Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Computer science / Informasjonsteknologi	Vår semesteret, 2014 Konfidensiell
Forfatter: Christian Stigen Larsen	(signatur forfatter)

Fagansvarlig:

Veileder(e): Hein Meling

Tittel på bacheloroppgaven:
Distributed Switch-level Message Ordering for Replicated Services

Engelsk tittel:
Distributed Switch-level Message Ordering for Replicated Services

Studiepoeng:

Emneord: Networking, Paxos, OpenFlow, software-defined networking, Open vSwitch, Python, C, Ethernet, fault-tolerance, replication, mirroring	Sidetall: ………75………… + vedlegg/annet: …0……… Stavanger, 15.05.2014 dato/år

Distributed Switch-level Message Ordering for Replicated Services

Christian Stigen Larsen

May 15, 2014

Abstract

Sofware-defined networking (SDN) decouples the control plane from the forwarding plane in switches, making it possible to create controllers in software. This has made it easier to build arbitrarily advanced networks, such as networks that self-optimize for low power-consumption [18], cloud networks that route live traffic to moving virtual machines [8] as well as making it possible to research and test new networking protocols using simulators on a laptop.

A trend in the computing industry is that more services are moved to the Cloud, reaching a larger number of users. Thus, it becomes more important that these services remain available despite failure of individual machines, calling for distributed service replication. A huge body of research in the distributed computing community have been devoted to coming up with protocols that guarantee strong consistency among the replicas of such a service. The most cited paper on network resilience is the Paxos algorithm [21] for message ordering.

By extending the OpenFlow protocol, we have implemented steady-phase Paxos on a software switch, showing how it is possible to compose network flows that leverage the message ordering guarantees of Paxos as a constituent element. As a demonstration of its use, we have built a system where distributed UDP-based services were replicated in-order using Paxos transparently.

We also implemented these Paxos primitives on the switch itself by modifying Open vSwitch, but were unable to fully test this solution. However, we have demonstrated that by moving primitive operations down to the switch, it became vastly easier to build advanced flows—compared to programming them in the controller.

Due to the elegant way in which flows can be built with new OpenFlow primitives, we believe that our work shows a possible future direction for the OpenFlow specification that encompass programming the switch.

Acknowledgments

I would like to thank my thesis advisor Hein Meling for his close collaboration, guidance, patience and insightful comments throughout this work. Hein approached me with the fascinating idea of putting Paxos in the switch, and I thank him not only for letting me explore this problem, but also for getting me hooked on software-defined networking.

Multi-Paxos would not have made it into the implementation without the clear tutorial given by doctoral student Tormod Erevik Lea.

During my modifications of Open vSwitch, I received helpful advice from its friendly community on the project mailing list.

Finally, none of this would be possible without my amazing fiancée Siri.

While I spent my days at work and University, she had the overwhelming pleasure of taking care of our two lovely children, Mari and Bjørn.

Thank you for helping me achieve a long-sought dream: I dedicate this thesis to you.

List of Tables

List of Figures

List of Algorithms

Listings

Contents

Acronyms

ARP	address resolution protocol	39
BPF	Berkeley packet filter	68
GPG	the GNU Privacy Guard	59
ICMP	Internet control message protocol	41
IP	Internet protocol	41
L2	OSI link-layer	36
MTU	maximum transmission unit	23
NAT	network address translation	41
ONF	Open Networking Foundation	17
PGP	Pretty Good Privacy	59
PRNG	pseudo-random number generator	47
QoS	quality-of-service	51
RTT	round-trip time	45
SDN	software-defined networking	15
TCP	transmission control protocol	29

ToS type-of-service 30

UDP uniform datagram protocol 29

VM virtual machine 17

WAN wide-area network 41

Chapter 1

Introduction

With the emergence of software-defined networking [4], the switch becomes a much more central component in the network infrastructure.

That is, the switch can be programmed to perform a mude wider range of functions that previously could not be done, or had to be done on end-hosts or in the router. This includes support for packet filtering (firewall), intrusion detection and much more.

Emerging from research at Berkeley and Stanford around 2008, the basic idea of SDN was to decouple the control plane from the *forwarding plane* (or *data plane*). The forwarding plane is where packets are transferred from one point to another, perhaps changing some header fields along the way. By design, the operations in the forwarding plane are simple relative to the capabilities of a controller to enable high performance. The decisions about *how* the forwarding plane should function is made in the control plane—by a controller—and is much more advanced in comparison. For a forwarding plane to operate efficiently, the controller must have access to information such as the network topology, an overview of traffic and so on. It can then program the forwarding plane accordingly.

OpenFlow [9] is one way to use software-defined networking. It specifies a communication protocol between a controller and a switch. Using OpenFlow, one can write controllers in practically any programming language, making it easy to build networks in a much more dynamic fashion than before.

Although invented quite recently, software-defined networking has already seen heavy use both in academia and industry. Google, for instance, are using OpenFlow to ease deployment and increase utilization in their backbone networks [7] and Stanford has deployed several OpenFlow-controlled networks on their university network.

Another trend in the computing industry is that more and more services are being moved to the cloud, reaching a larger number of users. Thus, it becomes even more important that these services remain available despite failure of individual machines running those services. This calls for replicating these services on several machines. A huge body of research in the distributed computing community has been devoted to coming up with protocols to guarantee strong consistency among the replicas of such a service. The most cited of these protocols is the Paxos protocol [21, 20].

Paxos is a family of distributed, fault-tolerant consensus algorithms. It allows network nodes to reach *agreement* even in the face of intermittent network failures. For example, one can design a database system using Paxos to make sure that transactions are executed in the same order on all nodes.

Most general-purpose implementations of Paxos are built as middleware software that must be integrated into server code. This has several drawbacks. For instance, it operates at the upper level of the networking stack—at the application level—and away from the central switching points in a network. Another drawback is that one has to tailor Paxos specifically for each and every service.

In this thesis we want to investigate how we can build Paxos at a networking level, closer to its central parts and in particular, if this can be done transparently, making it possible to offer Paxos to services not originally designed for distributed operation. Furthermore, to demonstrate its applicability, we want to build a system of replicated services, where each *end-host* (running general purpose software services such as log-servers, etc.) is replicated by receiving the same packets from clients. Using Paxos, we ensure that the services receive the packets in-order, even in the face of failures.

There may be several benefits in doing this, and we will return to them later in this thesis, but list the three most important here:

- **Transparency:** We guarantee consistent in-order delivery of network packets at all replicas. This has the benefit that network applications can be replicated without the need for complicated Paxos logic. While this is usually handled by a middleware framework, we can avoid imposing a specific API on the application developer. For example, frameworks typically impose a language specific API, preventing users of other languages from leveraging the replication service. We ignore the need for state transfer in this thesis.

- **Performance:** By implementing Paxos in the switch, we expect to see a performance improvement. This is because, assuming the latency between the switches is low, we can avoid the extra messages and latency of traversing the links and IP stack of the end-hosts running the replicas of the services.

- **Traffic reduction:** Using a setup with “Paxified” switches, every application that wishes to have their messages totally ordered can use that functionality offered by the switches. Thus, we can leverage one setup to provide several replicated services with ordered message delivery. Therefore, instead of having each replicated service runing its own Paxos protocol, they can all receive ordered messages from the switches. While this still requires Paxos messages exchange between switches, it can reduce the traffic on the links between the switches and replicas.

There are a few drawbacks and restrictions that are worth mentioning.

The performance improvement we expect to see is fairly small compared to wide-area network latencies. Thus, it may not make sense to place Paxos logic in the switches that communicate over wide-area links. That said, there are several works in which one runs two separate Paxos protocols, one within each data center, and another between data centers. In such a configuration, it would definitely make sense to use in-switch Paxos deployments within the data center, even though the other Paxos does not.

While most of the time a single switch operates as an autonomous unit, to implement Paxos in the switch, we need to impose some restrictions on the topology between the machines.

A drawback with implementing Paxos in the siwtch is that a physical switch typically has a very restricted instruction set available, and thus implementing Paxos at this low-level can be very challenging and prone to unforeseen issues.

Despite these restrictions, we believe that the benefits can be of value to a wide range of deployments where it is desirable to provision enough replication to ensure high availability of services.

To reduce the scope of this thesis, we will constrain our scope to a few primitives of classic crash Paxos in *phase two*, where we have steady-state flow with no failures.

Most importantly, ths is a study of *feasibility*. We want to explore whether implementing a distributed protocol at lower networking levels is doable, viable and whether it can be offered transparently to the services.

Chapter 2

Background

In chapter 1 we stated our thesis goal of offering Paxos at the networking-level and building a distributed replication system on top of it. Here, we will present the backround material for algorithms and technology we will use in the rest of this thesis.

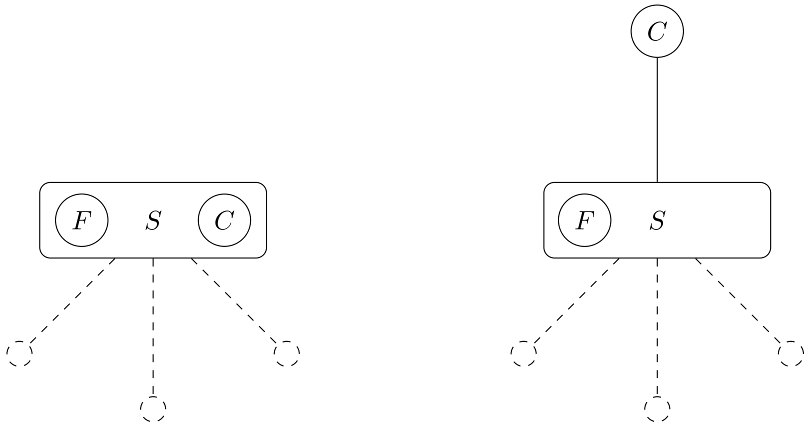
2.1 OpenFlow

As mentioned in chapter 1, the core idea of SDN is to dissociate the controller from the switch, enabling software developers to "program the network". This is illustrated in figures 2.1a and 2.1b on the following page.

OpenFlow [25, 9] is one of several ways to enable software-defined networking (SDN) It is a specification of a messaging protocol for the interaction between a switch and controller, detailing what kind of actions that can be performed by each of these components. Examples of SDN solutions predating OpenFlow are SANE [3], Ethane [2], 4D [16].

In OpenFlow, the controller is a piece of software can inspect packets and send commands down to the switch. It can instruct the switch perform actions like forwarding a packet to an output port going to another switch, changing header fields, drop it and so on. Additionally, the switch can be set up to perform such actions by itself, using a *flow table.* This is what we have previously referred to as the *forwarding plane*, and as highly performant in comparison to handling each individual packet in the controller.

OpenFlow controllers can be written in any programming language that offers an OpenFlow framework. While the OpenFlow switches are usually also implemented in software, several vendors now ship hardware switches that support the OpenFlow protocol.



(a) A typical switch S combines the control and forwarding planes (C and F) on the same device. The control plane C is usually locked down by the vendor and inaccessible to users.

(b) SDN decouples the control plane C from the forwarding plane F by moving it out of the switch S to an external network node. The *OpenFlow protocol* enables communication between S and C, making it possible to implement C in software on an ordinary computer. In OpenFlow, the connection between S and C is encrypted.

2.1.1 The Flow Table

The aforementioned flow table consists of several flow entries—colloquially called *flows*. These contain rules for matching packets and actions to perform when there is a match. In addition there are several counters used for collecting various statistical metrics (e.g., how many times a flow has been executed) and timeouts that dictate how long an entry will exist in the table.

When a switch receives a packet, it will try to match it with entries in the flow table (table 2.1). Each flow contains a *matching pattern* and a set of actions to perform in case there is a match. The actions can be to rewrite a header field, forward the packet to a port, drop it and so on. If a packet does not match any flows, the switch will forward a buffer ID and packet headers to the controller C_1 on a secure channel.

Header fields	**Counters**	**Actions**
...	...	...

Table 2.1: The OpenFlow 1.0 flow table.

The controller can then decide what to do with the packet. Using the OpenFlow protocol, it can issue immediate actions to the switch or install flows, so the switch can operate on its own.

The flow tables are initially empty, meaning that all packets are by default sent to the controller. During this phase, the controller will explicitly handle every packet. At the same time, it will incrementally build up an internal map of the network. As the map forms, it can start installing flows in the switch.

For instance, if a controller learns the port numbers for a pair of addresses, it can instruct the switch to automatically forward packets to their appropriate output ports when they communicate. If a packet's destination port is unknown, the controller can *rebroadcast* it out on all ports except where it came from, knowing that only designated receivers will accept the packet. What we have described here is a *learning switch*, and is explained in detail in chapter 4.2 on page 36.

Along with each flow entry is an associated set of timeouts. Flows are removed from the flow table when they time out. This serves several purposes. First of all, it makes sure that flow tables do not fill up quickly. Secondly, because flows—and packets—are transient by nature, controllers will be given the chance to update rules based on changes in the network. Finally, this mode of operation adheres to the principle of autonomous operation commonly seen in networking devices.

Well-designed controllers should not need elaborate configuration to work. So while they initially do all the heavy-lifting by themselves, they will offload work to the switches, who can then dispatch packets very quickly.

2.1.2 Applications of OpenFlow

We would like to briefly mention a few real-world uses of OpenFlow.

Researchers at Stanford built an OpenFlow network that was able to migrate a virtual video game server virtual machine (VM) from California to Japan—while it was running, without interruption, using the *same* IP-address [8] [19].

Google are using OpenFlow in their backbone network to increase utilization [7] and is an official member of the governing institution of OpenFlow, the Open Networking Foundation (ONF).

The Open Networking Foundation released the first specification of OpenFlow in 2009 and continually publish point versions, errata and new versions [9, 11, 14, 10, 12, 13, 15]. The most recent one, at the time of writing, is version 1.4.

2.2 Mininet, POX and Open vSwitch

Mininet [28] is an open source network simulator that supports OpenFlow. Using the Python programming language, one can deploy virtual networks on a laptop, configuring link speeds, percentage of packet loss and so on. Controllers are written using the POX [31] OpenFlow framework.

At the bottom of all this is the Open vSwitch program [30]. It is a powerful software switch used by many cloud providers. It supports most of the OpenFlow functionality, and runs both as a Linux kernel module and in userspace. Open vSwitch is written by many of contributors of the OpenFlow specifications, and the three projects share close ties.

Together, they form a powerful combination of software that makes it easy to experiment with networking technology. We have used them for our thesis implementation.

2.3 Paxos

Paxos [21, 20] is a family of fault-tolerant, distributed consensus algorithms, allowing nodes to reach agreement in the face of intermittent failures. Originally published by Leslie Lamport in 1989, Paxos has spawned numerous extensions, including cheap Paxos, fast Paxos and Byzantine , fault-tolerant variants.

We will not give a full account of the Paxos algorithm here, but will mention the main parts that are relevant for this thesis. For reference, we will hereafter keep to the description given in [21].

Paxos consist of two main phases: *Phase one* and *phase 2*. The first phase consists of choosing a leader among all the Paxos nodes. This phase continues until a leader has been chosen by the majority. As we will only focus on phase two, we will not discuss this part further.

Phase two, also called *steady-phase*, is where *message ordering* takes place. We say that we want to reach *consensus* on some *value*. A *value* can be anything we want a majority of the nodes to agree on. For our purposes, it means we want to determine which packet to next process (i.e., we want to determine the *order* in which to process packets). Paxos nodes will send Paxos messages to each other, and they contain various parameters. We will return to these in section 3.2 on page 25.

The *Paxos leader* from the first phase will receive a message somehow—in our case, it will be a client packet that we wish to distribute and process in-order—send out an *accept* message containing a *value* to be agreed upon—the value can for example be a packet or a reference to a packet, for example—and sending out an *accept* message to its *Paxos slaves*. Only the leader sends out accept messages.

When a slave received an accept, it will first see if this is a message that came from its current leader. If so, and if it has not seen this message before, it will send a *learn* message to *all* Paxos nodes, including itself.

Upon receiving a learn-message, a node will first check if it belongs in the current *round*—meaning that the message belongs in the current round with the given leader. In other words, if the round number contained in the learn message is less than the node's current round, this was a message from a previous point in time in which there was *possibly* another leader. If the learn has not been seen before, it will record how many unique learns it has seen. Whenever a node has received learns from a majority of other nodes, it will start to process a queue of messages in the order specified by the message parameters.

We will refrain ourselves from discussing *why* the algorithm works, even in the face of failures. For such details, we refer to [21, 26].

This concludes our very brief account of Paxos, and we will return to it in section 3.2.

Chapter 3

Design

Based on our goals stated in chapter 1, we will present a design for a network-level Paxos system using the technologies from chapter 2.

3.1 Topology

A premise for implementing Paxos in the switch is that we also deploy multiple switches to provide the necessary resilience to failures. Thus we need to provision a switch topology.

In this section, we will present how we plan to achieve this. Starting with a single switch, we show how it can use a controller to shuttle packets between clients and services running on end-hosts (figure 3.1). By linking three such switches together, one will typically want to install Paxos middleware on each host to provide service replication and fault-tolerance (figure 3.3). We then propose that by moving Paxos from the hosts to the switches, one can offer the guarantees of Paxos, *transparently*, for a wide class of services (figure 3.4).

Let us start with the simplest case: One switch.

In figure 3.1 on the facing page, a switch S_1 is connected to several nodes through ports. On one of these, the switch will receive packets from a set of clients c_1, c_2, ..., c_n. It has an OpenFlow controller C_1 and three end-hosts h_1, h_2 and h_3 running software to service client requests.

Note that the actual location of the clients is irrelevant to our current discussion. Our focus is on the possible merits of placing Paxos on the switch, not on how clients are able to reach the services. We will therefore *assume* that we are able to communicate with a set set of clients on a designated port.

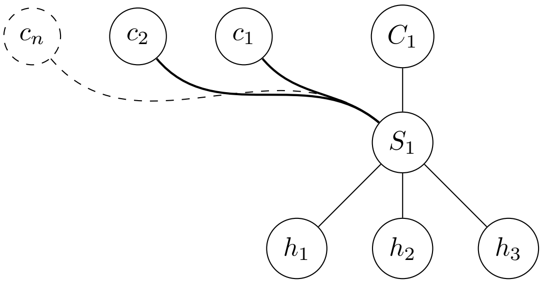


Figure 3.1: A single switch S_1 with its controller C_1, connected to end-hosts h_1, h_2, h_3 and clients c_1, c_2, $\ldots$, c_n through ports.

The topology in figure 3.1 allows us to program the controller to replicate client messages to all its hosts. This is also called *mirroring*. We can do this by duplicating each incoming client packet, forwarding packets with headers rewritten to match each host's Ethernet and IP-address. Duplicate replies to the clients should be discarded, but we will ignore that for now. The services will then process packets and—assuming the service protocol is deterministic with regards to its input—end up in equal states. While this should work well for UDP, which is stateless, it would be *significantly* more involved for TCP. We discuss TCP replication in chapter 7.2 on page 54.

However, a single switch is inherently prone to failure. Should it fail, it would take down all the services along with it. The obvious step is to add more switches. But adding a second switch will not be sufficient. Should the two sets of hosts end up in different states—in the case packet loss, for example—they would not be able to decide whose state is correct. We will therefore look at using three switches.

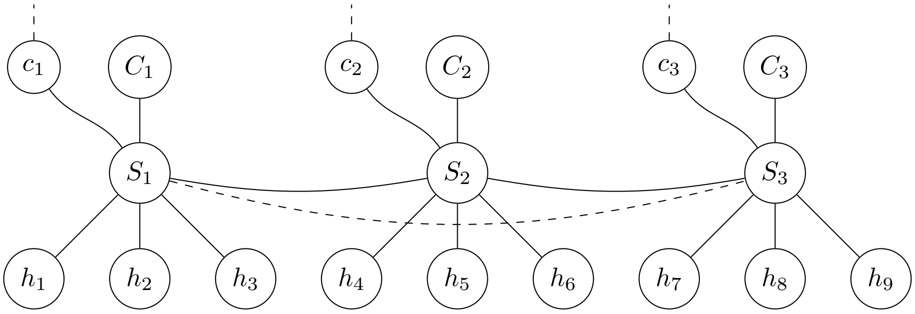


Figure 3.2: Three switches S_1, S_2, S_3 with controllers C_1, C_2, C_3 acting as Paxos nodes. The dashed line between S_1 and S_3 is a potential fail-over link. Each client is assumed to be able to connect to any switch.

Figure 3.2 presents three interconnected switches. Potential fail-over links are indicated as dashed lines. This is in case a switch fails. Again, we disregard the details concerning the fail-over system for clients.

We still want to perform service replication using this topology, but now our latencies are *asymmetrical*: Given equal link-latencies, a packet sent from c_1 to h_9 traverses more links than a packet from c_1 to h_1. The packets may therefore arrive at different times at h_1 and h_9. This causes an *ordering problem.* If packets from several clients reach hosts at different times, they will also arrive in different order. This may break the service protocol or cause service states to differ. To mitigate this problem, we need a mechanism to make sure that packets are delivered *in the same order.* The switches should agree on some order—i.e., reach a *consensus.* In case of disagreements, we will let a majority decide—a *quorum*—and to form a quorum, we need at least three switches.

We have chosen the Paxos algorithm to resolve the ordering problem.

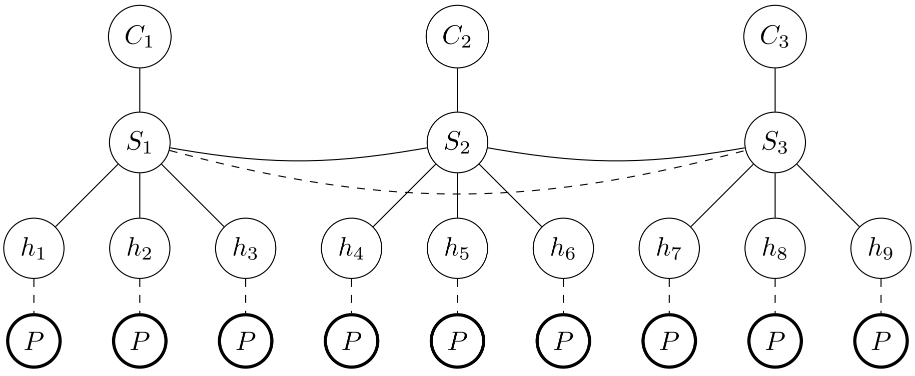


Figure 3.3: Support for Paxos on the servers $h_1, \ldots, h_3$ requires a copy of the Paxos code P on each server.

Figure 3.3 shows an straight-forward way of deploying Paxos as middleware software frameworks on the end-hosts.

First let us reiterate our aim: To provide replication of the services running on the hosts $h_1 - -h_9$. We do this by duplicating each incoming client packet and use Paxos to reach an agreement on the order in which they should be delivered to each end-host.

This configuration is obviously severely simplified, far from actual, application-level Paxos configurations. Specifically, one would most likely spread each kind of replicated service across each switch, so that, e.g., h_1, h_4 and h_7 replicated service A, using Paxos nodes P_A^1 on h_1, P_A^2 on h_4, or one could have an additional Paxos system on a higher level, and so on. Still, figure 3.3 is useful to highlight some problems with some conventional Paxos deployments.

We have removed the clients from the current and following figures for clarity. We still assume that the switches receive and transmit packets to an unknown number of clients.

Now, assuming that Paxos has been deployed as middleware software as shown in figure 3.3, we can point out a few drawbacks for this particular configuration:

- **Application-level messaging:** Paxos messages between each node (P) begin and end at the application-level, and must traverse down the whole networking stack, over the links and back up again. There is some inherent overhead in doing so, and for certain traffic patterns–when there are a lot of messages to transmit, e.g.—this *could* become a problem for total system responsiveness.

- **Non-central network placement:** The Paxos nodes are positioned away from the central switching components of the network (i.e., the switches $S_1 \ldots S_3$), requiring additional link-hops to reach their destinations.
- **Software requires built-in Paxos:** Finally, only software that that comes with Paxos built-in can possibly support it. This can potentially become a problem for, e.g., businesses that unexpectedly need to scale a service—especially when considering services that run in the cloud.

Now consider the situation where the *switch* (along with its controller) provides Paxos capabilities (figure 3.4).

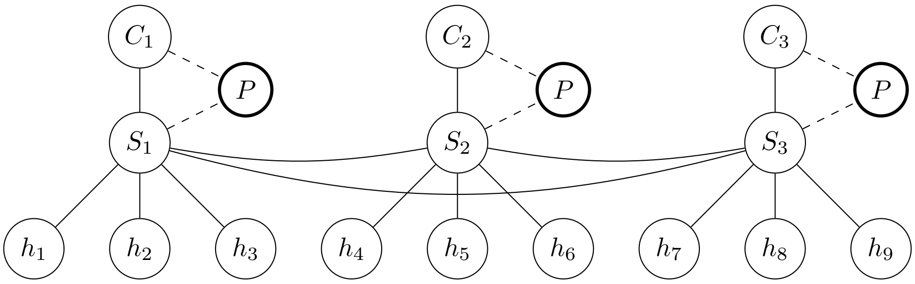


Figure 3.4: Paxos (P) in the switches S_1, S_2, S_3 mitigates the need for special code on the servers.

In figure 3.4, the switches themselves (and their controllers) enable support for Paxos.[1] This should let the end-host services be oblivious to the fact that Paxos is used to order the arrival of packets to them, providing Paxos *transparently.*

Besides, Paxos is now run at a much lower networking level and at the point where switching is done—there will be less hops for each packet.

Of course, this solution is neither without limitations:

- **Must perform ordering on each packet:**

  Performing Paxos ordering at the level of each packet means that a lot of Paxos messages must be exchanged. For networks with a small maximum transmission unit (MTU), a large packet may be split up into several smaller ones. Thus, the advantage of switch-level Paxos may soon become a bottleneck in itself, especially when client messages are large.

- **Cannot exploit application-level protocol:**

  Also, at this level we lose opportunity of exploiting knowledge about each client-server protocol. As a hypothetical example, a server may know that

[1] We have moved the servers to several switches to indicate a distributed nature between the switches. Paxos on a single switch would not be very useful, as that would be a single point of failure and—after all—be the sole decision point for message ordering.

some client operations are side-effects free (a *read*-operation on a key-value server, for instance), and can choose not to initiate a Paxos exchange for such operations.

- **Only supports deterministic services:**

  And some service protocols simply do not fit into this model of replication at all. The obvious example is services that are not deterministic on their input—meaning that if two identical services receive exactly the same input, in exactly the same order, their internal states may become different.

But many services should still fit in to a model of switch-level Paxos replication. For instance, replicating a UDP-based logging server could prove to be beneficial.

We will not delve deeper into this matter. Especially, we completely disregard details concerning replies back to the clients: We silently drop duplicated replies. A deeper study would certainly take a closer look at such situations, and we discuss a result regarding TCP in section 7.2.

Neither have we looked more at the situation when a switch goes down. We have indicated fail-over links in the topologies, but have not implemented such functionality. For the part that concerns Paxos, such a situation should start a new leader election. If a switch comes back up again, it should synchronize the state of its services with the others. All of these are separate studies on their own, but we would like to note that OpenFlow does have some support for monitoring link-status, and it would be very interesting to make use of that in the face of failures.

Finally, we have not attempted to make an optimal design. We concentrate solely on the part of finding out if implementing Paxos on the switch-level is *possible* and if it could be *useful*.

3.2 The Paxos Consensus Algorithm

3.2.1 Full Paxos

The Paxos papers by Lamport [20, 21] do not describe a specific implementation algorithm. Details such as liveness checking, message structures and so on have been omitted because they are irrelevant to the algorithm at large and may be different from each implementation. Even though Paxos is conceptually simple, it has been shown to be non-trivial to implement correctly [5].

Our implementation has therefore been based [26], which in turn is a condensed form of the ones given in [32, 21]. Algorithms 1 and 2 implement full, non-Byzantine Paxos. We will simplify these later on. These algorithms do not perform *leader election.*

Algorithm 1 Full, classic crash Paxos — Proposer c (leader)

```
$A$                                                    ▷ Set of acceptors
$crnd \leftarrow 0$                                    ▷ Current round (unique)

on $\langle \texttt{TRUST}, c \rangle$ from $\Omega_c$ do
    $crnd \leftarrow$ pickNext($crnd$)                 ▷ Phase 1a
    $MV \leftarrow \emptyset$                          ▷ Set of $\langle round, vote\ value \rangle$ tuples
    send $\langle \texttt{PREPARE}, crnd \rangle$ to $A$

on $\langle \texttt{PROMISE}, rnd, vrnd, vval \rangle$ from acceptor $a$ do     ▷ Phase 2a
    if $rnd = crnd$ then
        $MV \leftarrow MV \cup \langle vrnd, vval \rangle$
        if $|MV| \geq 1 + \lfloor |A|/2 \rfloor$ then
            if $(vrnd = \bot)\ \forall\ \langle vrnd, vval \rangle \in MV$ then
                $cval \leftarrow$ pickAny()
            else
                $cval \leftarrow$ pickLargest($MV$)
            send $\langle \texttt{ACCEPT}, crnd, cval \rangle$ to $A$
```

Algorithm 1 on the current page is for the *proposer role.* `TRUST` messages are received during phase 1a, and `PROMISE`s are received during phase 2a. See [21] for details.

First we initialize the proposer. We assume it already knows (the addresses of) the acceptors A. The current round number *crnd* is initialized to zero, although later we will implement a version that guarantees a sequence of unique *crnd*s (shown in equations 3.1 and 3.2).

Upon receiving a `TRUST` message from Ω_c, it will pick the proposal number larger than *crnd*, reset the set of $\langle round, vote\ value \rangle$ tuples and then send a `PREPARE` message to all acceptors A. Finally, it will send a `PREPARE` message to all acceptors. The implementation of **pickNext** is found in algorithm 3, on the following page.

When it receives a `PROMISE`, it will first check that it is a reply that belongs in the current round. It will store the acceptor's *vrnd* and *vval*, then it will check whether it has received accepts from a *majority* of acceptors. It will also check that it has not already accepted this message. If it has, then it will simply resend the ACCEPTed value. If it hasn't, it will go on and send the next value from its buffer. We don't show implementations for **pickAny** or **pickLargest**, but one can imagine these are simple queues of messages to process.

Algorithm 2 Full, classic crash Paxos — Acceptor a

```
$P$                                                  ▷ Set of proposers
$L$                                                  ▷ Set of learners
$rnd \leftarrow 0$                                   ▷ Highest round seen
$vrnd \leftarrow \bot$                               ▷ Round in which value was last accepted
$vval \leftarrow \bot$                               ▷ Value last accepted

on $\langle \texttt{PREPARE}, n \rangle$ from proposer $c$ do      ▷ Phase 1b
    if $n > rnd$ then
        $rnd \leftarrow n$
        send $\langle \texttt{PROMISE}, rnd, vrnd, vval \rangle$ to $c$

on $\langle \texttt{ACCEPT}, n, v \rangle$ from proposer $c$ do    ▷ Phase 2b
    if $n \geq rnd \land n \neq vrnd$ then
        $rnd \leftarrow n$
        $vrnd \leftarrow n$
        $vval \leftarrow v$
        send $\langle \texttt{LEARN}, n, v \rangle$ to $L$
```

Algorithm 2 on the current page is for the *acceptor role.* It will act on PREPARE and ACCEPT messages. When PREPARE is received, it will simply check if the round number is larger than the highest round it has seen so far. If so, it will update its round number *rnd* and send back a PROMISE to the acceptor.

Upon receiving an ACCEPT message, it will again check if the round number is equal to or larger than the largest it has already seen, and that it has not already accepted that value. If this is the case, it will update its values and send a LEARN message to *all* learners.

The full Paxos algorithm for the *learner role* has been omitted, but we will detail one for the simplified version.

3.2.2 Simplified Paxos

Because this is a feasibility study, and we have limited time, we will only focus on the most common case of message exchanges in Paxos. This is what [21, 20] refer to as *phase two*, and only involves *accept* and *learn* messages. Implementing phase one (*prepare* and *promise* messages) *correctly* is very hard, and we will assume from now on that this has already taken place. This leads to the simplified version of Paxos that we present here.

Refering to algorithms 1 and 2 above, we see that we can remove *vrnd* altogether. Also, each Paxos node in our system will take on all three roles, so we don't need separate sets for the acceptors, proposers and learners. We therefore replace the sets A, P and L with the single set N to refer to all Paxos nodes.

While *crnd* is static for phase two, we will keep it so that future implementations can more easily extend the solution to support phase one. Instead of initializing it to zero, we wil set it to the node's unique identifier. In **pickNext** (algorithm 3) we will increment *crnd* with the total number of Paxos nodes, $|N|$. This is a common technique for ensuring that every *crnd* will be unique, with the added benefit that we can deduce $node_{id}$ from it (equation 3.1).

Given

$$crnd_i = \begin{cases} n_{id} & \text{for } i = 0 \\ crnd_{i-1} + |N| & \text{for } i \geq 1 \end{cases}, n \in N \tag{3.1}$$

then, by definition,

$$n_{id} \equiv crnd \pmod{|N|} \text{ for } n \in N \tag{3.2}$$

where n is the node and N is the set of all nodes. This leads to our definition of **pickNext** in algorithm 3.

Algorithm 3 Definition of **pickNext** based on equation 3.2

```
$N$                                   ▷ The set of all Paxos nodes
$n_{id} \leftarrow$ Unique Paxos node id
$crnd \leftarrow n_{id}$              ▷ Replaces initialization of crnd in algorithm 1

function pickNext
    return $crnd + |N|$               ▷ Unique per equation 3.2
```

Equation 3 and algorithm 3 could also be used to uniquely tag each incoming client packet with a unique identifier (see section 4.4).

As we only intend to show that we can implement `ACCEPT` and `LEARN`, we can ignore `TRUST`, `PROMISE` and `PREPARE` messages. This leaves us with simpler algorithms that should be easier to implement.

Algorithm 4 Simplified algorithm for processing `ACCEPT` messages

```
$N$                                   ▷ The set of Paxos nodes
$rnd \leftarrow 0$                    ▷ Current round number

on $\langle$ACCEPT$, n, seq, v\rangle$ from $leader$ do
    if $n \geq rnd$ then
        $slot[n, seq].hrnd \leftarrow n$
        $slot[n, seq].vval \leftarrow v$      ▷ The client packet
        for $node$ in $N$ do
            send $\langle$LEARN$, n, seq, v\rangle$ to $node$
```

Algorithm 5 Simplified algorithm for processing LEARN messages

```
$H$                                   ▷ The set of end-hosts connected to this switch

on ⟨LEARN, $n$, $seq$⟩ from acceptor do
    if got_majority($n$, $seq$) then
        process_queue($n$, $seq$)
```

Note that we have introduced a new parameter *seq* in algorithms 4 and 5. This is because we have also incorporated *multi-Paxos slots* [32], which requires us to store sequence numbers *seq* as well. While the Paxos leader decides the order in which messages should be processed, it is the learner who actually makes sure that they are delivered in order (algorithm 5).

In algorithm 5, the the procedure **process_queue** will process the queue in-order of increasing sequence numbers, starting from the lowest that has not been processed. It will process each message (i.e., send the stored packet in *vval* out to its final destination) until a slot has not been learned (meaning it has received enough learn-messages for this sequence number in this round) or until there is a gap in the sequence numbers. If there is a gap, it means that it has received messages in the wrong order, and it must therefore wait until it has a consequtive sequence of slots.

The number of required learns is simply $1 + \lfloor |N|/2 \rfloor$, or one more than (the floor of) half the number of nodes: For three nodes, this number is two.

Algorithm 6 Algorithm when leader receives a client packet

```
$N$                                   ▷ The set of Paxos nodes

on ⟨CLIENT, $v$⟩ from client do
    $seq \leftarrow seq + 1$                    ▷ Sequence number
    for node in $N$ do
        send ⟨ACCEPT, $crnd$, $seq$, $v$⟩ to node
```

Finally, we need a way to catch packets from clients that we want to trigger an ACCEPT for. Shown in in algorithm 6, we do not specify how we identify a client packet. In our actual implementation, we deduce which port on the switch the clients are on by using an L2 Learning Switch (section 4.2 on page 36).

3.3 OpenFlow

Based on our previous discussion of network topology, we will here look at how we can build such a system using OpenFlow.

We will start by looking at what capabilities OpenFlow can offer us, then see how we can suit it to fit our needs.

3.3.1 Capabilities in OpenFlow

Now that we have discussed the main algorithms and our simplification of them, we must take a look at what OpenFlow can offer us to reach our goals.

To see how we can enable Paxos functionality in OpenFlow, we need to take a look at what features it can provide us. There are several versions of the OpenFlow specification, so we'll review the differences in each one.[2]

Naturally, we could implement the whole Paxos algorithm in the controller itself. Doing so should be quite trivial: One could modify an existing implementation and make it use OpenFlow to transmit packets between switches. However, that would be very inefficient compared to running the entire algorithm, or parts of it, in the switch.

In the tables below, you can see what version 1.0 offers in terms of core functionality. Some details have been omitted in favor of giving a clear overview. For details, see the full specification [9].

What's most important in 1.0, compared to later versions, is that it only has *one* flow table and only supports IPv4. Other than that it has counters per table, per flow, per port and per queue. The headers that can be used for matching packets are listed in table 3.1 on the following page and the actions in table 3.2 on the next page.

By *transport* address and port, we mean transmission control protocol (TCP) or uniform datagram protocol (UDP), depending on what packet is currently matched.

The specification requires compliant switches to update packet checksums when modifying fields that require it.

3.3.2 Limitations in OpenFlow

By looking at what OpenFlow versions 1.1–1.3 offer, one can see that we can't really make use of any of the added functionality for running Paxos. What we need is the ability to run programs on the switch, which is something OpenFlow does not support at all. Neither do their action primitives add up to anything that could be used for remembering state (such as the round and sequence number) or executing if-then-else statements. We also need to store messages somewhere, so they can later be sent out in the correct order.

[2] Unfortunately, the most widely supported version of OpenFlow in simulators and controllers seem to be OpenFlow version 1.0.

Header-field
Ingress port
Ethernet source address
Ethernet destination address
VLAN ID
VLAN priority
IP source address
IP destination address
IP protocol
IP type-of-service (ToS) bits
Transport source port
Transport destination port

Table 3.1: Header-fields that can be matched in OpenFlow 1.0.

Action	**Required**	**Options**
Forward	Required	To all To controller To local switch To flow table To port
Forward	Optional	Normal Flood
Enqueue	Optional	Can be used to implement, e.g., QoS
Drop	Required	Drop packet
Modify-field	Optional	Set or replace VLAN ID Set or replace VLAN priority Strip any VLAN header Replace Ethernet source address Replace Ethernet destination address Replace IPv4 source address Replace IPv4 destination address Replace IPv4 ToS bits Replace transport source port Replace transport destination port

Table 3.2: Actions in OpenFlow 1.0.

One possible solution would be to insert a lot of flow table entries that would match on specific round and sequence numbers. But that would require a lot of flow entries, and would be neither an elegant or practical solution.

OpenFlow does, however, offer us the ability to implement Paxos entirely in the controller. We also want to see if we can move parts of Paxos down to the switch itself.

To remember round and sequence number, we could possibly have used the meta-data that is available in later versions of OpenFlow. However, metadata only exists as the packet is processed in the pipeline of flow tables, and is erased when the packet actions are applied at the end. The switch simply needs to store the state somewhere.

Mininet seems to support whatever version of OpenFlow that Open vSwitch uses, as this is what it uses as a switch. Open vSwitch supports OpenFlow versions 1.0—1.3 almost fully, but support for 1.4 is flaky, and may crash. So 1.4 is out of the question.

The most obvious component to look at is POX, our controller framework in Python, which only supports OpenFlow 1.0.[3]

But the major point for our decision is what OpenFlow can offer us. There simply is no way of executing general code, and there is no way to remember state.

We have therefore implemented Paxos entirely on the controller, then modified Open vSwitch to provide Paxos as a new, primitive OpenFlow action.

3.3.3 Conclusion

We have seen that OpenFlow does not seem to offer us the capabilities needed to implement Paxos in the switch.

Thus, we have decided to create a prototype Paxos controller, then implement it on the switch.

[3] It does seem to support some *Nicira extensions*, though. These are extensions that were originally added to early OpenFlow versions, but much of it has been implemented in later versions. There is also a fork of POX (and other software projects) written by CPqD that adds support for newer OpenFlow versions, but we haven't looked at it.

Chapter 4

Implementation

Based on the design in chapter 3, we will now look at implementation details.

To be able to run Paxos in the switch, we must first extend the OpenFlow Switch Specification with a new *Paxos action*. This will allow us to freely *compose* flows that run the Paxos algorithm as one part of their actions. Finally, we must modify Open vSwitch so that we can run the new action.

4.1 Extending the OpenFlow Specification

As discussed earlier in chapter 3.3, it would be impractical to attempt to use existing actions in the OpenFlow specification to implement the Paxos algorithm. The OpenFlow specifications, as of version 1.4 [15], are backward-compatible, meaning that a newer OpenFlow version will support all features in older ones. We will therefore choose to extend version 1.0 [9], because it was the first public version and therefore the most widely supported.

The idea is to add a new *Paxos action* with a parameters specifying whether to run the *On Client*, *On Accept* or *On Learn* parts of the Paxos algorithm given in chapter 3.2.2 on page 26. As shown in 2.1 on page 15, we can then specify precisely what kind of events we want to trigger Paxos ordering for and combine that with other actions, such as which port the output should go to.

The part of the specification we need to extend is the *Flow Action Structures* [9, pp. 21–22], and it will be an *optional* action [9, pp. 3–6]. Listing 4.1 shows the modification made to the C enumeration type `ofp10_action_type` from the Open vSwitch source code.[1] We have included listing listing:ofp10.action.type as-is because this is how it is defined in the published OpenFlow specification [9]. The listing is identical to the official specification, except for the number suffix in `OFPAT10`.

Listing 4.1: Adding the `OFPAT10_PAXOS` action to the OpenFlow specification

```
enum ofp10_action_type {
```

[1] `ovs/include/openflow/openflow-1.0.h`

```
    OFPAT10_OUTPUT,             /* Output to switch port. */
    OFPAT10_SET_VLAN_VID,       /* Set the 802.1q VLAN id. */
    OFPAT10_SET_VLAN_PCP,       /* Set the 802.1q priority. */
    OFPAT10_STRIP_VLAN,         /* Strip the 802.1q header. */
    OFPAT10_SET_DL_SRC,         /* Ethernet source address. */
    OFPAT10_SET_DL_DST,         /* Ethernet destination address. */
    OFPAT10_SET_NW_SRC,         /* IP source address. */
    OFPAT10_SET_NW_DST,         /* IP destination address. */
    OFPAT10_SET_NW_TOS,         /* IP ToS (DSCP field, 6 bits). */
    OFPAT10_SET_TP_SRC,         /* TCP/UDP source port. */
    OFPAT10_SET_TP_DST,         /* TCP/UDP destination port. */
    OFPAT10_ENQUEUE,            /* Output to queue. */
    OFPAT10_PAXOS,              /* Extension: Run Paxos algorithm. */
    OFPAT10_VENDOR = 0xffff
};
```

The Paxos action has only one parameter: Which part of algorithm 3.2.2 to run. The structure of this parameter is given in listing 4.2 and its its possible values are defined in table 4.1. All action structures are required to start with the `type` and `len` fields.

Listing 4.2: The `OFPAT10_PAXOS` parameters

```
struct ofp10_action_paxos {
    ovs_be16 type;            /* Required: OFPAT10_PAXOS. */
    ovs_be16 len;             /* Required: Length is 8. */
    ovs_be32 paxos_event;
};
OFP_ASSERT(sizeof(struct ofp10_action_paxos) == 8);
```

As we can see, `paxos_event` is encoded as a big-endian, unsigned 32-bit integer. Its possible values are given in table 4.1.

Value	**Meaning**	**Algorithm**	`ovs-ofctl` **argument**
`0x7A01`	Run “On Accept”	4	`paxos:onaccept`
`0x7A02`	Run “On Learn”	5	`paxos:onlearn`
`0x7A40`	Run “On Client”	6	`paxos:onclient`

Table 4.1: Possible values for `paxos_event` in listing 4.2

The values in table 4.1 have been chosen to correspond to the Ethernet types given in table 4.2 on page 37, although they could have been simply zero, one and two. The last column contains the command-line arguments that will be accepted by `ovs-ofctl` when adding flows.

Because of the thesis scope, we have only added a single action parameter `paxos_type`. In a production environment, however, one would likely need several more. For example, it could be useful to distinguish between different *sets* of Paxos nodes so they could operate independently of each other on the same network. Here, we have only *one* set of Paxos nodes who all have the same leader.

4.1.1 Modifications to Open vSwitch

As mentioned in section 2.2 on page 17, the component in our system that actually executes OpenFlow actions is *Open vSwitch*. To fully implement the new Paxos OpenFlow action, we need to do this in Open vSwitch. Details can be found in sectio A.3.2 on page 65.

Looking at table table 4.1 on the previous page, the rightmost column (**`ovs-ofctl` argument**) contains arguments to the Open vSwitch command-line tool `ovs-ofctl`, that can be used to program Paxos actions as smaller parts of bigger flows.

To demonstrate how elegantly one can set up flows that use Paxos ordering, consider the below example for installing a flow on the switch `S1`.

```
sudo ovs-ofctl add-flow S1 \
               in_port=3,dl_type=0x7a40,actions=paxos:onclient,output:5
```

The above command installs a new flow entry on `S1`, matching packets coming in on port 3 with the Ethernet type `0x7a40`. Referring to table 4.1 on the preceding page, we see that this flow will match on packets of type `CLIENT`.

Furthermore, under `actions=`, we instruct Open vSwitch to run the Paxos action with the parameter `onclient`. This means that for matchin packets, Open vSwitch will dispatch the packet to the *on client* function (the argument `paxos:onclient`), described in chapter 4.4 on page 39 and algorithm 6 on page 28. This algorithm will output an accept message to output port 5 (`output:5`) If we want to explicitly set the destination address of the packet, one can just prepend the output with the modification action `mod_dl_dst=a1:b2:c3:d4:e5:f6`. To send out on several ports, one just needs to add more `output:<N>` actions, or the packet can be flooded on all ports with `output:flood`.

What we are doing here is programming the switch's flow table using Paxos primitives as constituent elements. For the actual implementation, we refer to the appendix, section A.3.2 on page 65.

We have implemented all of the actions in table 4.1, including multi-Paxos storage of packets in slots, but with the important exception of the queue processing (algorithm 5, section 3.2.2).

The above command translates client packets to Paxos `ACCEPT` packets. For a Paxos node on another switch, we can simply use the `paxos:onaccept` action. Since switch S_2 of figure 3.4 on page 23 may receive Paxos messages from both S_1 and S_3, we may want to only react on packets that are explicitly addressed to S_2. To do so, assuming the MAC-address is `22:22:22:22:22:22`, one may simply add a matching pattern for it, along with the obligatory check for the Ethernet type field corresponding to an accept message (`0x7A01`):

```
sudo ovs-ofctl add-flow S2 \
    dl_src=11:11:11:11:11:11,\         # match from leader S1
    dl_dst=22:22:22:22:22:22,\         # match S2 MAC-address
    dl_type=0x7a01,\                   # match ACCEPT message
    actions=paxos:onaccept,\           # run "On Accept"
    mod_dl_src=22:22:22:22:22:22,\     # set source MAC address
    mod_dl_dst=33:33:33:33:33:33,\     # set destination MAC to S3
    output:5,\                          # output to port 5
    mod_dl_dst=11:11:11:11:11:11,\      # set destination MAC to S1
    output:1                            # output to port 1
```

The flow above is an *actual* flow that we used—and verified to work—in our network simulator. If all conditions of algorithm 4 are met, this flow wil send out a learn message to S_1 and S_3.

Comparing this with writing equivalent flows as procedures in Python, this is *vastly* easier to do. An *excerpt* from the code for accept-hanling in the Python controller is given below.

Listing 4.3: Shortened excerpt of Python code for handling Paxos accept messages

```
def on_accept(self, event, message):
  n, seqno, v = PaxosMessage.unpack_accept(message)
  src, dst = self.get_ether_addrs(event)

  # From leader?
  if src != self.leader.mac:
    return EventHalt # drop message

  slot = self.state.slots.get_slot(seqno)

  if n >= self.state.crnd and n != slot.vrnd:
    slot.vrnd = n
    slot.vval = v

    # Send learns to all
    for mac in self.state.ordered_nodes(self.mac):
      self.send_learn(mac, n, seqno, self.lookup_port(mac))

  return EventHalt
```

The code in listing **??** is a shortened version of the actual implementation. Of course, we have had to actually *implement* the above code in equivalent C code in Open vSwitch, but the big gain is that the flows are happening on the switch, and requires no upcall to the controller.

As discussed in 2.1.1 on page 16, well-designed controllers should install flows incrementally as they learn the network topology. We must therefore first implement a system that works entirely without flow entries. (Dette er en del av design-diskusjon, vi skal bare implementere det her).

Next, we will implement flows in the system. As discussed previously, this requires an extension to the OpenFlow-protocol and the switch software we use, Open vSwitch.

We will implement algorithms 4 and 5 in a combination of OpenFlow matches and its extensions that were introduced in **??** on page ??.

4.2 An L2 Learning Switch in OpenFlow

When you write an OpenFlow controller, the flow table is empty and all packets will by default be delivered to the controller via an *upcall*.

The controller must then decide what to do with the packets. If we don't implement any sort of forwarding behaviour for the packets, none of the hosts will be able to communicate.

So our system will need a forwarding mechanism below the level where Paxos operates. The simplest system is just to implement a *hub*: For each packet coming in to the switch, flood it (or, *rebroadcast*) to all ports except the input port. Each node receiving packets will silently drop those who are not addressed to them (algorithm 7).

Algorithm 7 An L2 hub algorithm

```
on packet $e$ from port $p$ do
    flood $e$ except port $p$          ▷ Send packet out on all ports except $p$
```

A slightly better approach is to implement an OSI link-layer (L2) learning switch. The difference from the flood-to-all hub above is that we create a table that maps MAC-addresses to ports and then forward each packet to a single port. We then achieve less traffic on the network. Note that an L2 switch does not explicitly state that it is operating on Ethernet frames and MAC addresses. This is because we have reused the algorithm for IP addresses.

As we build up this table we could also install flow table entries so that the switch will be able to forward packets by itself.

Algorithm 8 An L2 learning switch algorithm for an OpenFlow controller

```
$M \leftarrow \emptyset$                                  ▷ Map of $address \rightarrow port$

on packet $e$ from port $p$ do
    $M \leftarrow M \cup \langle e_{src},\ p\rangle$          ▷ Learn port $p$ for $e_{src}$ (source MAC-address)

    if $\{\exists q : \langle e_{dst},\ q\rangle \in M\}$ then       ▷ See if we know the destination port $q$
        add flow(for packets from $e_{src}$ to $e_{dst}$, forward to port $q$)
        add flow(for packets from $e_{dst}$ to $e_{src}$, forward to port $p$)
        forward $e$ to port $q$ for $e_{dst}$ in $M$
    else
        flood $e$ except to $p$                         ▷ Act as hub; algorithm 7
```

As you can see, algorithm 8 will need to run at least twice before it will know both the source and destination ports for two MAC-addresses.

Assume that we are running algorithm 8 on an OpenFlow controller connected to a switch whose flow table is empty. Recall that packets who do not match any flow table entries are upcalled to the controller.

If we now send an *ICMP ping packet* from host a to host b, the controller will learn which port host a is on, but will not know the port for host b yet. It will then flood the packet out on all ports except the input port.[2]

Host b will then receive the packet and send an ICMP ping reply packet addressed to host a. When the controller receives the reply, it knows the port for host a and can then *forward* the packet to its port instead of flooding it. Simultaneously, it will learn which port host b is on. At this point the controller can decide to install forwarding flows in the switch so that packets are automatically forwarded. One such flow can be to match on source address a, destination address b with the action to forward to the port for host b. Another flow can handle the reverse case.

Algorithm 8 uses a well-known implementation technique for learning switches. The one we have implemented is based on the one given in the OpenFlow tutorial [29]. There are additional checks that we do not perform, like not installing flows that echo packets to the incoming port (see, e.g., the pseudo-code in figure 3 of [1]).

There is another very important point to be made here: *Packets that are handled by a flow may not be seen by the controller.* In the above algorithm, it could be tempting to install a forwarding flow as soon as we know which port a host is on. Now consider the ping example involving hosts a and b. The controller would learn the port for host a and install a forwarding flow. But when host b replies, the switch would silently forward the packet to host a. The controller would never learn which port b is on, until host b sends a packet addressed to someone other than host a. The lesson is that building controller algorithms may have corner cases that are not easily recognized.

4.3 Paxos Message Wire Format

When exchanging Paxos messages between switches, we need a way to identify them. A well-known use of OpenFlow is to create entirely new, non-IP protocols by matching on fields in the Ethernet header [25, Example 4, p. 73]. We will tag Paxos messages with special values in the *Ethernet type*-field. This field is two octets wide (i.e., 16 bits), so we can use the most significant one to mark packets as carrying Paxos messages, and the least significant one for the kind of Paxos message (table 4.2).

	Ethernet Type Field 16 bits	
Message Type	**Most Significant**	**Least Significant**
`PAXOS JOIN`	`0x7A`	`0x00`
`PAXOS ACCEPT`	`0x7A`	`0x01`
`PAXOS LEARN`	`0x7A`	`0x02`

Table 4.2: Encoding of `PAXOS` messages in the *Ethernet type* field.

There is no particular reason for the specific values used in table 4.2, but since

[2] Hosts who receive unsolicited packets will silently drop them, unless they are running in *promiscuous mode* or similar, capturing all packets.

ACCEPT and LEARN messages share the first parameters, they could be bits that could both be turned on to send a combined ACCEPT-and-LEARN message. If both bits are zero, it becomes a JOIN message. We cannot use values below 0x600, because that is used by Ethernet to signify payload size.

Using the Ethernet type for identifying Paxos messages makes it very convenient to match the different messages in OpenFlow's flow tables.

We now have to define the payload structure for Paxos messages. Table 4.3 defines the parameters each message type will contain. It will consist of consecutive 32-bit values for storing parameters, followed by the a full client packet in ACCEPT messages. Each type of message will trigger the corresponding algorithms in 3.2.2 on page 26. The JOIN message is discussed in chapter 4.3.3.

...	**Ethernet Type**	**Parameters**		**Payload**
	16 bits	32 bits	32 bits	...
...	`PAXOS JOIN`	$node_{id}$	MAC source	
...	`PAXOS CLIENT`	*ignored*	*ignored*	v (client packet)
...	`PAXOS ACCEPT`	n (round)	seq (sequence)	v (client packet)
...	`PAXOS LEARN`	n (round)	seq (sequence)	

Table 4.3: The structure of L2 Paxos messages. Not shown her is the preceding Ethernet fields.

At this point we should discuss what will happen when the round or sequence number reaches the maximum number possible. A good solution would be to program the Paxos nodes to allow values to roll around to zero when passing the maximum value of $2^{31} - 1$, so that we would never run out of numbers. This is a detail that is irrelevant for our stated goals, but a complete implementation should naturally allow for infinite sequences.

4.3.1 The PAXOS ACCEPT Message

The ACCEPT message contains the round and sequence numbers for the embedded client packet. They correspond to the variables n, seq and v of the Paxos algorithms in chapter 3.2.2 on page 26, respectively.

It will start algorithm 4 and send out LEARN messages, if the conditions are right.

Since it shares the first parameters with the LEARN message, and since only the leader send them out, a triggering ofshare the first parameters with the The ACCEPT message share the first parameters with the LEARN message.

4.3.2 The PAXOS LEARN Message

The LEARN message triggers algorithm 5 on page 28.

We have implemented this using multi-paxos, which will then update slots with the number of learns.

4.3.3 The `PAXOS JOIN` Message

When the system starts up, the switches need to announce themselves to each other and learn which ports they are on. To avoid having to rely on configuration files, we built a very simple system for announcing the presence of Paxos nodes, loosely based on the address resolution protocol (ARP).

Each node will send out a `JOIN` containing its own node ID and MAC-address,, sending it out on all ports with the Ethernet broadcast destination of `ff:ff:ff:ff:ff:ff`.

When receiving a `JOIN`, the node will store the node ID and MAC-address in a table and pass the MAC-address and source port number ot the L2 learning switch as well. If the MAC-address is not already in the table, it will reply to the sender with a `JOIN`.

This will continue until a node knows about at least two other nodes—the minimum required for Paxos execution. If it does not know enough nodes after some seconds, it will send out a new `JOIN` broadcast. No other Paxos messages will be processed until enough nodes are known.

Since we are only interested in Paxos phase two, we do not perform any leader election, but it would be natural to start Paxos leader election with prepare and promise right after the `JOIN`-phase. In our setup, we have simply designated a switch as leader, and we do not support new nodes to join the Paxos network.

4.4 The `PAXOS CLIENT` Message

The `PAXOS CLIENT` message is used for distributing client packets among the Paxos nodes. To keep consistent with the established structure, the client packet itself starts at an offset of 64 bits from the end of the Ethernet type field. The two preceding parameters are unused.

Its intended use is to forward client packets to the Paxos leader, who will then issue an `ACCEPT` message. But this means that some Paxos nodes will see the same message several times. Referring to figure 3.4 on page 23, if switch S_3 receives an incoming client packet, it will forward it in a `PAXOS CLIENT` message to S_2, who will forward it to the leader S_1. S_1 will then send back a `PAXOS ACCEPT` to S_2, whose L2 switch will forward it to S_3 again. All containing the same client packet.

Clearly, this design could be improved. One possibility would be to generate a unique identifier for each incoming client packet. Each `PAXOS CLIENT` message would carry it, and each node would receive a copy of the message, storing it in a table with the identifier as key. The `PAXOS ACCEPT` message would then contain this key instead of the full client packet. The identifier could be generated on each node by using the same technique as for *crnd* in equation 3.2 on page 27. Again we must stress that—while tempting—we have decided not to spend time on building an optimal system. Our goal is to build a distributed replication system using Paxos on the switches, and along the way we uncover important result such as these that could be investigated further.

4.5 Handling Incoming Client Packets

First, when a switch gets a client packet it needs to add flow table entries that forward it to all the other switches. We need several OpenFlow matching rules for all of this to work. Note that all Paxos nodes except the leader will be called for *Paxos slaves* from now on.

Switch	**Flow Table Entry**
Leader	Store packet Send `ACCEPT` to slaves.
Slaves	Forward to leader

Table 4.4: OpenFlow flow table entries.

Each switch need to store the full client packet and then forward it to the other switches.

We also need entries for matching Paxos messages and their respective actions. This is done by inserting entries that match on Ethernet type `PAXOS` and ingress port from the leader. The action will be to go to a new entry that looks at what kind of Paxos message we have received.

Finally, when matching on Paxos message types, we would execute special code using the new `run_code`-action (see **??** on page ??) and forward packets based on the return value from the code.

We also need new OpenFlow protocol messages so that the controller is able to install flows with these new actions. However, to save time, we will simply install these flows by using the `ovs-ofctl` command-line program from the Open vSwitch-distribution. While Open vSwitch has been modified to support the new OpenFlow actions in the switch-to-controller protocol, we would have to modify the POX framework to be able to parse such messages, update its feature table and so on. This is considered trivial to do, but time consuming and irrelevant to our task.

4.6 Paxos in the Controller

Our first step will be to implement simple Paxos [21] entirely in a controller.

The aim is to show that a topology with a Paxos-enabled controller will satisfy the requirements of Paxos.

We will use Mininet [22] for running the network simulation. It uses Open vSwitch. We will use and POX [24] for implementing a Paxos in an OpenFlow controller. POX is part of the NOX-project [17]. They both use Python [33] as the implementation language, which means we can share some code between them. Both projects are mature and easy to use. The Mininet simulation itself will run on a virtual machine using VirtualBox.

There are many things to consider when building a working Paxos controller. We want to provide an internal ordering of packets coming from the wide-area network (WAN). But it would not be wise to do so for *all* packets. For instance, we need to make sure that ARP-packets are working properly, or else clients will not be able to look up addresses needed for transmission. If we perform network address translation (NAT) and Paxos-ordering on these packets, we quickly need to perform all kinds of tricks to ARP. In essence, we are dragged into the minute details of ARP. The same goes for basically any other protocol that may show up on the network, e.g., Internet control message protocol (ICMP). This is clearly out of scope for this thesis, so we will simply let ARP and ICMP pass through the entire network in normal fashion, letting the L2 switch forward them to each hop.

Our aim is to mark packets from the WAN with the special `PAXOS CLIENT` Ethernet type and perform ordering for each one. It means that, depending on the MTU, we would be ordering a lot of packets.

First we need to be able to route Ethernet frames to their correct destination. For this we need an L2 learning switch (see 4.2 on page 36).

Next, we need to decide which port represents the WAN. We have chosen to create a separate switch and controller for this. The reason is that it could perform network address translation (NAT) so that the Paxos network would be reachable by a *single* IP-address. We will not build a complete NAT, however, but leave it there as a potential point for implementing such behaviour.

Now, the Paxos-controllers will inspect the packet headers and act only on those with the correct `PAXOS` Ethernet type. If we see a `PAXOS CLIENT` message, the leader will initiate an `ACCEPT` and, subsequently, `LEARN`s.

Note that this happens for *every* TCP or UDP packet. Depending on the MTU, we could potentially perform ordering on a lot of packets. This is clearly a downside to the system. Consider a TCP session. First it will need to establish the TCP-connection with a SYN, then SYN-ACK and a final ACK. All of these will be handled and ordered by the Paxos controllers. To provide a complete mirroring solution,[3] we need to change destination addresses in the Ethernet and Internet protocol (IP) headers when processing each packet.

There is another important consideration when it comes to switches. A switch is supposed to be *self-autonomous*, meaning that it should not require any particular configuration to be able to function. One just plugs it into the net, and it should start learning which ports each MAC-address can be reached on.

This also applies to the Paxos controllers. For them to implement mirroring, in the sense that each packet coming in from the WAN is sent to each end-host, it needs to modify Ethernet and IP addresses for each packet. When it starts up, it doesn't know either of these, and has to learn along the way. This complicates matters somewhat, and using a configuration file with a full map of the network would be contrary to the principle of self-autonomy. What we have done is to let the L2 switch track both Ethernet and IP-addresses. When each Paxos node announces itself, we learn which ports the Paxos nodes are on. This is

[3] Replication across end-hosts.

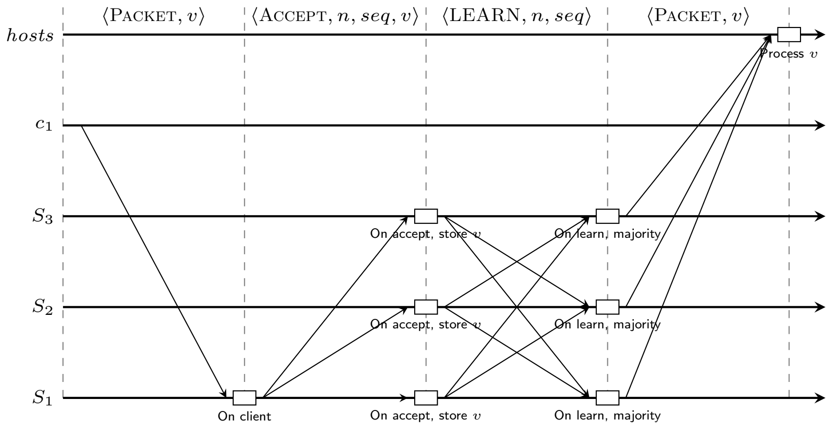


Figure 4.1: A client c_1 sends a request to the system. The message is forwarded to the Paxos leader S_1, which sends the client packet as a parameter to an accept message, incrementing the current sequence number seq. All nodes store the packet upon receiving an accept. Learns are exchanged, and when a node has received learns from a majority of the nodes, the packet will be processed in order and sent to each end-host.

important for the WAN-controller, who needs to shuttle packets between each network.

Each Paxos controller will then know which port the WAN is on (only applies for leader) and which ports other Paxos controllers are on. It can therefore deduce that the remaining ports are links to their end-hosts.

Messages going from the client to the end-hosts are ordered by the Paxos system. But packets going the other way, from end-hosts to the clients, pass through the network unhindered. Consolidating replies is out of scope for the purpose of implementing steady-phase Paxos, and we have therefore not looked more into this matter.

4.7 Example of a Full Networking Flow

In figure 4.1, we show the complete flow of a client packet through the system, running through the Paxos algorithm on the switches and finally being processed and forwarded to the end-hosts. Not shown here is any reply from these end-hosts.

What we have accomplished here is using Paxos for ordering the client requests down to the hosts, so that each host will receive packets in the same order. To test that the hosts have received packets in the same order, we have run a simulation where several clients send packets to them and then compare their output checksums using the SHA-256 algorithm.

4.8 The Final Set of Flow Entries

In tables 4.5 and 4.6, we show the final table of events that the Paxos leader and slaves will handle, respectively. The tables have been implemented both on the Paxos controllers and the Paxos switch (Open vSwitch), except for the process queue on the switch, making it impossible to get results for round-trip times.

Match	**Action**
Packet from client	Stamp with PAXOS CLIENT Ethernet type Forwward to leader
Packet from end-host	Forward to client
PAXOS JOIN	Store MAC address and node id of switch (*Paxos-on-controller only*)
PAXOS CLIENT	Execute on-client Send PAXOS ACCEPT to all nodes with a copy of the packet
PAXOS LEARN	Execute on-learn, forward client packet to hosts

Table 4.5: The final event table for the Paxos leader.

In the case of the controller, these events have been implemented as procedure calls, dispatching on Ethernet types signifying different Paxos messages. To forward messages on the network, OpenFlow commands were sent down to the switch.

For the implementation on the switch (Open vSwitch), these tables have been implemented as flow entries in the flow tables. Everything except the processing of the queue were imple,ented, making it impossible to measure round-trip times.

Match	**Action**
Packet from client	Forward to leader Optionally, set unique packet ID
Packet from host	Forward to client
PAXOS JOIN	Store MAC address, node id and leader-flag
PAXOS ACCEPT from leader	Execute program on-accept Store packet Send learn to all, if applicable
PAXOS LEARN	Execute program on-learn Run process queue, forwarding to hosts in-order

Table 4.6: The event table for the Paxos slaves.

Some details have been omitted from tables, and we refer to the source code for further detail (see the appendix, chapter A.2).

Chapter 5

Performance Analysis

We will now look at the performance profile of various network configurations.

As we are running these benchmarks on the Mininet simulator, there are natural limits on how realistic the results will be. However, we should be able to get good *relative* results. Therefore we will first need to establish a baseline to which we will compare the Paxos implementation.

Instructions on how to run these tests are given in A.4 on page 67.

5.1 Baseline — ICMP Ping on L2 Learning Switch

All our controllers use the L2 learning switch from chapter 4.2 to forward packets on the network. We can therefore use a setup with only these L2 learning switches as a basis for our performance tests. We will use the topology in figure 5.1 and send ICMP ping packets from one end of the network to the other. The reason for removing the fall-back link between S_1 and S_3 in figure 3.2 is to prevent the packets from taking different paths on the network.

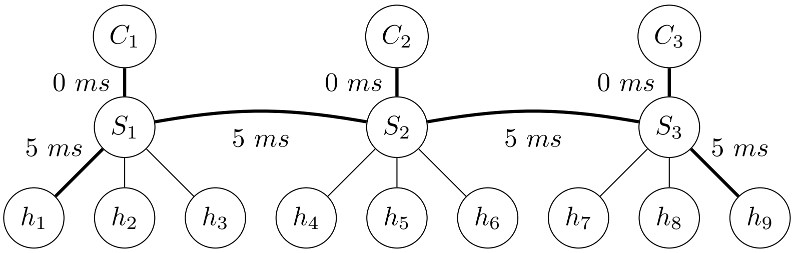


Figure 5.1: Baseline topology with three switches S and their controllers C. Node h_1 will send ICMP ping packets to h_9. The packets will go through four links with a configured latency of 5 ms and back again. We assume that link-latencies between switches and controllers are practically near zero.

The first ICMP ping request from h_1 will cause all switches to rebroadcast the packet to all their ports. When the ping reaches h_9, it will send back a reply, causing all controllers to learn both the source and destination ports for h_1 and h_9. At this point in time, the controllers can install flow table entries to automatically forward packets to the known ports.

We will run the test twice: Once using flow tables for forwarding, and once without. When not using flows, the controller will issue a *forward packet to port* to the switch for each packet.

5.1.1 Linear Relationship of Expected RTT

Before we present the results, let's look at what we should expect from the configuration above [27].

Using the link-latency L and node processing time P, and a constant K for background noise, we would expect the round-trip time (RTT) between h_1 and h_9 to be

$$RTT_{h_1,h_9} = 2\left(\sum_{n=1}^{4} L_n + \sum_{n=1}^{3} P_{S_n} + \sum_{n=1}^{3} P_{C_n}\right) + P_{h_1} + P_{h_9} + K \tag{5.1}$$

We can simplify by assuming that the controller processing time P_C will be negligible when complete flow entries have been installed, as the switches will then handle the forwarding themselves. During this initial ramp-up, $L_{C,S} \to 0$ as $P_C \to 0$. We will also ignore the few cycles spent on ICMP-processing, setting P_{h_1} and P_{h_9} to zero. As we will not attempt to measure K, we will simply set it to zero as well.

Using the link latencies $2\sum_n^4 L_n = 2 \cdot 4 \cdot 5$ ms $= 40$ ms and our simplifications above,

$$\begin{aligned} RTT_{h_1,h_9} &= 40 \text{ ms} + 2\sum_{n}^{3} P_{S_n} + 2\sum_{n}^{3} P_{C_n} && (5.2)\\ &= 40 \text{ ms} + 6P_S + 6P_C && (5.3) \end{aligned}$$

5.1.2 Results

Results for the two tests are shown in table 5.1 and figure 5.2.

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	Std. dev
Flows	40.70	43.10	43.90	44.85	44.60	238.00	11.05
No flows	61.40	74.80	76.10	83.57	78.30	2134.00	111.98

Table 5.1: Summary of baseline ICMP ping RTTs (ms).

There is a noticeable ramp-up as port numbers are learned and packets are rebroadcast. After some time, the RTTs seem to get more steady. Because of this short period of high RTTs—compared to the total number of samples—we

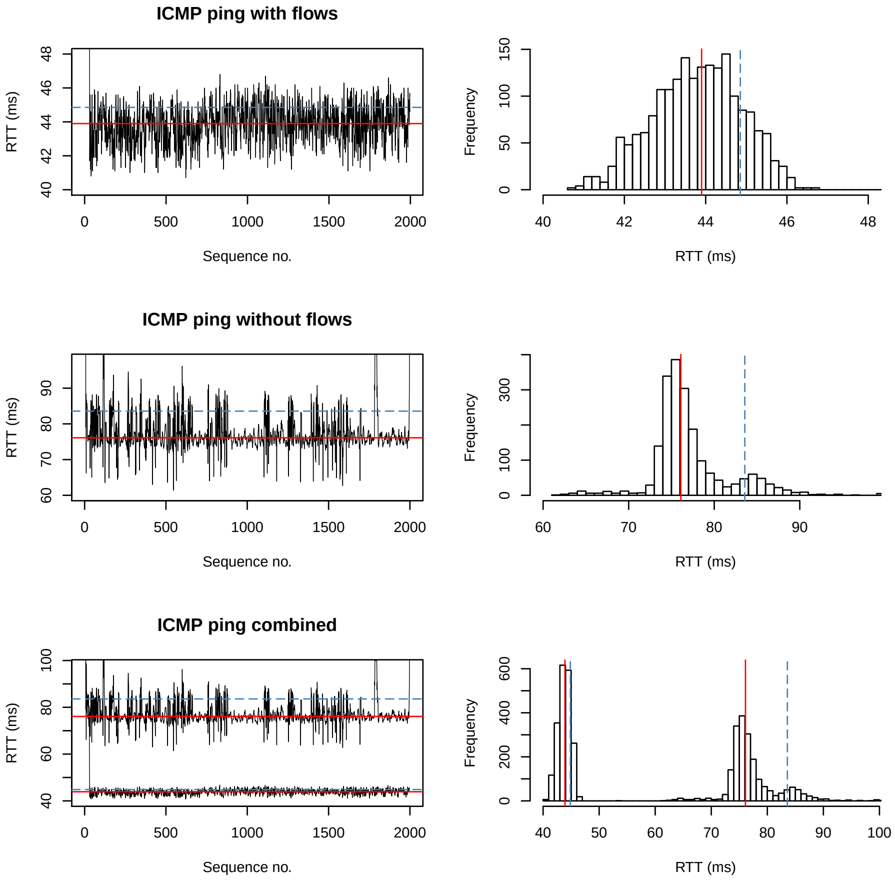


Figure 5.2: RTTs for baseline test. Medians are shown in red and the mean values in dashed blue. These plots do not show the large RTTs during ramp-up. The first row shows RTT for switches using flow table entries for packet forwarding. The second row are measurements when using the controller to forward packets. The last row plots both of them. Note that the histograms have a long tail. We only plot up to 100 ms.

will use the *median* as a more realistic value for the RTT. We also see that the RTT is consistently above the theoretical minimum of 40 ms.

Using the median RTT for the first run (with flows) in equation 5.3, we can estimate P_S.

$$
\begin{aligned}
6P_S &= RTT_{h_1,h_9} - 40 \text{ ms} - 0 \text{ ms} && (5.4)\\
&= 43.90 \text{ ms} - 40 \text{ ms} = 3.90 \text{ ms} && (5.5)\\
P_S &\approx 0.65 \text{ ms} && (5.6)
\end{aligned}
$$

In other words, the one–way latency per switch is somewhere around 0.65 ms.

The value for P_S in the first run should be very near the value for the second run, because the switch still needs to forward packets. The only difference between the runs is that, in the first, it must perform a flow table match, and in the second, it must forward the packet to the controller. We will therefore use the value of P_S in the first run to estimate P_C in the second run.

$$
\begin{aligned}
RTT_{h_1,h_9} &= 40 \text{ ms} + 6P_S + 6P_C && (5.7)\\
&= 40 \text{ ms} + 6 \cdot 0.65 \text{ ms} + 6P_C = 76.10 \text{ ms} && (5.8)\\
P_C &\approx 5.37 \text{ ms} && (5.9)
\end{aligned}
$$

Again, we would like to reiterate that we are running these tests on a *simulator*. That is why the results match so well with the expected RTT. We also made a *Q-Q plot* (see fig. 5.3) to see if the samples were normally distributed, and—indeed—they match perfectly (except for the ramp-up phase).

This is most likely because the underlying simulator uses some pseudo-random number generator (PRNG)[1] to produce simulated latencies. But our benchmarks will still be useful to us. Using the same topology and link latencies, we can easily see what kind of implementation techniques that will make the system more responsive.

5.2 RTT for Paxos-on-controller

Here, we present our results of handling Paxos entirely on the controller. For reference, we use the topology given in figure 3.4 on page 23, using events from tables 4.5 on page 43 and 4.6 on page 43.

From the host c_1, we sent UDP messages containing the client's host name, a sequence number and the current date and time to hosts h_5, h_8 and h_9, measuring the round-trip time (RTT). These messages were sent at the same time from c_1, and the client waited for replies from all three hosts before sending another one. Only a hundred messages were sent, but should give an indication of performance.

[1] Or other, implicit means.

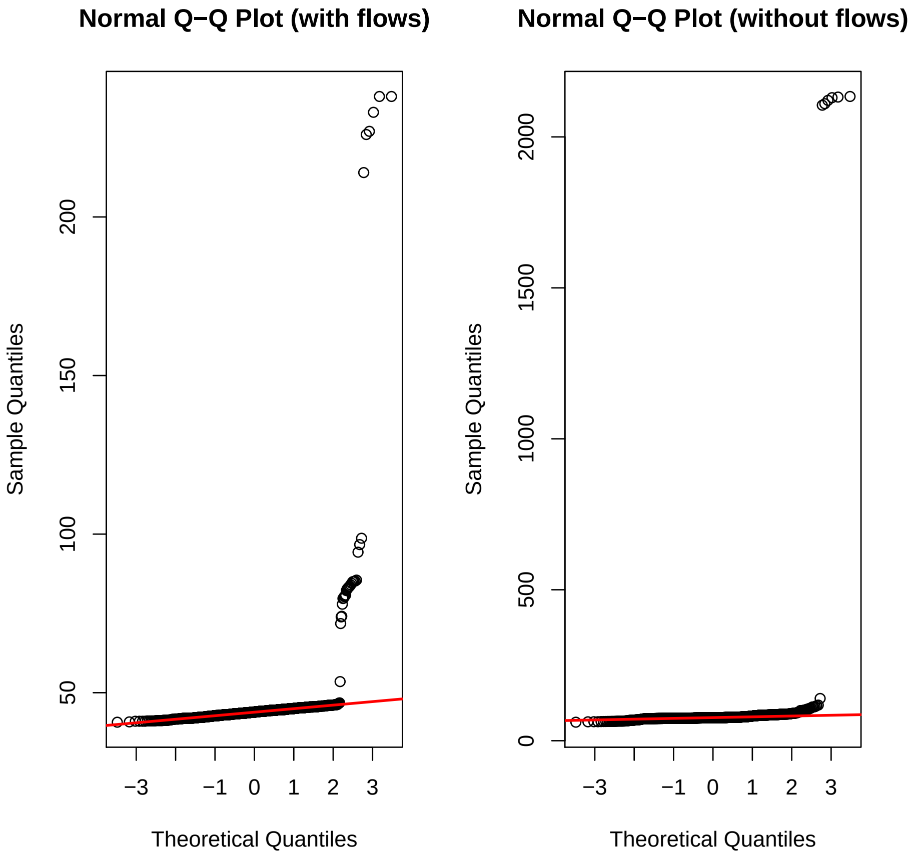


Figure 5.3: Q-Q plot for ICMP ping RTT (ms).

Paxos-ordering was performed for each UDP packet going from c_1 to the hosts h_5, h_8 and h_9, but no ordering were done for the replies, which were forwarded using the L2 Learning Switch from section 4.2 on page 36, running on the OpenFlow controller.

The results are given in table 5.2 and plotted in figure 5.4.

While the maximum values were in the same range, there were some variations in the median values. Surprisingly, h_9, which has the longest path from c_1, has the lowest mean and median round-trip time.

The host h_9 also showed consistently more stable RTTs. We have not done any further analysis on the load of various links, but there could for instance be timing issues between the Paxos messages, in the way the system has been set up, that consistently lead to better performance for $c_1 \leftrightarrow h_9$. However, this is mere speculation, and a more detailed analysis would be called for.

The interesting part is comparing the combined values against the results in table 5.1 on page 45. It shows that the medians between these results and the

	h_5 **RTT**	h_8 **RTT**	h_9 **RTT**	**Combined**
Minimum	54.21	95.35	54.21	54.21
Median	135.06	139.16	97.44	117.90
Mean	131.00	143.85	104.69	126.50
Max.	277.13	227.80	212.01	227.80

Table 5.2: Results of 100 round-trip times from c_1 in ms.

flowless one in table 5.1 differs by a factor of 1.4, or about 34 ms. As these complete packets are sent up from the switch to the controller, down again and then onwards, this is not too bad.

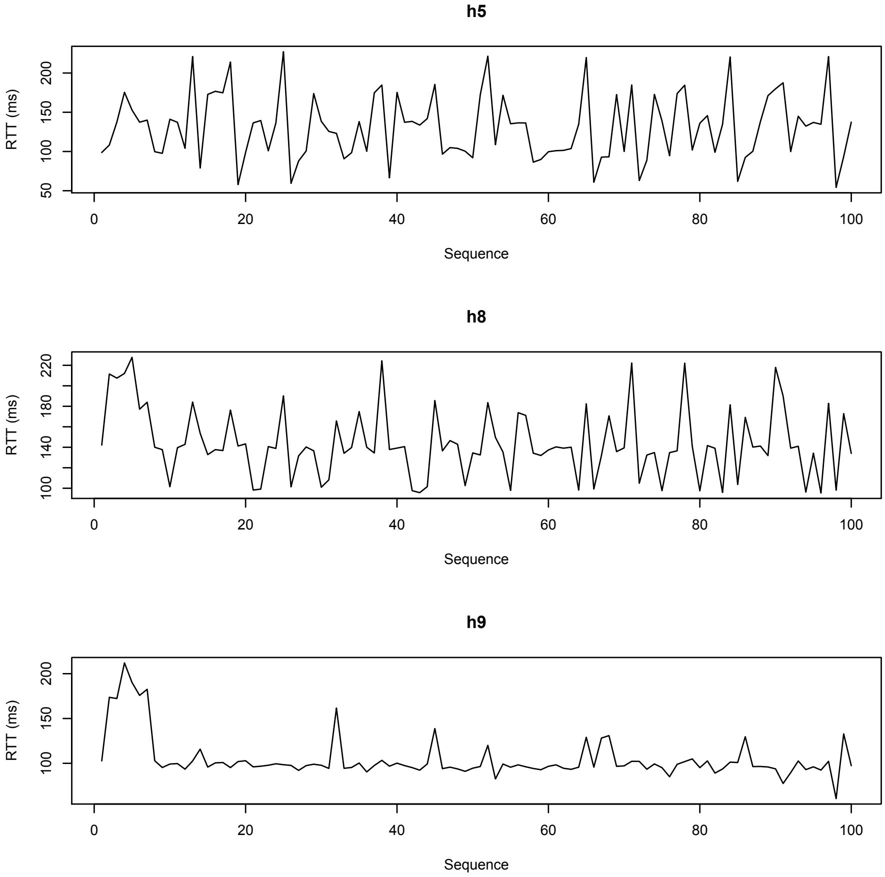


Figure 5.4: Results of 100 round-trip times from c_1.

Chapter 6

Improvements and Future Work

Here we present findings during design and implementation of switch-level Paxos that suggest further study.

6.1 Using the OpenFlow queue

WHile the Paxos leader decides the order of each packet, it is the learner that actually makes sure that they are processed in the correct sequence. In multi-Paxos, the learner has essentially a *queue* of packets to be sent. As soon as it has a contiguous sequence of packets, starting from the last processed one, it will process them.

OpenFlow has a queueing mechanism, which can be used to implement quality-of-service (QoS). It would be very interesting to investigate whether this could be used for multi-Paxos slots. We have not looked more into this.

6.2 Monitoring Link Status

OpenFlow makes it possible for controllers to receive notifications when link-status changes. In OpenFlow 1.0, this is restricted to receiving *link up* and *link down* notifications.

We could take advantage by monitoring the links to other switches, triggering, for instance, leader election when needed.

6.3 Full Paxos Support

The most obvious improvement to this project would be to implement all of Paxos: Trust, prepare and promise messages and what has been mentioned above.

We clearly stated the scope of exploration in chapter 1, restricting ourselves to only handling accept and learn messages. At the same time, it should be possible to build upon our implementation to support full Paxos. We have left several placeholders for code to do so, and adding new message types is trivial.

Chapter 7

Results and Conclusion

We set out to explore the merits of moving Paxos to the switch-level. After having finished this thesis, we see that what we set out to do was a gargantuan task.

It would be enough just to implement Paxos on the controller, as it requires a lot of supporting software components like booting the Mininet network, packing and unpacking of the Paxos IP-less messages, the L2 Learning Switch (which also fully supports flows, without any preconfiguration) for forwarding packets correctly, and finally to attempt to do all of this on the large and complex Open vSwitch software, running as a Linux kernel module.

7.1 Paxos in the Controller

In doing so, we implemented Paxos on a software controller, sending commands to a switch. Using this, we designed a distributed, replicated system with guaranteed ordering for the UDP protocol. The switches sent all packets up to the controller, who then performed Paxos ordering before delivering duplicated packets to end-hosts.

The main findings for this part of the thesis are:

- Providing Paxos ordering transparently is possible, as we have demonstrated, and we were able to provide ordering of UDP packets on a network consisting of three switches and nine end-hosts.

  Especially for UDP-based services, being able to guarantee same-sequence replication may be useful for certain types of services.

- There was some overhead in sending full packets to the controller and not using the flow tables for fast switching. This was expected, but the RTT was only around 1.4 times larger than the equivalent test using flowlwss ICMP pings (section 5.2).

  This result is not surprising, though, as packets need to perform additional link-hops to get from the switch to the controller and down again.

7.2 TCP Replication

After verifying that UDP-replication worked and that the order of the packets were correct, we attempted to move on to TCP.

We were able to duplicate TCP packets to all the hosts on the networks, but as our system sent back replies to the client without ordering, we got a race condition on the TCP packets: The host with the shortest network path to the client responded first back to the client, causing the client to send a TCP FIN packet to it. But because of replication, all hosts would receive this, and they would thus close their connections before they had time to serve the original client request.

While we were able to communicate with one client and one host, the TCP replication was unreliable for the other hosts.

The TCP protocol's design runs contrary to the way we attempted to perform replication: It has an ordering scheme of its own, and seeks to establish an end-to-end connection.

Being able to replicate TCP in this manner is most likely hard to get right, and we chose not to investigate further.

7.3 The Paxos Messages as a Network Protocol

We designed a simple, non-IP protocol for exchanging Paxos-messages, putting the parameters in the payload of Ethernet frames. By definition, this is a separate, non-IP protocol.

While the protocol was very simple, not accounting for retransmission, data corruption and the like, it was demonstrated to work reliably on the software simulator.

Furthermore, being so simple, it was very easy to reuse it when moving from the Paxos controller down to Paxos in Open vSwitch. Extracting Paxos parameters was simply a matter of looking at given offsets in the packet data.

7.4 Paxos in the Switch

We partially implemented Paxos on the switch itself by modifying the Open vSwitch source code. Due to complexities in the architecture of such an advanced, production-grade software system, we were unable to fully test it.

While we implemented all parts of the simplified, multi-Paxos algorithm, including handling of incoming client packets, accept and learn messages, we were not able to send out stored packets on the network. The sole reason was lack of time: This is complex software to understand, and sometimes its architecture prevented us from making progress.

On the other hand, we were able to easily program flows using Paxos as constituent primitives, and could combine them with existing OpenFlow actions. We believe this shows that programming not only the controller, but new actions on the *switch* as well can be very useful and even make it easier to build complex networking flows.

Bibliography

[1] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[3] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[4] Martin Casado and Nick McKeown. The Virtual Network System. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 76–80, New York, NY, USA, 2005. ACM.

[5] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[6] Christian Stigen Larsen. High OpenFlow upcall RTTs introduced between 2.0 and 2.1? `http://openvswitch.org/pipermail/discuss/2014-April/013786.html`. [Online; accessed 2014–05–02].

[7] Edward Crabbe and Vytautas Valancius. SDN at Google: Opportunities for WAN Optimization. In *IETF84, IRTF meeting*, 2012.

[8] David Erickson, Glen Gibb, Brandon Heller, David Underhill, Jad Naous, Guido Appenzeller, Guru Parulkar, Nick McKeown, Mendel Rosenblum, Monica Lam, et al. A demonstration of virtual machine mobility in an OpenFlow network, 2008.

[9] Open Networking Foundation. OpenFlow Switch Specification, Version 1.0.0 (Wire Protocol 0x01). `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`, 2009.

[10] Open Networking Foundation. OpenFlow Switch Specification, Version 1.1.0 (Wire Protocol 0x02). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf, 2011.

[11] Open Networking Foundation. OpenFlow Switch Errata, Version 1.0.1. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.1.pdf, 2012.

[12] Open Networking Foundation. OpenFlow Switch Specification, Version 1.2 (Wire Protocol 0x03). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf, 2012.

[13] Open Networking Foundation. OpenFlow Switch Specification, Version 1.3.0 (Wire Protocol 0x04). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, 2012.

[14] Open Networking Foundation. OpenFlow Switch Errata, Version 1.0.2. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.2.pdf, 2013.

[15] Open Networking Foundation. OpenFlow Switch Specification, Version 1.4.0 (Wire Protocol 0x05). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, 2013.

[16] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[17] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[18] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[19] Masayoshi Kobayashi, Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan van Reijendam, Paul Weissmann, and Nick McKeown. Maturing of OpenFlow and Software-defined Networking through deployments. *Computer Networks*, 2013.

[20] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[21] Leslie Lamport. Paxos made simple. *ACM SIGACT News 32*, 16:133–169, December 2001.

[22] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[23] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[24] James McCauley. POX. `http://www.noxrepo.org/pox/about-pox/`, 2011. [Online; accessed 2014–04–19].

[25] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[26] Hein Meling. Paxos Made Insanely Simple. 2014.

[27] Kevin Phemius and Mathieu Bouet. Monitoring latency with OpenFlow. In *CNSM*, pages 122–125. IEEE, 2013.

[28] The Mininet Authors. Mininet GitHub Repository. `https://github.com/mininet/mininet`. [Online; accessed 2014–04–28].

[29] The NOX and POX Authors. POX OpenFlow Tutorial, L2 learning switch. `https://github.com/noxrepo/pox/blob/4aa6e94051c56333e61ae3d2955aff83dfbef1ef/pox/misc/of_tutorial.py`. [Online; accessed 2014–04–30].

[30] The Open vSwitch Authors. Open vSwitch GitHub Repository. `https://github.com/openvswitch/ovs`. [Online; accessed 2014–04–28].

[31] The POX Authors. POX GitHub Repository. `https://github.com/noxrepo/pox`. [Online; accessed 2014–04–28].

[32] Robbert van Renesse. Paxos Made Moderately Complex. March 2011.

[33] Guido van Rossum and Fred L. Drake. *PYTHON 2.6 Reference Manual*. CreateSpace, Paramount, CA, 2009.

Appendix A

The Thesis VM Image

The best way to run the thesis code is to download a virtual machine image. We used VirtualBox for running this image, but it should also work on VMWare. It comes preloaded with Mininet, Open vSwitch, POX, Wireshark and more.

The user `mininet` has the password `mininet` and `sudo`-rights.

A.1 Setting up the Virtual Machine

The Linux VM image containing a ready-to-run version of the code in this thesis, along with all its tools, can be downloaded from

`http://csl.name/thesis/mininet-vm-x86_64.vmdk`

To verify that this image has not been modified after the time of thesis submission, you should download the author's GPG-key (listing A.2, p. 61) and use it to verify the file digest in (listing A.1 on the next page — A.1, p. 60).

To import the author's key, you can use the GNU Privacy Guard (GPG) or any software compatible with Pretty Good Privacy (PGP). Importing the key is done by running the command `gpg --import` and pasting a copy of the author's key, ending the input by hitting `CTRL+D`.[1]

```
$ gpg --import
# paste in author's key and hit CTRL+D
```

You should now see they key on your key-ring.

```
$ gpg --list-keys
```

The key's fingerprint should be the same as below:

```
pub   4096R/FA475DD2 2013-04-23 [expires: 2016-04-22]
      Key fingerprint = D611 0F24 4813 9908 1CFE  79BA 1AB4 2C77 FA47 5DD2
uid                  Christian Stigen Larsen (General key) <csl@csl.name>
```

[1] You can also copy the key to a file `key.asc` and importing it with `gpg --import key.asc`

```
sub   4096R/D2495ED9 2013-04-23 [expires: 2016-04-22]
```

Finally, you need copy the VM image digest (listing A.1 on this page) to a file called `mininet-vm-x86_64.vmdk.asc`, placed in the same directory as the downloaded VM image `mininet-vm-x86_64.vmdk` (from **??** on page ??). You can then run `gpg --verify mininet-vm-x86_64.vmdk.asc` to verify the digest against the author's key.[2]

```
$ gpg --verify mininet-vm-x86_64.vmdk.asc
gpg: Signature made Thu Apr 24 11:52:02 2014 CEST using RSA key ID FA475DD2
gpg: Good signature from "Christian Stigen Larsen (General key) <csl@csl.name>"
```

Listing A.1: GPG signature for the thesis VM image.

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQIcBAABAgAGBQJTdKZ6AAoJEBq0LHf6R13SAcQQAKVN25/s2VOozyTKTOw9yWMF
pF/lbUZrggFqarkX2nfKY7AlQa8qcTGt8nOdbTYKVwgbDKzZDbXh12qyAeGq33NG
VUV4QmOh9brprIcRjCYkIdIys6vM9VwzMYEEj2+2yhF44eidC3otOiQk5ikXBvWK
XQOvldpg1bAzgTYdHCqrYYiB91Q/zXs6We8EP1vmwBCiLGt4RSGxf4EtORU0c0dG
HAWU/uhKEWgayNYqHV3ueQ9n1+QbmD1jyrMwGJui80YTF419Np1yeadHzPUJNWxh
FqH2SGnyux7hzACcHbUrm0egtZnsF7RAl8uyQ3pUg1VTmdFTb+jmm/U63OKurvNK
vXjc0eEkvJRXPBK+Qf5mgAZd0vRy6LSzhWrvQmbo5/kHIwtPOMCAIplj3Ihn8lXk
lDawJoig9PNV07VPh67lAlfa9Mh9rWketUyPgwu23R+j5k899lghhUUcdiOmLhRO
ag5l/h+47/GiGtp2/zX2+xRXgK+OvDF/5aVA39p1cphgm6csVR7XwUFl4bUpzlTR
tiNaX/y7CyW+Y/cuZCKiNNqoI2AFalb9QQiLzDy7QHcogkZ8KN/sTYzDLIS9NipU
LhCRzpzsxuZF1xANG55Z9UEBprc+hSMlUetfWR/eeHhp4ForMUPqzJtCWZOM3Mf6
9MnRb6wPmi3Kr4pChNeP
=HvEl
-----END PGP SIGNATURE-----
```

[2] Note that if you haven't marked the author's key as *trusted*, you will get a warning about it. But it should say that the signature is good.

Listing A.2: The author's public GPG-key

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1

mQINBFF21LABEACrXyM2pQ9rl0TFlubOUBb9OtUF+wxCIqFCCvgVoYNCbldXMixc
wJzv6y76m/xfrpzZjl73tAnbY3WezHpY5MtPc/0taW0BL5nlx1i21tfoW4YpHF4N
jPXkLYFiglb43eChRE5xbH14iaJ335SLt4YKFMIArug6g9tBjyjkXvvZNXJOKxDr
jUFX13hIKzyF2x298j+sNwlXy8SUB1NM6NNrtmhD46QMqlxBn2/+U1gXQrDpBxJR
PlXomaiTNb9hEH2XQcTfPMnFbZTzGO2fWA/njzSRC3L5gwsbGronFbnkf0rN8RrF
amPOu171NsH5YhHbsVpGG6KepGA/i1zq9sPXXYHKC0NdOHdrGhEuOjDNWOy/hu0n
eldUclSmnL04QHIwGkZomrgyQT2GcNM0U5wyXZ4QHKjamNoT0etLpXPmXGMQhSVY
psVvchQfBzzohuiOqtb3KdbAqua09FB86AfmWOXsHnvv3A7q864QllpPhP0SwpZ/
bW9e/2POWYpaEZ1jWLkhj1ctnRDBP+2lvKu0zp8wvN4xRDVCPxHonbKhrWV0vCD7
UP+PAvqfhlgorXSeaSZvhMa4evSrYxZwYj52m3x3P5dSFDeFZFgrbJ/fNyRJRino
8dTYT/+ccEj1y+mGDPV2T0A60c+ZHmhpg1cshtOXk+c0qCfYEndmSAQT7QARAQAB
tDRDaHJpc3RpYW4gU3RpZ2VuIExhcnNlbiAoR2VuZXJhbCBrZXkpIDxjc2xAY3Ns
Lm5hbWU+iQI+BBMBAgAoBQJRdtSwAhsDBQkFo5qABgsJCAcDAgYVCAIJCgsEFgID
AQIeAQIXgAAKCRAatCx3+kdd0p8aD/9a6MBhfkanB4vZCKhNYjM89Oxo/E06zEAm
KCxy3N2TnRBkqwQFkflKVJe9nw2PsSITBbKDIVWV3jJ06Lgx+hVMyu5srvCPirSv
dF/0chS0tNL52ZIz13EJnprZZg80+CYPQoHPg0nS8xg+qV1bROFB5n4K+cyCy4Db
l36k316zZ042NwoaEMHqgLd7Lr55FpNyoHaGtGOLSmW4BQkfldx6G1kdJefFY43h
bj2YICuj6vY8ztmPVjrmtoiomMFAj+dIWW+z1TAsQ4lUhWpXEEIW5lNePBce2jVZ
1d+oWe9u//RzBRBKy/jI0GxE+Pq3ZdLOxM87tejYDcdb/QUQSGmQ4QugxvKXol5W
XT8wsfNm+c2amolnpVuWHMCHHZSY8PGLJwBc+oZOlEBrXxE8dU8uD3A3fBQgiABD
ETLEs0KzS9oo1m6EGrZSD0v3cbR4XtrEw9elUIhS6mwUpjoFqanNgwUXJiomsvjR
63FOw1wzO7TMz8weRr+ZUXMwvg7QEuxkIhGqnaAk5t1BMKNFneKs7NApwa2FyZ6q
oVZXG6INEee8Uw+SBph9jq9O2mBueiWVNei7+tHcZgqZAmNyPh1cwo37c4yHX707
hqzmdnnW54/HPDeIly4gC/wRQZUkWb5z981XiJgSLzppKdCyiX4ygJ730WdBjjYq
QW8ZING9xrkCDQRRdtSwARAA6xNB6yQIhYUkIZ1UfBq91iQIFrt6v0cxCUqCqFYG
y5+bnqOIoZqvNOW0oMPWKYNaCGow+lPFz8+alFBrrNzznoHYTQwNC68qaXxoIMH4
o4Ah+IK5KQ9g/iUu9fbCPcpFYH90z4TJM2uAeZO9eJQmQqaWqfj2Y9IFDNEgDEln
rq0BgpD1R+qArPsGT0i34wTQ+cO9apGZB8FH7bGqvqReBmifW9vHh7DVtA+og9Js
wWNJAaRCHv/AXnqfRxwCoeFQQGW4cncDDuaYvC6ADPPvQ9c6FzaFE36p45OKPzy/
Z1gAjDhs9dlkdtYxKj8uyMDBlDaC4J/rPfZ7kD2PFi0iXfEk0JyAXihaAjz3eJaH
xwnjRQVNrP4PW+3We3DI15VzgDmzjem2rVHBVigNhi5dFNbhSmTThQx2wRlOI57s
D3QDvJl+ppPHlP0yQETe4F9Op6diLm7jXHmvezII5RN29gKjgnz0jvXBKqHfXIco
SgB418Jp+XTzsuA/SuhuiOo/d0ac/RiM95pu8upYLbRo1VqJwQyTPqUzj0zWiSTl
waekJpwwkQONZwFtquYYVdLhR30X1GX92OfAd9JdbMGK7WR3RZfwNSUWKnCSbqoi
WAjHW9sEK7ltU8TKKXJjbseSPNKk9Khc8h87skmNofOknYO3nTG50YTvHSWpxxdL
feEAEQEAAYkCJQQYAQIADwUCUXbUsAIbDAUJBaOagAAKCRAatCx3+kdd0npOEACU
Kw4AR5GBjBTcrb0JQj+YFBZO4heYA/4UMcbPqvwJzeX0nSuZ+vkB9GV08nd7/jC2
+CiHlTlrtoH40S7SdOGoZgNx7WHeHFKLo8i29lUf0ID55TOs0EctYwX/MWLWt8JJ
uZ8OXQnrXL9Mtg1yybkftMiqSmAYtWfcX6Wv86zpPzM8K4kWkd5o3NNFWBareEqL
fmdjNyyumpYX+5tHMc2v7oxG/oEC5SCVJmF5ZzFuiBBvJPPIxfYHajoR1Xz+kKf/
RDtbTJxnR0Ftfhwb5Tv4iX6rYXHwvC6bhKB4Knndba+WeN7tYKAX7rLEasdzubGj
vSZgJD3CZykk7WIq5exDHTw5E4BtgOZc5LxOf2KsWBxks6vdjbeAICA135tKxtDB
lPMytrCMNy6JNoA0fFB/KxSnOhOIxH4Ar/vXaBJxI0mQuhs9Qt+27alf6R6OgjIN
S7KQ5l3MaUFUaiRwvGJeeolT+e6X/ssLZQyWDkrNxkonZ+GhbSO5p5zgGXA3PXGg
X/18vQyTCn1jt+jzr+f/6BW+E9pqusJ4MYdDM5ThKv7TjyslMW71cdFjCHNkf1/k
9zduikQwVddgU4Ha3T6+jOY0VLncguA7UnMTOdFGdL4SchZjswDzOHPc0xC4+Un5
x7JoKcugqMAINWCfPKu5IXU6SqSkKA6ddVG/SJ5wcA==
=iAwm
-----END PGP PUBLIC KEY BLOCK-----
```

A.1.1 Settings for VirtualBox

The author's settings in *VirtualBox* for the Linux VM are given in table A.1. The fields marked as *needed* must be set as shown, otherwise the VM may not work properly.

Start VirtualBox and create a new VM. Then point to the provided VM image (the option *Use an existing virtual hard drive file*) and copy the settings in table A.1.

When you boot the VM, you should try to ping a remote host on the internet, then you should attempt to `ssh` into it from a terminal on the host computer.

Needed	Field	Value
	Name	mininet
*	Operating system	Ubuntu (64 bit)
	Base memory	1024 MB
*	Boot order	Hard disk
	Acceleration	VT-x/AMD-V, Nested Paging
	Display Video memory	16 Mb
	IDE Secondary Master	vboxguestadditions.iso
		CD/DVD
*	SATA Port 0	`mininet-vm-x86_64.vmdk`
*		Normal, 8,00 GB
*	Network Adapter 1	Intel PRO/1000 MT Desktop
*		NAT
*		MAC: `FEEDFACEBEEF`
*	Network Adapter 2	Intel PRO/1000 MT Desktop
*		Host-only Adapter, 'vboxnet0'
*		MAC: `0800270A8160`

Table A.1: Author's settings for the VM image.

It is important that you set up the network *exactly* as shown, otherwise it may not function correctly.[3]

The `vboxguestadditions.iso` is not needed. We have used it only to enable sharing of folders between the VM and host computer.

Remember that you can log in using the user `mininet` with the password `mininet`. This user should be able to get a root shell by typing `sudo bash`.

A.1.2 Network Settings

To be able to use NAT on your VM, you need to set it up on your guest OS networking settings in VirtualBox.

[3] If it still does not work, make sure you have set up the guest OS networking settings correctly (ch. A.1.2). You may also want to edit the file `/etc/udev/rules.d/70-persistent-net-rules`. Update the corresponding MAC address and comment out all other lines, then reboot the VM. You may also need to change the VM's IP-address in ch. A.1.3 on the next page. If unsure of the IP-address, type `ifconfig eth1 | grep inet` to see the VM's address.

In the VirtualBox manager, go to preferences, network, *NAT Networks* and add a NAT-network called *NatNetworking*. Use the settings from table A.2.

You also need to add an entry under the tab *Host-only Networks* using the settings in table A.3.

Field	**Value**
Enable network	Yes
Network name	NatNetwork
Network CIDR	10.0.2.0/24
Supports DHCP	Yes

Table A.2: Settings for guest OS NAT networking.

Field	**Value**
Adapter	
Name	vboxnet0
IPv4 address	192.168.56.1
IPv4 network mask	255.255.255.0
DHCP server	
Enable server	Yes
Server address	192.168.56.100
Server mask	255.255.255.0
Lower address bound	192.168.56.101
Upper address bound	192.168.56.254

Table A.3: Settings for guest OS Host-only Networks.

A.1.3 SSH Settings

In order to work with the VM, you need to add the following options to your local `~/.ssh/config`

```
Host mininet
  Hostname 192.168.56.102
  User mininet
  ForwardX11 yes
  ForwardAgent yes
  RequestTTY yes
```

X11-forwarding is required in case you want to start xterms on Mininet nodes or run Wireshark. Note that it also forwards your ssh-agent—you may not strictly need this. The `RequestTTY`-option is **very important**, because it lets us start terminal programs on the remote host. Without it, some of the examples here will leave processes running in the background on the remote host when you SIGINT them (CTRL+D).[4]

[4] If you don't use this option, you can manually type `ssh -t` to request TTYs correctly.

You may need to change the `Hostname`-parameter for your particular system. Doing this correctly saves you time when the host computer changes networks.

To be able to log on password-less, you need to upload your public key:

```
$ cat ~/.ssh/id_rsa.pub | ssh mininet "cat - >> ~/.ssh/authorized_keys"
```

Boot the VM and make sure you can log on to it without typing a password:

```
$ ssh mininet
```

A.2 Compiling

Here follows instructions on how to compile all the code needed for running Open vSwitch, Mininet, POX and the thesis code.

Note that the VM image should already have compiled and installed the latest versions. This section is only included for the sake of completeness.

A.2.1 Software Versions

Software	**Version**
Mininet	2.1.0+
Open vSwitch	2.1.2
POX	0.3.0 (dart)

Table A.4: Software versions used in the thesis.

The modifications we made to Open vSwitch were originally based on 2.0, because there were big differences in OpenFlow upcall latencies between 2.0 and 2.1. However, we moved these changes over to 2.1.2, which was released after a bug was discovered based on findings by the author [6].

A.3 Thesis Code

The thesis code consists of Mininet topologies and POX-controllers, both written in Python, and modifications to Open vSwitch.

The Python code does not need compilation, and symlinks and paths have already been set up correctly, provided that you use the Makefile in the home directory of the mininet user.

A.3.1 POX Paxos Controller

On the Linux VM, under `/home/mininet/bach/paxos/` is the entire Paxos implementation on the controller, including an L2 Learning Switch that installs flows.

The most important files are:

- /home/mininet/bach/paxos/controller/paxosctrl.py contains the complete Paxos-on-controller implementation.
- /home/mininet/bach/paxos/controller/baseline.py contains a highly performant L2 learning switch that can operate both with and without flows, able to learn MAC addresses, IP addresses and their port numbers. If instructed to install flows, it will do so dynamically, being able to learn about new nodes joining the network.
- /home/mininet/bach/paxos/topology.py contains the various Mininet network topologies used in the thesis.
- /home/mininet/bach/message.py contains the implementation of the Paxos message format, including packer and unpackers for such messages.
- /home/mininet/bach/tools contains various testing tools used for verifying ordering, message senders and listeners.

A.3.2 Open vSwitch

Our modifications of Open vSwitch can be found on the Linux virtual machine in the /home/mininet/ovs directory. It uses git as a repository, and a complete log of all changes can be seen using git log.

The most important files are given below. In general, code has been integrated into existing Open vSwitch code. Searching case-insensitively for "paxos", one will find the various pieces of code.

- lib/ofp-paxos.h and lib/ofp-paxos.c contain the multi-Paxos implementation and various Paxos utility functions.
- lib/odp-execute.c contains the actual implementations of onclient, onaccept and onlearn handling.
- ofproto/ofproto-dpif-xlate.c contains the preparations for setting up a Paxos datapath action when receiving a packet.
- lib/ofp-parse.c contains the parsing of command line arguments involving Paxos.
- include/openflow/openflow-1.0 contains the Paxos extensions to OpenFlow.

To build the Open vSwitch code on the VM, do the following (or use the command rebuild-ovs).

```
# Remove the preinstalled ovs
$ sudo apt-get remove \
    openvswitch-common openvswitch-datapath-dkms
    openvswitch-controller openvswitch-pki openvswitch-switch

$ cd ~/ovs
$ ./boot.sh
$ ./configure --prefix=/usr \
              --with-linux=/lib/modules/$(uname -r)/build

# To compile ovs
$ make
$ make test # optional; takes some time

# To install
$ sudo make install
$ sudo make modules_install
$ sudo rmmod openvswitch
$ sudo depmod -a

# Now restart Open vSwitch
$ sudo /etc/init.d/openvswitch-controller stop
$ sudo /etc/init.d/openvswitch-switch stop

# Disable start of controller on boot
$ sudo update-rc.d openvswitch-controller disable

# And start again
$ sudo /etc/init.d/openvswitch-switch start

# Check that it's running
$ ps auxwww | grep openvswitch

# The executable ovs-controller changed name to test-controller
# in a recent ovs version, and Mininet relies on it:
$ sudo cp tests/test-controller /usr/bin/ovs-controller

# Check the version
$ sudo ovs-vsctl show
cd702e96-d4af-4803-9e2f-ecc2f7abcd6a
    ovs_version: "2.1.0"

$ modinfo openvswitch
filename: /lib/modules/3.8.0-35-generic/kernel/net/openvswitch/openvswitch.ko
license:        GPL
description:    Open vSwitch switching datapath
srcversion:     15C32AD9E04F379CAC3D68E
depends:
intree:         Y
vermagic:       3.8.0-35-generic SMP mod_unload modversions
```

The Open vSwitch-directory is a git-repository, so you can switch branches, fetch updates and so on at will. If you switch branches, you need to do a full rebuild. The script `~mininet/rebuild.sh` will do this for you, but beware that it runs `git clean -fdx` on the directory, which removes all non-tracked files.

A.3.3 POX

POX is a pure Python-implementation and thus does not need any compilation. The directory `~mininet/pox` is a *git-repository*, so you can update it at will.

A.3.4 Mininet

Mininet is mostly written in Python but has two C-files. You can update with *git* here as well, but remember to run `sudo make install` afterwards.

A.4 Running the Code

After setting up the VM image correctly (chapters A and A.1.3), you may want to restart your Mininet before running benchmarks. This is to make sure that no processes from previous runs are hanging in the background.[5] The test code mostly takes care of this, but to be on the safe side, reboot with

```
$ ssh mininet sudo shutdown -r now
```

A.4.1 Baseline benchmarks

To run this benchmark, run the following on your local computer

```
$ ssh mininet make bench-baseline
$ ssh mininet make bench-baseline-noflows
```

If you prefer to run Mininet and the controller in separate terminals, you can start each one independently:

```
# Terminal 1 (start first)
$ ssh mininet make bench-baseline-pox

# Terminal 2 (start after POX is up)
$ ssh mininet make bench-baseline-mininet
```

After the test runs complete, there are two result files that you can download locally:

```
$ scp mininet:~/pings.txt mininet:~/pings-noflows.txt .
```

A.4.2 Running Paxos

Log on to the VM using two different terminals. Start Mininet in one window with `make paxos-net` and the controllers in another with `make paxos-pox-noflows`. This starts the controllers in a mode where they will not add flows.

[5] There may even be hanging processes from the time the VM image was uploaded.

You should have X11 set up as well (see chapter A.1.3). Open up two X11 terminals on

The network should initialize and the Paxos controllers should announce themselves to each other. You can now run Mininet-commands such as: `nodes` to see all nodes on the network, `net` to see their links and `pingall` to have all nodes ping each other.[6]

To run a command on a node, just type the node's name along with an ordinary shell command. For instance,

```
paxos/mininet> h1 ping c9
```

To test things, we can start a web-server on `h9`. The best way to do this is in `h9`'s X11-terminal:

```
root@mininet-vm:~# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.0.1 - - [06/May/2014 19:09:48] "GET / HTTP/1.1" 200 -
10.0.0.1 - - [06/May/2014 19:09:55] "GET / HTTP/1.1" 200 -
10.0.0.1 - - [06/May/2014 19:09:58] "GET / HTTP/1.1" 200 -
```

This small web-server will list all files in its current directory, and make them available for download. On `c1`'s X11-terminal, you can use `curl` to fetch webpages from `h9`.[7]

```
root@mininet-vm:~# curl http://10.0.0.12:8000
```

You should get some HTML-output. In the controller console, notice all the log messages. What is happening is that each individual TCP-packet will be ordered by the Paxos-system.

To get a rough indication of round-trip time (RTT), you can run the `time`-command:

```
$ /usr/bin/time -p curl http://10.0.0.12:8000
```

A.4.3 Monitoring Network Traffic

If you want to monitor network traffic, you can use the `tcpdump` command. You specify an interface to listen to with the `-i` option (e.g., *S1-eth1*; `ifconfig` gives a full list) and you can optionally give packet filtering rules using the Berkeley packet filter (BPF)-syntax [23]. For instance,

```
$ sudo tcpdump -nNeXS -s64 -iany \
    "(port not 22) and (port not 6633) and (dst 10.0.0.1)"
```

specifies in BPF to capture packets bound for `10.0.0.1`, except if the source or destination port is 22 (*ssh*) or 6633 (the default controller port). The option `-iany` instructs `tcpdump` to capture on all interfaces. The remaining options are

[6] The Mininet boot-script should do this automatically, as it will make sure that the controllers learn which ports different Ethernet addresses can be reached on.

[7] You can also do this from Mininet's command-line prompt by typing `c1 curl http://10.0.0.12:8000`.

explained in the manual for `tcpdump`.[8] Example output of the above command is given below.

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 64 bytes
20:34:00.063947   P 0e:df:8b:76:a5:1a ethertype IPv4 (0x0800),
  length 100: 10.0.0.9 > 10.0.0.1: ICMP echo reply, id 15566, seq 868, length 64
  0x0000:  4500 0054 54e4 0000 4001 11bc 0a00 0009  E..TT...@.......
  0x0010:  0a00 0001 0000 0390 3cce 0364 9893 6253  ........<..d..bS
  0x0020:  0000 0000 0284 0000 0000 0000 1011 1213  ................
20:34:00.070384 Out 0e:df:8b:76:a5:1a ethertype IPv4 (0x0800),
  length 100: 10.0.0.9 > 10.0.0.1: ICMP echo reply, id 15566, seq 868, length 64
  0x0000:  4500 0054 54e4 0000 4001 11bc 0a00 0009  E..TT...@.......
  0x0010:  0a00 0001 0000 0390 3cce 0364 9893 6253  ........<..d..bS
  0x0020:  0000 0000 0284 0000 0000 0000 1011 1213  ................
```

Our boot-scripts for Mininet usually start of with the `pingall`-command, which makes every node ping all other nodes. If you want to capture them, you need to start up Mininet first, then tcpdump and then the controller last.

Mininet will create network interfaces for each link in its topology. So, after starting Mininet, you can see the interfaces for S_1 by typing

```
$ ifconfig | grep S1
S1        Link encap:Ethernet  HWaddr 3a:6a:80:cd:b1:43
S1-eth1   Link encap:Ethernet  HWaddr 7e:56:63:c8:4a:f8
S1-eth2   Link encap:Ethernet  HWaddr 96:66:dd:b5:1c:c7
S1-eth3   Link encap:Ethernet  HWaddr 8e:9f:d0:8c:10:71
S1-eth4   Link encap:Ethernet  HWaddr 86:30:25:58:12:7e
```

The interfaces starting with `S1-eth`... are the port interfaces. To monitor one of them, simply type

```
$ sudo tcpdump -nev -iS1-eth1
tcpdump: WARNING: S1-eth1: no IPv4 address assigned
tcpdump: listening on S1-eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
11:50:53.967333 06:b6:0d:37:d0:cd > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42:
    Ethernet (len 6), IPv4 (len 4), Request who-has 10.0.0.9 tell 10.0.0.1, length 28
[...]
11:51:14.636681 9e:df:10:0b:a7:da > 06:b6:0d:37:d0:cd, ethertype IPv4 (0x0800), length 98:
    (tos 0x0, ttl 64, id 32782, offset 0, flags [DF], proto ICMP (1), length 84)
    10.0.0.9 > 10.0.0.1: ICMP echo request, id 13921, seq 1, length 64
11:51:14.641836 06:b6:0d:37:d0:cd > 9e:df:10:0b:a7:da, ethertype IPv4 (0x0800), length 98:
    (tos 0x0, ttl 64, id 36379, offset 0, flags [none], proto ICMP (1), length 84)
    10.0.0.1 > 10.0.0.9: ICMP echo reply, id 13921, seq 1, length 64
```

[8] `man 1 tcpdump`

Index

Colophon

This thesis was typeset using LaTeX from the TeX *Live*-distribution on a Mac OS X along with several packages.

The figures were made using *TikZ*. Statistical calculations and plotting were made using *R*.