{s}}$ up to $500\ \frac{\text{flows}}{\text{s}}$. One can clearly see a split between the rates $460\ \frac{\text{flows}}{\text{s}}$ and $470\ \frac{\text{flows}}{\text{s}}$. At a rate of $460\ \frac{\text{flows}}{\text{s}}$ the Arbitrator can not work off the whole backlog created by the peak times created by reordering the internal data structure, but in the intervals between the peaks, one can see, the delay is decreasing, thus we assume, with a “warmed up” data structure the Arbitrator would be just able to handle $460\ \frac{\text{flows}}{\text{s}}$. In contrast to that at a rate of $470\ \frac{\text{flows}}{\text{s}}$ the delay is even increasing in the time intervals between the peaks, thus the Arbitrator is not able to handle that flow rate.
Figure 5.9 shows only the first 5000 flows and only the rates between $400\ \frac{\text{flows}}{\text{s}}$ and $450\ \frac{\text{flows}}{\text{s}}$. With the highest of that rates the Arbitrator is building a small backlog over time, as can be seen in the plot the curve is not getting near zero after a while in comparison to the other curves. But the Arbitrator would, as seen before, also be able to handle this rate with a “warmed up” dictionary. $440\ \frac{\text{flows}}{\text{s}}$ is always getting near zero, thus it is about the highest rate the Arbitrator can handle without a “warmed up” data structure.
Figure 5.10 shows the ECDFs of the tests with high rates. As expected the higher rates typically show a higher delay.

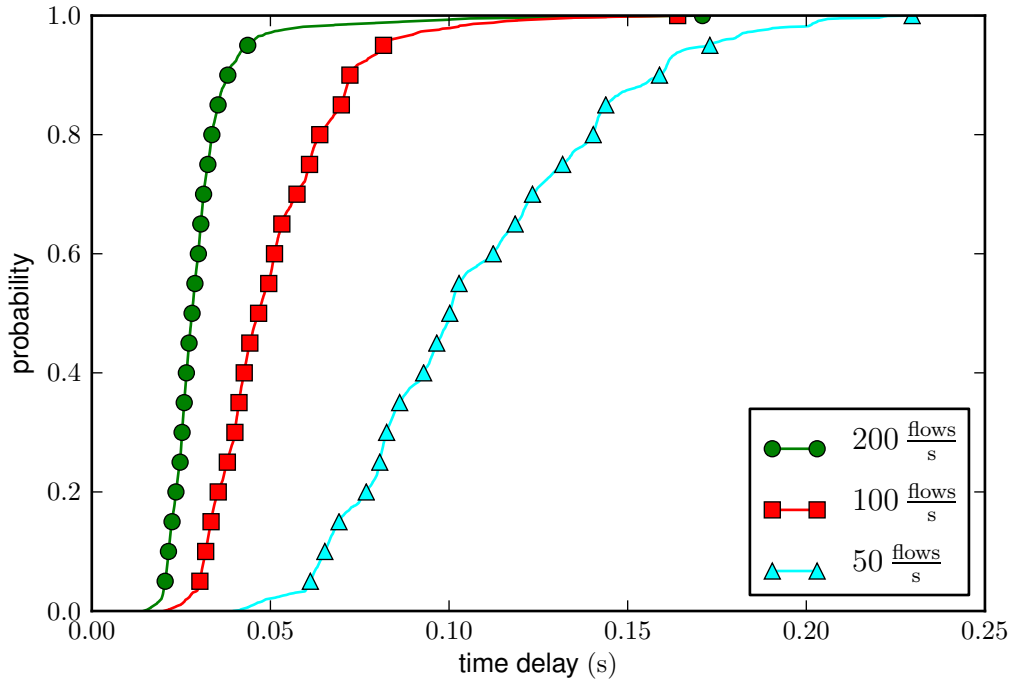

Figure 5.7: ECDFs for the summed up time delays between packet-in and packet-out for all events of each bi-flow traversing the testbed. For the high flow rate 90% of the delays are below 25 ms.

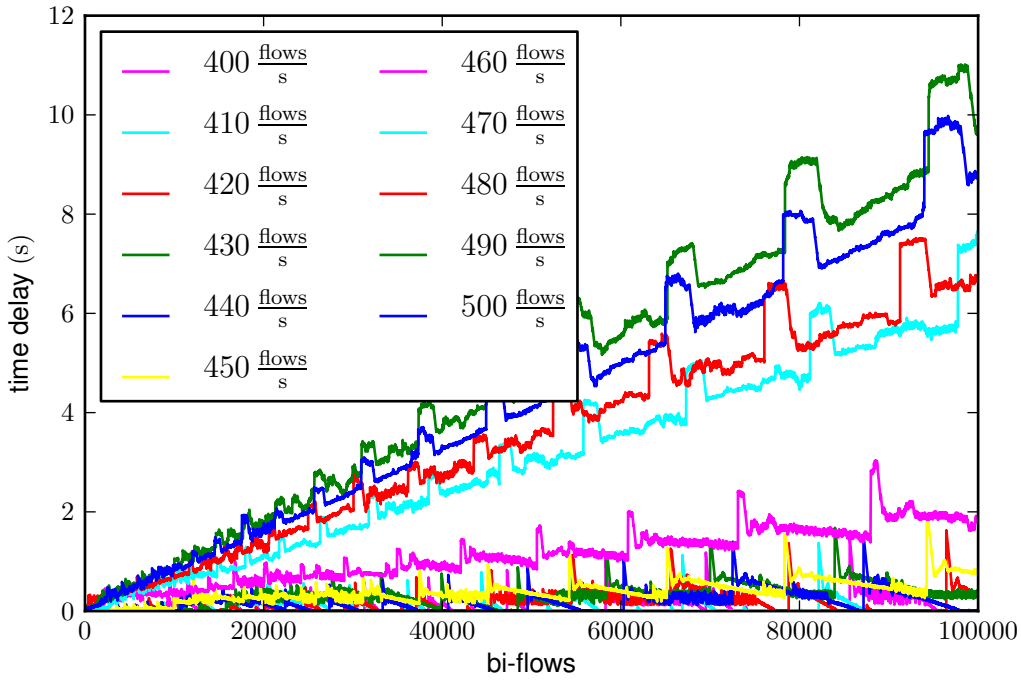

Figure 5.8: Time delays between packet-in and packet-out for the first event of each bi-flow for several high flow rates. A clear separation between $460\ \frac{\text{flows}}{\text{s}}$ and $470\ \frac{\text{flows}}{\text{s}}$ is visible. We assume this is about the maximum flow rate the Arbitrator is able to handle.

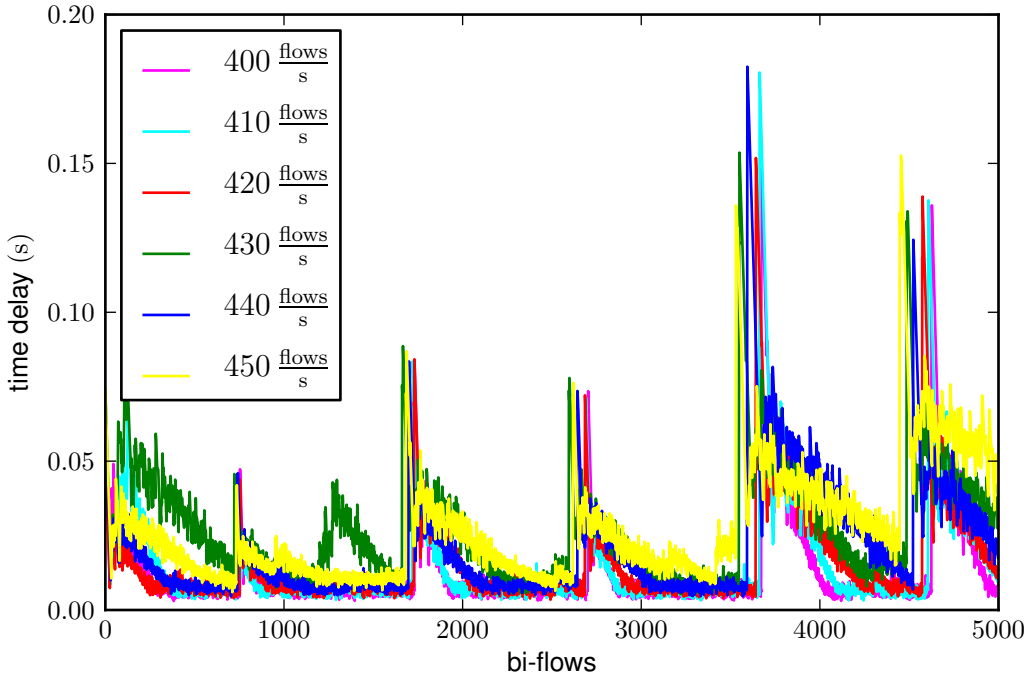

Figure 5.9: Time delays between packet-in and packet-out for the first event of each bi-flow for high rates. The yellow curve shows, at that flow rate the backlog created by reorganizing the internal data structures cannot be completely worked off.

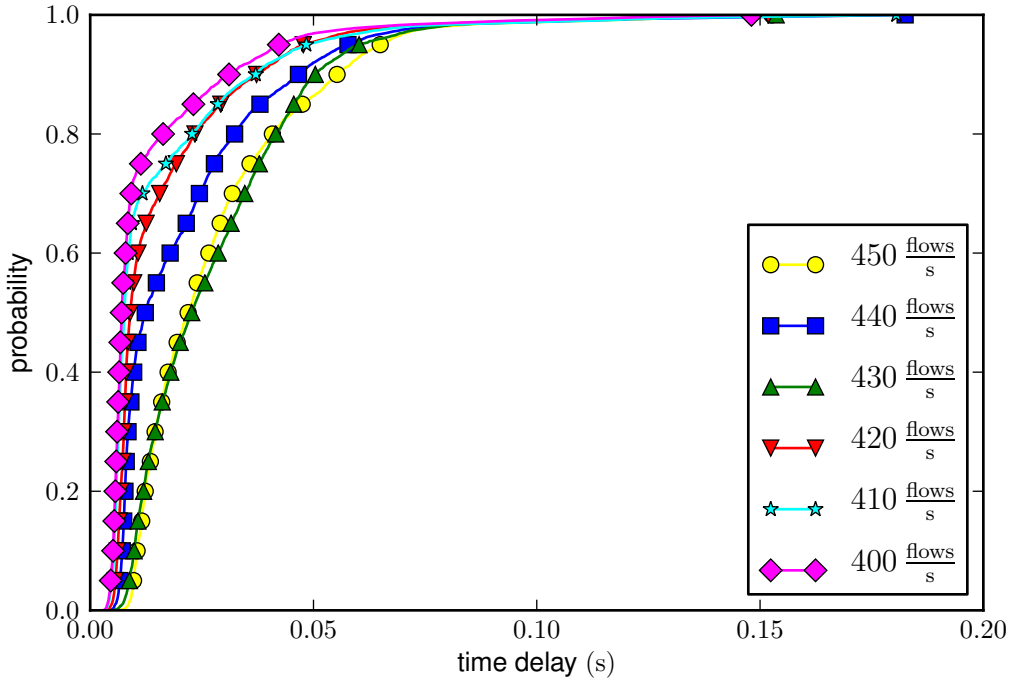

Figure 5.10: Distribution of delays between packet-in and packet-out for the first event of each bi-flow for high rates. As expected, higher flow rates typically show a higher delay.

5.4.4 Comparison with flow arrivals at the TIK network

We compared the flow arrival rate at the TIK network with the capability of our setup. As can be seen on Figure 5.11 the TIK network[2] shows on a regular Thursday during the semester a baseline of about $10\ \frac{\text{flows}}{\text{s}}$ and about twice that much during working hours. When looking at a bin size of 1 second we have peaks up to $210\ \frac{\text{flows}}{\text{s}}$. With a bin size of 0.1 seconds we see peaks up to $1200\ \frac{\text{flows}}{\text{s}}$. The $1200\ \frac{\text{flows}}{\text{s}}$ during 0.1 seconds correspond to 120 flows. Assuming our system can handle $460\ \frac{\text{flows}}{\text{s}}$, the backlog of 120 flows will be worked off in about 260 ms.
Thus we assume our system can handle the traffic of the nature of the TIK network with introducing small delays.

5.5 Correctness Evaluation

In order to test the Arbitrator and the PAL proxy a testbed was set up in the very beginning of the work. The goal of the testbed is to evaluate the correctness of the Arbitrator in combination with the PAL proxy. It was also an important aid during development and debugging to find programming errors. The testbed was built using the Mininet framework.
In the beginning testing was done manually, by starting each component by itself and using Mininet's interactive console to send traffic through the virtual testbed, but this is a tedious task. To overcome this, a simple test script has been written. The test script takes care of setting up the Mininet testbed and launches all the needed components which are the Arbitrator, the PAL proxy and some experimenters' controller. It then uses tcpdump on some links to capture traffic for later analysis.
For testing, we configured some static policies in the Arbitrator. We configured policies to allow some traffic being forwarded to the testbed and others to shortcut the traffic and not let it be forwarded to the testbed.
The test script then sends packets through the virtual network and uses captured traffic by tcpdump to evaluate the path the traffic took. If it matches the statically configured policy then the test passes, otherwise it fails.
The manually performed as well as the automatically performed tests were successful and the implemented program behaves as specified.

[2] Two high traffic hosts have been omitted from the measurement.

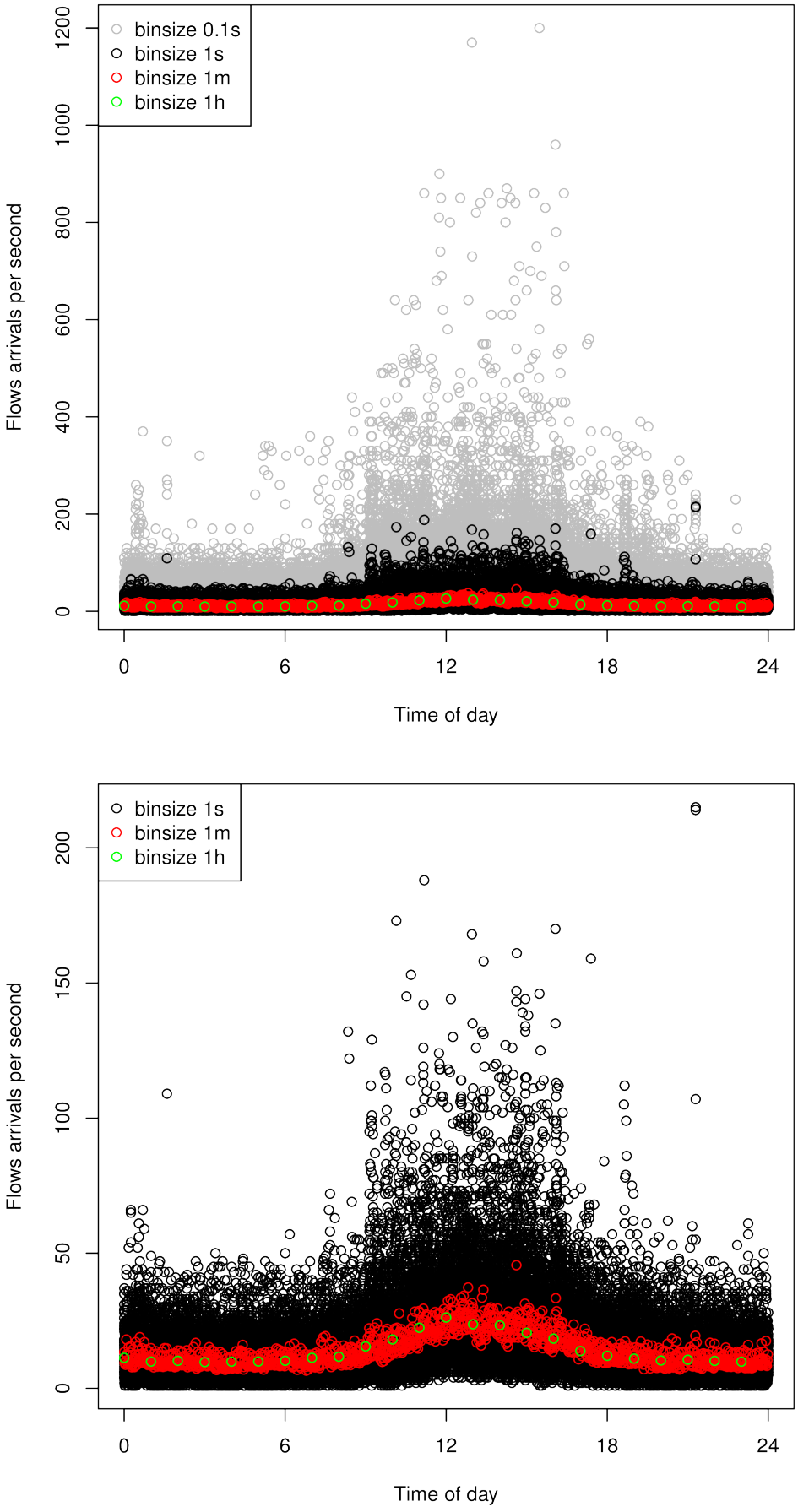

Figure 5.11: Flow rate at the TIK network for different time scales. (Plots provided by Bernhard Ager)

Chapter 6

Future Work

In the following sections we describe possible improvements to the Arbitrator and the PAL in general.

6.1 Possible improvements to the Arbitrator

A future improvement of the Arbitrator would be a user interface, where users can configure flows, state guarantees for their flows and select experiments, they want to donate traffic to.

This user interface then would have to be backed by a database for permanent storage. It has to be evaluated if the Arbitrator looking up guarantees in the database is fast enough, or whether a copy of the database's content is needed in the Arbitrator itself.

Furthermore to motivate the user to donate traffic, a marketplace should be created. Experimenters should be able to trade something with users for the donated traffic. A trade value can come from the experimental service itself, like a transparent BitTorrent cache or an anonymizing proxy, or credits to access restricted web services or simple money could be a trade value.

6.2 Possible improvements to the PAL proxy

The PAL proxy needs some refactoring or a redesign. In order to benefit from future POX improvements and to keep the maintenance overhead low the PAL proxy should be written as a POX module.

As we have seen in Section 4.2.2, keeping the flow tables in sync is not a trivial task. The part of the PAL proxy which is responsible for maintaining a copy of the switches' flow tables could be refactored to a separate library. Separation would allow independent and more extensive testing and would possibly enable other project to benefit from it.

We have also seen in Section 4.2.2, the PAL proxy heavily depends on the FlowVisor and we had to implement a work around for a problem resulting from them being separate applications. This leads us to other possible improvements. The PAL proxy should use more communication channels with FlowVisor to learn how rules get handled. Other options would be to build the PAL proxy into FlowVisor, or at least implement a subset of FlowVisor in the PAL proxy.

Assuming experiments are perfectly isolated by FlowVisor, each experiment could be assigned it's own PAL proxy. This would allow us to scale the PAL proxy to multiple CPUs and even multiple hosts, at almost no development effort.

Performance evaluation of the PAL proxy was out of scope of this thesis. In order to evaluate the performance of the of the PAL proxy one would have to design a testing environment with different experimenters.

The impact of the introduced barrier requests as described in Section 4.2.3 should be evaluated. Possible improvements would be to send the barrier requests in batches.

6.3 Deployment in ETHZ's OFELIA island

Deploying PAL in ETHZ's OFELIA island is another pending task. This task can be performed as soon as outstanding issues with the PAL proxy have been resolved. More insights are expected from this task.

Chapter 7

Summary

In this thesis we have designed and implemented a traffic Arbitrator for a public SDN testbed. It has been integrated in the already existing PAL. The integration of the Arbitrator in the PAL consisted of designing and implementing a communication interface to the already existing PAL proxy and extending the PAL proxy to fulfill our needs.
Also some bugs and design errors in the PAL proxy were discovered and fixed or had to be circumvented during this thesis. Furthermore we learned some more insights into the OpenFlow protocol and the use of FlowVisor in the PAL.
We learned that OpenFlow is a sub-optimal choice for a privacy layer. It was already known, that handling idle timeouts is an issue, but the presented solution was not complete. An issue we learned was that the lack of acknowledgement messages to flow mod messages complicates our task. We circumvented that with barrier requests.
In order to work around a design error in the PAL proxy we had to reimplement parts of FlowVisor's behaviour into the PAL proxy. We have seen the PAL proxy within FlowVisor is now more feasible than before.
Furthermore we have evaluated the performance and the correctness of the Arbitrator. We compared the performance with the nature of the traffic in the TIK network. We could show the Arbitrator is typically fast enough to handle a network of this size and deployment would be possible.

Bibliography

[1] SPDY: An experimental protocol for a faster web,
`http://www.chromium.org/spdy/spdy-whitepaper`,
last visit: 2014-02-11

[2] OpenFlow Switch Specification,
Version 1.0.0 (Wire Protocol 0x01), December 31, 2009

[3] On Bringing Private Traffic into Public SDN Testbeds,
V. Kotronis, D. Schatzmann, B. Ager, ETH Zurich, 2013

[4] Supporting Header Field Re-writing for Policy-bound Flows in a Software Defined Network,
Hildur Ólafsdóttir, Semester thesis at ETH Zurich, 2013