Report: Self–Adaptive Routing

Research Project 2
University of Amsterdam
Master of Science in System and Network Engineering

Class of 2009-2010

Arthur van Kleef (arthur.vankleef at os3.nl)
Marvin Rambhadjan (marvin.rambhadjan at os3.nl)

Supervisor: Rudolf Strijkers (strijkers at uva.nl)

August 13, 2010

Abstract

The complexity of network administration can be simplified by implementing control programs that automatically configure network services. To implement this self-adaptive behaviour we implemented a feedback control loop in the control plane of a network. An architecture that implements a centralized control program is preferred over a distributed control program, as is the case in most current state of the art network protocols. The OpenFlow Switching Consortium defines an architecture that uses a centralized control program and also allows network operators to configure the control plane to their wishes. The programmability of the control plane makes OpenFlow well suited for the implementation of self-adaptive behaviour. We implemented a test bed based on OpenFlow to develop control programs and show how self-adaptive control programs can be written in this environment.

Contents

1 Introduction

Changes in networks such as link failures, congestion, quality of service, but also company policies require network operators to fine tune the protocols running on their networks to maintain a desired level of service. Different protocols exist and each offers different parameters to control the operation of network devices. For example, configuring link weigts can influence route calculation in routing protocols, some protocols also allow the configuration static paths throughout the network. Most state of the art protocols can be configured via the control plane that resides inside network devices and is limited to the protocols implemented by the vendor.

ForCES[1] proposes a framework that separates forwarding- and control plane in network elements and describes a standard protocol for communication between the two planes. *OpenFlow*[2] is a similar effort that is already available for use. OpenFlow uses a centralized controller[3] for control plane funcionality. The API offered by the controller allows operators to program the control plane using *C* or *Pyhton*. The programmability of networks makes OpenFlow an ideal candidate for implementing self-adaptation in networks.

1.1 Research Question

In this research project we want to investigate the possibility of creating self-adapting networks. Self-adaptive networks monitor the health of the network by implementing a *feedback loop* and when necessary apply configuration adjustments to network devices. Section 3 investigates the architecture required and which protocols are available to support a feedback control loop. Therefore our research questions are:

- What is the architecture of a network that supports a self-adapting software control plane?
- If the control plane becomes a programmable piece of software, what is the general pattern of the programs that implement routing and network management?

After identifying the architecture and protocols a test environment will be presented, which makes it possible to easily create experimentation networks that we will use to implement self-adapting behaviour. The software we used in this project will be made available via the OS3 website at [1]

[1] `https://www.os3.nl/2009-2010/students/arthur_van_kleef/openflow`

2 State of the Art in Network Control Protocols

This section describes the current state of the art in networking protocols. A disadvantage of these protocols is the use of a distributed algorithm to calculate forwarding paths, every network element makes its own forwarding decision.

In order to reach self–adaptive behavior, the input parameters of the algorithm need to be adapted such that the algorithm makes the desired forwarding decision. This comes down to calculate what input parameters need to be configured to get the appropriate routing behaviour.

The most common datacenter, campusnetworks and Internet protocols are drawn in figure 1 below and the protocols are mapped on the appropriate layer(s) of this model. Every protocol will be discussed, compared and investigated on the usage for self-adaptive behaviour.

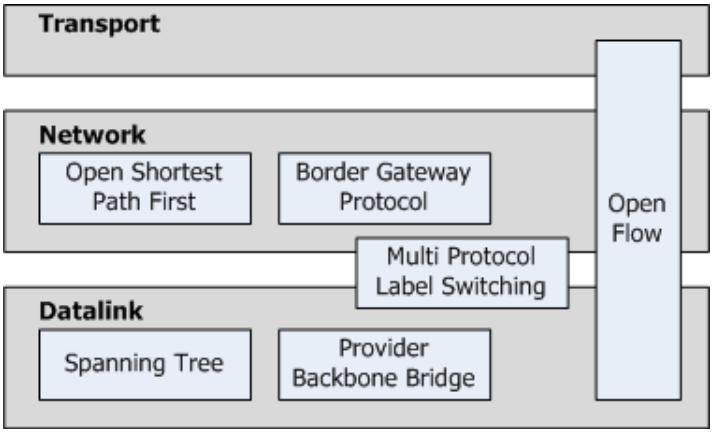

Figure 1: TCP/IP Model

The lowest layer in figure: 1 is the *Data Link Layer*, frames are delivered according to hardware addresses (MAC addresses). It is not possible to determine if two different hosts are on the same Data Link Layer segment based on their MAC addresses. A problem that needs to be addressed on Data Link Layer segments are *bridging loops*. Bridging loops occur in topologies that have multiple paths between two bridge devices. Since Data Link Layer frames have no notion of *time to live (ttl)*, frames continue to loop forever. Routing occurs on the *Network Layer (TCP/IP)* and uses hierarchical addressing, which allows routers to use a single address that represents a group of addresses when forwarding packets. On this layer logical addresses are used to address the network elements. The *Transport Layer (TCP/IP)* is responsible for delivering data to the appropriate application on the node. Port numbers are used to distinguish the incoming traffic for the different processes on the node.

Spanning Tree Protocol

The solution to the flooding problem is to create a *Spanning Tree*. A Spanning Tree is a subgraph of a graph which is a tree, connecting all vertices together. In layer two networks the *Spanning Tree Protocol* [4] (IEEE 802.1D-2004) is used to create loop free topologies.

The Spanning Tree Protocol (STP) calculates a spanning tree from a fixed root bridge. The goal of this protocol is to create a loop free topology. This is done by *blocking* paths that create loops. The 802.1d STP and 802.1w RSTP protocol create a single spanning tree for the complete topology. As a result of constructing a spanning tree, links on blocked ports are only used as backup, in case a link that is in the spanning tree fails. If no problems occur, these links are never used.

Improvements over STP resulted in the *Per-VLAN Spanning Tree (PVST)* and *Multiple Spanning Tree Protocol (MSTP)* protocols that can divide the network in different spanning trees per VLAN, or per group of VLAN's. With this method, backup paths can be used for load balancing and are not unused. During spanning tree calculation, paths are fixed for that specific spanning tree.

STP operation can be influenced by configuring the following parameters:

Parameter	**Usage**	**Description**
Root Bridge	Lowest Brigde Identifier	Define the starting point of the spanning tree
Path Cost	Lowest Path Cost (to Root Bridge)	Cost of the path
Port Priority	Lowest Port Priority	Preferred switch port

Table 1: STP parameters

Many STP variants are limited to a single broadcast domain. The STP instance can only be used within a single IP subnet. Load balancing within a broadcast domain is not possible.

Provider Backbone Brige - Traffic Engineering

Provider Backbone Bridge Traffic Engineering [5] (IEEE 802.1Qay-2009) is a technology that is based on MAC in MAC encapsulation and layered VLANs. In PBB the flooding mechanism, dynamic creation of forwarding tables and spanning tree protocols are eliminated.

Multi Protocol Label Switching (- Traffic Engineering)

The first function of MPLS is the *Forwarding Equivalence Classes* (FECs), which is responsible partitioning the entire set of possible packets to the FECs. Packets can be mapped based on network layer header, but also on for example incoming port number. The second function is for the mapping between the FECs and the next hop.

Packets that are mapped to FECs are indistinguishable from each other in such a way that all the packets in the FECs will use the same forwarding path. The assignment of packets to FECs is done when a packet enters the network, before it is forwarded. Based on the FECs, MPLS will label the packet. Further analysis of the network layer header is unnecessary, although it is possible to analyze the header to determine the "precedence" or "class of service". In this case, the label is a combination of the FEC and the quality of service options of the packet.

The MPLS label is based on the network layer header and can also be associated with the incoming port. The restriction of MPLS is that the forwarding decision is based on the appropriate FEC, which means the network layer header.

The label assignment is based on the following criteria:

- Destination Unicast Routing
- Traffic Engineering
- Multicast
- Virtual Private Network
- Quality of Service

MPLS also contains methods for Traffic Engineering (MPLS TE) to do constraint-based routing. Path calculation is done using the *Constraint-based, Shortest Path First* (CSPF) algorithm. This algorithm is basically a shortest path algorithm with the ability to enforce specific guaranties over the path. An example is that the links must have at least 50 bandwidth units. All paths with lower bandwidth units will not be taken into account in calculating shortest paths.

Open Shortest Path First

OSPF[6] uses links cost as input for the shortest path algorithm. Routes can be manipulated by modifying the weights of links. Lower costs make a path more interesting to route traffic. The shortest path algorithm is uses the cost of a path and does not consider any other configuration parameters to handle path selection.

OSPF is a link state protocol, which means that every router has a complete overview of the network. Limitations with this method is that when the greater the network is, the harder it is to synchronize all the topology information about the network. This problem is circumvented by introducing areas, where the topology inside an area is known to every router. Routers running the area border router role are then used to connect different area's to each other.

2.1 Conclusion

As shown in the previous paragraphs there are many protocols available that can be used to adapt network behaviour. Whenever new functionality is required in network equipment, a new protocol (or an extension) needs to be implemented

at the control plane of these devices. Instead of implementing new functionality on top of existing protocols we want a fully programmable control plane, which is not possible with the discussed protocols.

Our observation is that management of the control plane is becoming more complex by the introduction of new protocols. To reduce this complexity a centralized programmable control plane enables better management and adaptation to new network service requirements.

3 Self-Adaptive Networks

Networks can be considered self-adapting when they automatically adjust to changes inside the network in order to maintain a desired level of service. Feedback control can be used to achieve this [7]. Figure 2 shows an example of a *closed* feedback control loop.

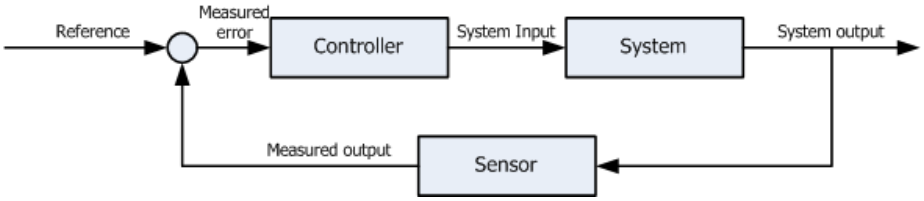

Figure 2: Feedback Control Loop

The *controller* decides when configuration adjustments need to be made in the system. For this it uses *reference* input, describing the desired level of service, and *measured output* from the *sensors* that describe how the network is performing. The controller compares this data and determines if adjustments are necessary. Adjustments are applied to the *system* and the result of adjustments is again obtained via *measured output*.

3.1 Control plane centralization

How a feedback control loop can be integrated into a network is depending on the network architecture. State of the art network equipment consists of two major components, a *control plane* and a *forwarding plane*. The control plane is responsible for maintaining a view of the network and takes forwarding decisions. Because of the complexity involved in these tasks the control plane is implemented in software. The forwarding plane is used for high speed forwarding. It uses a lookup table in memory that is filled by the control plane to find out through which ports to forward packets. This task is not very complex and is implemented in hardware, which makes high speed forwarding possible. Figure 3 gives an example of an architecture that uses control planes distributed across each network element.

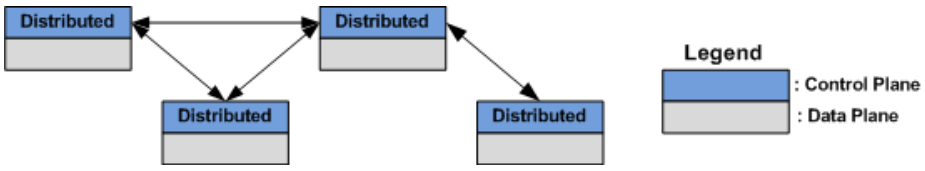

Figure 3: Distributed Control Plane

A distributed control plane througout the network makes it difficult to implement a feedback control loop, especially in larger networks. To implement a feedback control loop in a network that has a distributed control plane, means that each part of the feedback control loop needs to be distributed as well. Each

control plane is required to obtain the measurements from sensors on other systems in the network. Also this information needs to be synchronized at all control planes. To overcome the aforementioned issues the control plane needs to be centralized.

A centralized approach to implement the control plane of a network offers several advantages:

- Easy to maintain a view of the network
- Measurements are collected at one point, this eliminates the need for a mechanism that floods measurement data throughout the network.
- Out-of-band connections to the control plane allow faster dissemination of control input.

3.2 OpenFlow

A new open standard that supports the centralization of control plane in a network is OpenFlow[2]. OpenFlow is an initiative from Stanford University to open up networking devices by moving the control plane to a centralized controller. The controller takes care of high-level forwarding decisions. The controller offers network operators an interface to program their control plane.

Programmability of the network is not a new idea, *GSMP*[8], *ForCES*[1] and *IEEE 1520* are all efforts to introduce some level of programmability in the network. OpenFlow distinguishes by being more practical. It makes use of existing hardware in switches and it is already implemented by large vendors like HP, NEC, Juniper, Cisco, Toroki and Pronto in their networking devices.

An OpenFlow enabled device communicates with a centralized controller that offers the control plane function. On the OpenFlow switch a *flow table* is used to lookup how to forward packets. The flow table stores *flow entries* that are used to match incoming packets. Table 2 shows the parts that construct a flow entry.

Header	Counter	Actions

Table 2: OpenFlow flow entry

The *header* field is used to match incoming packates against. Table 3 lists the available fields inside the flow entry's header. An important observation is the available bits in the *IP Source/Destination* fields are limited to 32. In our experiments we tested IPv6 with OpenFlow. Flow entries that are created for IPv6 traffic have their fields *IP src* and *IP dst* set to empty. However, the correct *Ether type* value is set and can be used to identify IPv6 traffic. The number of bits available to the *Ingress port* field is implementation dependent.

The *counter* field is used to gather statistics and are maintained per-table, per-flow, per-port and per queue. Table 7 lists all available counters. The *actions* field describes the actions a switch has to perform when it matches a packet against a flow entry. Each flow entry has to contain zero or more actions

Ingress Port	Ether src	Ether dst	Ether type	VLAN id	VLAN prio	IP src	IP dst	IP proto	IP ToS	TCP UDP src	TCP UDP dst
X	48	48	16	12	3	32	32	8	6	16	16

Table 3: OpenFlow flow entry header fields

and are stored in the form of a list. If there are zero actions in a flow entry, the packet will be dropped. The OpenFlow 1.0.0 specification distinguishes between *OpenFlow-only switches* and *OpenFlow-enabled switches*, where the former only support actions marked *Required* and the latter also supports actions marked *Optional*. Table 4 describes the available actions.

Action	**Description**	**Required/Optional**
Forward	Forward packets to specified physical interface	Required
Enqueue	Forward through a queue, providing QoS	Optional
Drop	Do not forward packets	Required
Modify-Field	Rewrite addresses inside packets	Optional

Table 4: Actions

Besides forwarding packets to local ports, OpenFlow also supports forwarding packets to *virtual ports*. Table 5 lists the virtual ports used by OpenFlow:

Virtual Port	**Description**	**Required/Optional**
ALL	Send out all ports, except incoming port	Required
CONTROLLER	Encapsulate packet and send to controller	Required
LOCAL	Send packet to local networking stack	Required
TABLE	Perform actions in flow table	Required
IN_PORT	Send packet out the input port	Required
NORMAL	Use traditional forwarding path	Optional
FLOOD	Flood packet out all ports	Optional

Table 5: virtualports

OpenFlow communicates with a controller when it cannot find a matching flow entry for a packet. NOX[3] is an OpenFlow controller that provides a platform for writing network control programs using *C++* and *Python*. Control programs for NOX use *events*, which occur whenever NOX receives a message from an OpenFlow switch. When there is an event on the network (i.e. a link goes up or down) this will generate an event at the NOX controller. Control programs that have registered to events handlers are then triggered and executed. Table 6 lists common event types that can be used. In C we performed an analysis of the OpenFlow messages that are sent between an OpenFlow switch and NOX controller. Figure 4 shows the operation of an OpenFlow network using an OpenFlow controller.

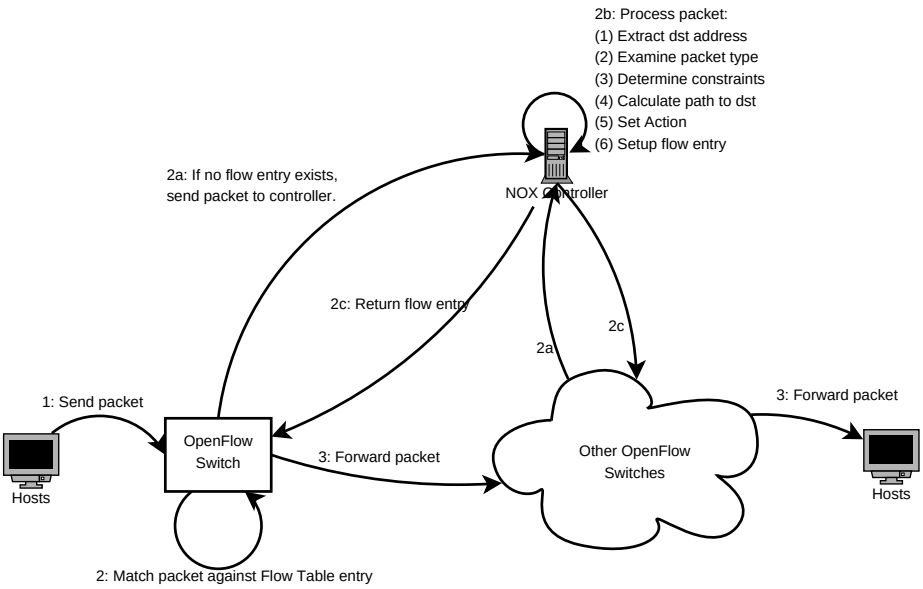

Figure 4: Operation of an OpenFlow network.

Switch Join
Switch Leave
Packet Received
Port Up
Port Down
Switch Statistics Received

Table 6: NOX Event Types

3.3 Writing a Network Control Program

After identifying the required protocols and software 3.2 this section focusses on the development of a control program that performs forwarding and network management and implements self-adaptivity. The following assumption simplifies the writing of a control program:

- The control program has a complete view of the network.
  - The topology of the OpenFlow network is statically configured in the control program. It is stored in a Python dictionary that represents a graph.
  - End-hosts location is stored in a host-to-switch association.

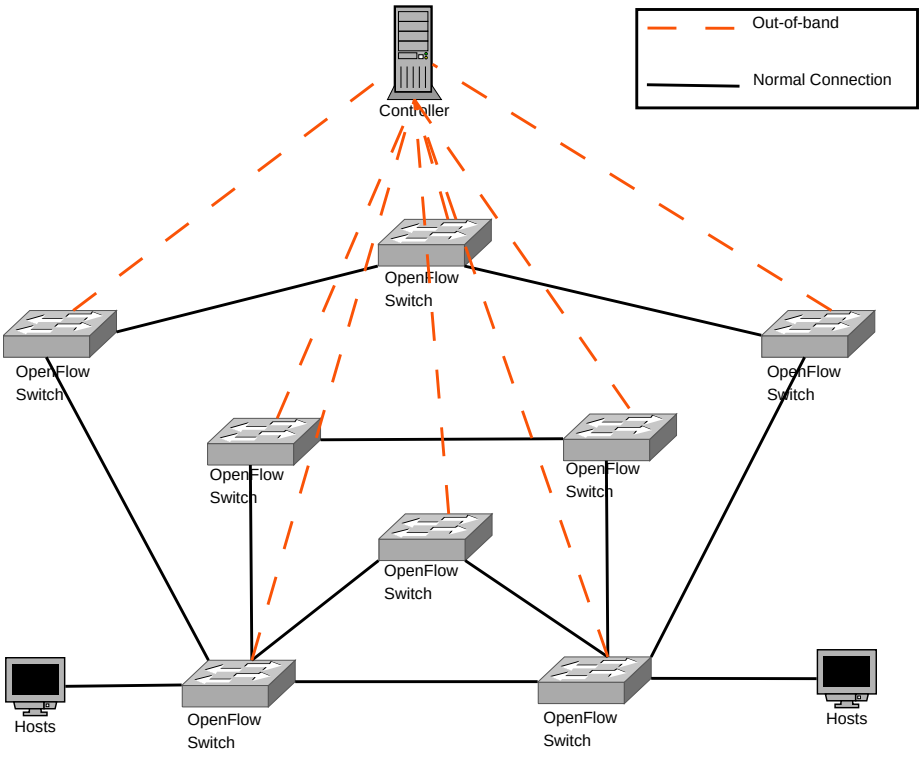

Figure 5: Network managed by control program

Figure 5 depicts the network that is managed by the control program. It consists of two hosts that can reach each other via four different paths where each link in the network has equal cost. RFC1812 [9] specifies requirements to routers, due to limited time we only focus on the following:

- Advanced Routing and Forwarding Algorithms
- Congestion Control
- High Availability

- Address Resolution Protocol - ARP

As stated before, the control program has a complete view of the network. In this case it is configured statically into the control program in a dictionary that represents a graph. Finding a path is done as follows:

1. The control program extracts the destination IP address and uses it to learn to which OpenFlow switch the destination host is connected.
2. Using a function that finds the shortest path between the source OpenFlow switch and the destination OpenFlow switch, the next-hop is learned. The path finding function returns a list of all OpenFlow switches in the shortest path. The second element in this list is always the next-hop. If the function returns a list that contains only one element it means that the end host is local to the OpenFlow switch.
3. The control program looks up the output port of the OpenFlow switch that leads to the next-hop.
4. Then a flow entry is created using the headers of the original packets and it adds an action that tells the OpenFlow switch to send packets matching this flow entry out of the specified port.

To find paths between vertices in our graph we used the functions described described by Van Rossum [10]. The author also presents a function that finds all possible paths between two vertices.

In some cases it is desired to let traffic be forwarded via different paths. As an example we implement a control program that monitors the type of traffic, based on TCP and UDP port numbers, that is active on the network. If traffic is detected that requests a higher level of service from the network, for example a researcher's application. It protects links used by such applications by finding paths that do not make use of the protected links. When protected links are not required anymore, their flow entries time out, the control program allows all traffic back on the previously protected links.

1. Control program checks the packet's TCP or UDP port number to determine the type of traffic.
2. If the packet is part of the researchers application, it marks links in the network as reserved.
3. If the packet is not part of the researchers application, the control program first checks if there are reserved links. If so, it routes traffic across the shortest path that does not contain reserved links. Otherwise traffic gets routed via the default shortest path.

Simulating link failures is a bit tricky in the test environment in use. All UML instances have their interfaces connected to a VDE switch that represents a direct link between the UML instances. If an interface at one side of the link goes down, this will go unnoticed to the interface on the otherside of the link. Whenever a link fails, or comes back up, on an OpenFlow switch, a message will be sent to the control program. The control program then performs the following steps.

1. It determines which link in the network has become (un)available.
2. Removes or adds the link to the graph.
3. Invalidates flow entries that used the now unavailable link.

The control program we wrote that implements the aforementioned functionalities can be found at the student's website [2].

[2] https://www.os3.nl/_media/2009-2010/students/arthur_van_kleef/pyswitch.zip

4 Test Environment

To experiment with OpenFlow, switches are needed that support the OpenFlow standard. During the period of this research project, the only options available were to make use of hardware switches that support OpenFlow, install multiple machines that are fysically connected to each other, or to make use of virtual machines like Xen [11], VMWare [12], or VirtualBox [13]. Because there is no hardware available and there will not be any budget available to invest in hardware for this project, we have to investigate the solutions that involve virtual machines. The OpenFlow websites offers several guides [14] guides to setup virtual environments to create OpenFlow networks in. Drawbacks of this approach that we encountered are: VM's like VMWare, Xen and VirtualBox require a lot of resources (CPU, RAM, storage), and because of this scaling becomes an issue in terms of available server machines and added complexity in VM configuration.

For this research project we have two dedicated server machines available that are equipped with an Intel(R) Pentium(R) D processor running at 3.0 GHz and 2GB internal memory. Storage is provided by a 80GB 7.200 SATA harddisk.

4.1 User Mode Linux

User-Mode-Linux (UML)[15] is a virtualization technology that enables Linux based systems to run multiple virtual machines to run as an application to the host.

By making use of programs like bridge-utils or Quagga, it is very easy to make a virtual machine act like a router or a switch. For OpenFlow the same is possible by making use of Open vSwitch [16], that offers a kernel module to be loaded into the Linux kernel. To use UML most Linux distribution provide precompiled packages that can be installed quite easily. Unfortunately, the UML binaries provided by for example Ubuntu are very outdated (based on Linux kernel version 2.6.22-rc5) and for which the kernel build files are not available. In this case the latest stable Linux kernel, version 2.6.34, was used to create an UML enabled Linux kernel. To support the Open vSwitch kernel module, the UML Linux kernel must have IPv6 and 802.1D bridging available as modules.

To run an UML based virtual machine a file system is required that provides the basic files required to start Linux. The most minimal virtual system is one that provides a console and a shell.

With only these components the virtual machine is useless, so more functionallity needs to be added. We added Busybox [17]. Busybox is a small executable that combines tiny versions of many common UNIX utilities in one binary. Because it is optimized for size and limited resources, we use it in the small experimentation environment for OpenFlow.

To support OpenFlow inside UML Open vSwitch is used. Open vSwitch is a virtual multilayer switch that supports the OpenFlow standard. To build Open vSwitch for use inside UML the kernel build modules of the UML enabled Linux kernel has to be available. The resulting kernel module and binaries have

to be installed to the file system that is used for UML.

UML based virtual machines can be connected by making use of virtual networking provided by Virtual Distributed Ethernet [18] (VDE[3]). UML enabled Linux kernels can be configured to create network interfaces that connect to a Unix socket created by VDE. In this setup VDE connections between UML instances will essentially represent the cables between the OpenFlow switches. To do this VDE switches are started in a so-called hub mode, which makes them act as a OSI model layer 1 device, and thus transparent to the UML instances.

4.2 UML Analysis

The amount of resources required by an UML based OpenFlow switch as described above is evaluated here. In this example an UML virtual machine with eight network interfaces was started and configured to run OpenFlow by making use of Open vSwitch. Each UML instance is by default started with 30M of RAM. The memory usage is determined via the *top* utility. The interesting column in the output from the host is the *RES* value, which stands for *Resident size* and reports the non-swapped physical memory a task has used. The value reported here is significantly lower than reported by the *VIRT* column (equivalent of *vSize*). So an UML virtual machine running Open vSwitch in this setup uses 36448Kb memory, of which 12MB is physical memory.

To test networking performance a topology was created consisting of an UML hosts with one network interface that connects it to an OpenFlow switch. This switch is again connected to a next OpenFlow switch and connects to a second host. The *iperf* utility can be used to perform network throughput measurements. One host runs *iperf* in server mode, the other acts as a client. Figure 6 shows the topology of the network, the results are shown in figure 7.

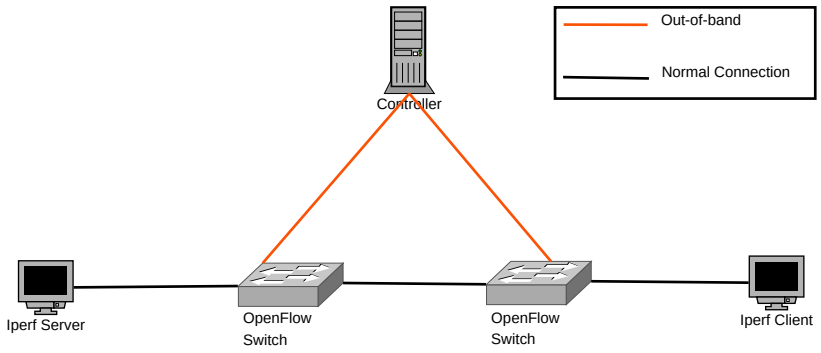

Figure 6: Iperf Topology with two OpenFlow switches.

To further test throughput performance more OpenFlow switches are added to the path between the two hosts. Using four OpenFlow switches decreased network throughput to 66 Mbit/sec, a drop of about60 percent compared to a setup that uses two OpenFlow switches.

[3] `http://vde.sourceforge.net/`

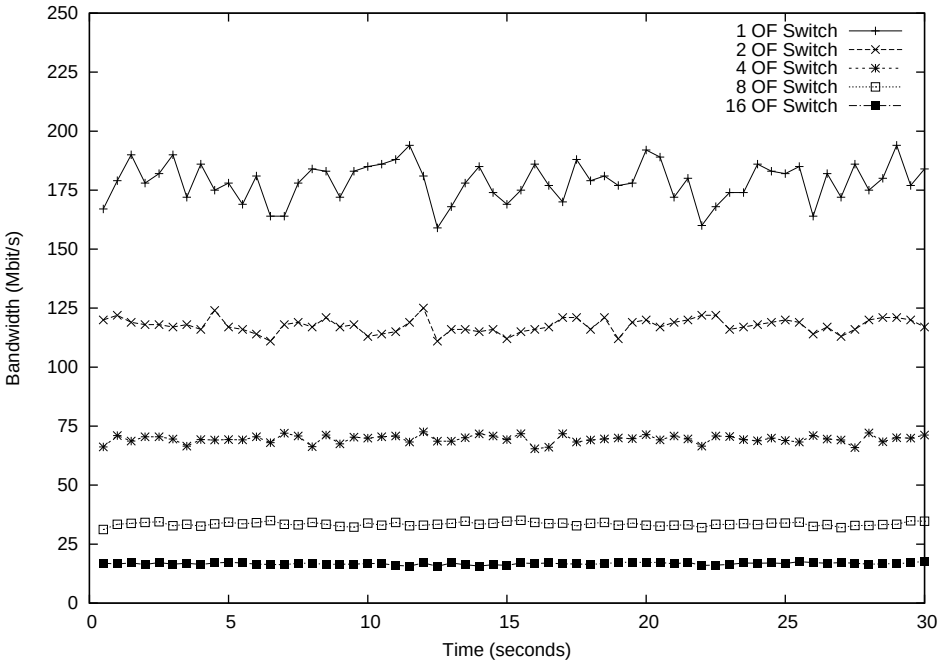

Figure 7: Measured througput on OpenFlow UML network.

The amount of space in use on the file system is obtained using *du*, a utility that shows disk usage statistics. The binaries that were installed together with Open vSwitch take 11.4M inside the */usr* directory. Not all of the binaries are required to run Open vSwitch, some of them can be deleted to further minimize disk usage. For debugging these were left them on the filesystem. Total disk usage on the file system is 19.8M.

When running multiple UML instances that require the same filesystem, two approaches can be taken in UML. First the filesystem can be mounted in read only mode, preventing the UML instances to write to the file system. But since Open vSwitch requires some directories to be writeable, these need to be made available. Since the process of making writeable directories available for each running UML virtual machine adds a lot of complexity, we decide to make use of the second option: copy-on-write (COW) files. With COW files, only one file system is required that serves as the base for all UML virtual machines. Whenever an UML instance makes a change to the file system, these will be stored inside the COW file rather than in the original disk image. For the purpose of running OpenFlow switches the UML instances do not write much to the file system. The average size for the COW file for one UML instance running Open vSwitch was about 280K.

5 Discussion

In section 4.2 the use of UML for running OpenFlow switches was analyzed. Unfortunately we did not manage to gather any significant statistics on CPU usage inside our UML instances. Using *top* CPU usage stayed at 0 percent at all times. Since the focus of this research project does not depend on maximum performance of the infrastructure, the limited bandwidth resources are not a big issue to perform experiments in. The result of using UML to create OpenFlow switches is that we can start *many* instances on moderate hardware, creation and destruction time of OpenFlow networks can be done in seconds by making use of start scripts, also because of making use of scripting we are not bound to a fixed topology to perform our experiments in.

In the experiments we performed we programmed the view of the network into the control program. In real-world use a discovery protocol like *Local Link Discovery Protocol* [19] (IEEE 802.1AB) should be used to construct a view of the network in the control program. Unfortunately this feature has not been implemented in the OpenFlow specification.

A point of concern when using a centralized control plane is the introduction of a single point of failure. This problem can be solved by replicating the control plane to a secondary OpenFlow controller. The OpenFlow switch implementation we used, Open vSwitch, chooses to cache flow entries for the case the control plane becomes unavailable. When this happens it will use cached flow entries to match packets.

Scalability may become an issue when the number of OpenFlow requests from network devices rise. The authors of [3] claim that NOX should be able to process over 100.000 requests per second. The cache timeout parameter in OpenFlow flow entries can help reduce the number of OpenFlow requests sent to the controller. Another option is to add an extra OpenFlow controller and load-balance requests as done by [20] and [21]. FlowVisor [22] can act as a virtualizion layer between OpenFlow switches and NOX. This way it is possible to balance the load of the request over the different controllers. It introduces the concept of regions to scale networks when they grow. Other applications of FlowVisor, like network slicing, are discussed in A.

While configuring UML to support OpenFlow using Open vSwitch we encountered several difficulties:

- To compile the Open vSwitch kernel module for use in UML we could not make use of pre-compiled UML binaries, because the Linux Kernel Headers for these versions are not available. To obtain these we had to compile an UML kernel, which also caused troubles since in many Linux kernel's the compilation process for an UML-enabled kernel failed. Eventually we managed to create an UML binary using the Linux kernel version 2.6.34.
- Documentation for Open vSwitch was very unclear. In our experiments we wanted to make use of static OpenFlow switch identifiers, documentation on how to configure this was wrong. One day after reporting this issue to the developers of Open vSwitch this bug was fixed.

Writing a control program for NOX proved to be a challenging task, very little documentation was available for us. We first had to thoroughly investigate the source code of the few examples that ship with the NOX software source code. Because of our limited programming experience it took some time to fully understand the programming environment offered by NOX.

6 Conclusion & Future Work

In this section we present our conclusions that we draw from the work we did on the topic of self-adaptive routing. We do this by reviewing our research questions from 1.

What is the architecture of a network that supports a self-adapting software control plane?

We have shown in section 3 that an architecture that offers a centralized control plane is best suited for the introduction of self-adaptive behaviour in networks. Supporting self-adaptiveness in networks that have distributed control planes requires all control planes to be consistent in terms of their view on the health of the network. An architecture that uses a centralized control plane does not have this issue. By making use of the OpenFlow protocol this requirement is met. The control plane resides on a centralized controller and allows network operators to program their network. Network configuration is done with a control program written in C or Python that runs at the NOX OpenFlow controller. The programmability of the OpenFlow controller makes an OpenFlow based architecture well suited for implemening self-adaptivity.

If the control plane becomes a programmable piece of software, what is the general pattern of the programs that implement routing and network management?

Self-adaptive behaviour is achieved by monitoring the health of the network and using this information in network configuration to get the desired level of service from the network. To implement this a feedback control loop has to be implemented in the control plane. By explicitly programming the control loop in the control plane, automatic network management is an integrated part of the control plane. OpenFlow enabled devices offer different measurement data 7 that can be used inside the control program. This data allows network operators to program constraints into their control plane to achieve self-adaptive behaviour. Even more advanced data can be obtained by making use of a protocol like NetFlow[23] or sFlow[24], which both are supported by the OpenFlow implementation Open vSwitch we used in our experimentation environment.

Furthermore, we have developed an environment that allows fast and easy creation of OpenFlow networks for the purpose of OpenFlow experiments. The combination of User-Mode-Linux, Open vSwitch and Virtual Distributed Ethernet allowed us to create virtual OpenFlow networks consisting of up to 100 OpenFlow switches running on only one server machine.

Future Research

A next step in self-adaptive networks is to create an interface to the control plane by which applications can specify their required service level. This makes the reference input of the feedback control loop variable. It would be interesting to see what implications this has for the control loop if too many applications request a different service from the network.

7 Acknowledgements

This report was created during the Research Project 2 course as part of the System and Network Engineering master at the University of Amsterdam. We would like to thank Rudolf J. Strijkers for providing for steering us in the right direction and giving us valuable feedback to our work.

References

[1] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and Control Element Separation (ForCES) Framework. RFC 3746 (Informational), April 2004.

[2] 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, 2008.

[3] 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, 2008.

[4] Wikipedia. Spanning tree protocol — wikipedia, the free encyclopedia, 2010. [Online; accessed 8-July-2010].

[5] Wikipedia. Provider backbone bridge traffic engineering — wikipedia, the free encyclopedia, 2010. [Online; accessed 8-July-2010].

[6] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. Updated by RFC 5709.

[7] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[8] A. Doria, F. Hellstrand, K. Sundell, and T. Worster. General Switch Management Protocol (GSMP) V3. RFC 3292 (Proposed Standard), June 2002.

[9] F. Baker. Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard), June 1995. Updated by RFC 2644.

[10] Guido van Rossum. Python patterns - implementing graphs. `http://www.python.org/doc/essays/graphs.html`, 1998. [Online; accessed 1-July-2010].

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[12] Inc. VMWare. Vmware virtual machine technology.

[13] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux J.*, 2008(166):1, 2008.

[14] OpenFlow Switching Consortium. Create an openflow network within a single pc. `http://www.openflowswitch.org/foswiki/bin/view/OpenFlow/Deployment/HOWTO/Virtual`. [Online; accessed 13-August-2010].

[15] Jeff Dike. A user-mode port of the linux kernel. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[16] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[17] Nicholas Wells. Busybox: A swiss army knife for linux. *Linux J.*, page 10.

[18] Renzo Davoli. Vde: Virtual distributed ethernet. In *TRIDENTCOM '05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Wikipedia. Link layer discovery protocol — wikipedia, the free encyclopedia, 2010. [Online; accessed 8-July-2010].

[20] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow networks, April 2010.

[21] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane, 2010.

[22] R. Sherwood, G. Gibby, K. Yapy, G. Appenzeller, M. Casado, N. McKeowny, and G. Parulkar. Flowvisor: A network virtualization layer, 2009.

[23] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[24] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational), September 2001.

A Network Slicing

There are various reasons to devide a network into multiple separated and isolated subnets. The most common reason to isolate parts of the network from eachother is enhance security and efficiency of the network. In classical networking the method of slicing by using Virtual Local Area Nnetworks (VLANs), which slices the network into isolated broadcast domains. With FlowVisor, the network can be sliced based on a wide range of criteria, which are described in section FlowVisor.

Virtual Local Area Network

In classical internetworking, networks are "sliced" using VLANs. VLANs can be used to share a physical infrastructure by multiple logical networks, whichout leaking information between the logical networks. THe 802.1q protocol adds a 4-bytes tag after the source MAC address in the Ethernet II frame when a packet from a node enters the switched environment. Between the switches, the packet is switchs including the header. Then the packet arives at the destination node, the last switch will strip the VLAN tag of the Ethernet II frame.

VLANs are used to separate the broadcast domains and isolate them from eachother. The 802.1q protocol is a layer two protocol, which separate the network in virtual local area networks. The VLANs are statically defined, although the assignment of VLANs to switch ports can be done dynamically based on layer 1 (physical switch port), layer 2 (MAC address), layer 3 (IP addresses) and layer 4 (protocol). Problems with assigning VLANs based on are experienced on various of these layers.

layer 2 The mapping between the VLAN and the MAC address give a big administrative overhead, because this needs to be done manually.

layer 3 Based on a table with VLAN/IP prefixes, the switch van map the source IP address to the VLAN id. With this method, the nodes already need to have an IP address and cannot request one using DHCP.

layer 4 End nodes can have connection to various network applications simultaniously, which creates the problem that a single end node can have multiple ports open on the network.

FlowVisor Slicing

`http://www.openflowswitch.org/wk/index.php/FlowVisor`

FlowVisor is a special controller that can act like a proxy between the OpenFlow switches and the NOX controllers. The FlowVisor controller can devide the resources in slices, which is a set of controlable resources. The slices can be defined as physical resources like switches and links, but also as packet header spaces like classes of packets (flow space). The slices can be defined based on the following criteria:

layer 1	switch ports
layer 2	source destination ethernet address ether type vlan vlan_pcp
layer 3	source destination IP address ip_proto ip_tos
layer 4	source destination TCP address source destination UDP address ICMP code type
QoS	

There are three actions which can be performed on a flowspace. With this actions, the allowed and restricted actions on the slice can be defined.

readonly Delegate only reading of the slice to a controller. This is useful for monitoring slices

allow Delegate the read and write control of a slice to a controller

deny Block the messages from this flowspace

The control of a specific slice is delegated to controller running a program like NOX or to another FlowVisor instance, which can delegate the control further. So it is possible to create a "default" slice for regular network traffic and a slice for the self–adaptive behavior for specific traffic on the network. The control of experiments can be delegated to another FlowFisor controller, for which the management can be delegated to the researchers. On this way, the production network and experimental network can be isolated from eachother.

Network slicing using flowspaces is much more flexibler that using classic VLANs. Slices can be used as a useful feature to create a framework to build self adaptive networks.

Slicing Applications

The FlowVisor controller can be used to slice the network. In the next chapter, a few applications of network slicing with FlowVisor are discussed.

Control Delegation

The control of the different slices can be delegated to a specific controller. This controller can be a NOX (or another) OpenFlow controller or to another FlowVisor controller to further slice the network.

Isolation

The different slices are separated from eachother in such a way that a problem on one slice will not affect the other slices. Error in a slice will not affect the other slices. This makes it possible to run more experiments on a network with more controllers, all controlling their own experiments.

Policies

In the network slice configuration files on the FlowVisor controller, the allowed or denied types of traffic can be defined. In the table the slicing criteria are defined. It is possible to define policies for specific users, nodes based on the slicing criteria the be handled in a specific manner. For example the web traffic of user A can be handled be another controller than the same web traffic for user B in the same subnet.

When certain types of traffic are not allowed on the network, they can be excluded from the flowspace. On this way, the packets of that traffic are not forwarded by the switches, because there is no controller who will calculate a flow entry for that type of traffic. Wherever the trafic enters the network. On this way the controllers are not overloaded with certain traffic that is not allowed on the network.

Implementation

To do experiments with the slices we setup an extensible architecture. The User Mode Linux environment is used to setup the OpenFlow switches, which are connected out of band with the FlowVisor controller. Below the FlowVisor controller, all type of networks can be created within UML.

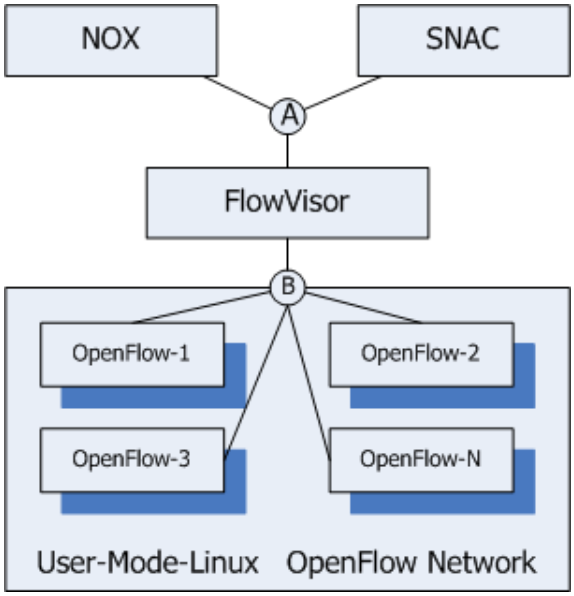

Figure 8: FlowVisor topology

The setup exist of two management subnets. The first subnet (subnet a) is the subnet between the NOX controllers and the FlowVisor controller. The following IP address schema is used to address the nodes of the network.

Network Address	192.168.0.0
FlowVisor	192.168.0.1
NOX	192.168.0.102
SNAC	192.168.0.103

The second network is the subnet between the FlowVisor controller and the OpenFlow switches in User Mode Linux. The connection between the controller and switches is out of band, because of lack of performance of in-band connection. The following IP address schema is used to address the switches and (proxy) controller of the network:

Network Address	192.168.1.0
FlowVisor	192.168.1.1
OpenFlow-01	192.168.1.101
OpenFlow-N	192.168.1.1N

Configuration

The FlowVisor controller has the directory *flowvisor-conf.d*, which contains the configuration files. There are two types of files, namely .switch and .guest files.

```
cat default.switch

# This configuration file is used for default switches
Default: 1
# The numbering of the switches will start at 1000
Id: 1000
```

This configuration file handles the switches that are not configured within this FlowVisor controller. It will start numbering the switches with 1000. This file can also contain specific switch configurations.

```
cat nox.guest
Name: nox
ID:  1
Host:  tcp:192.168.0.102:54321
FlowSpace: allow:

cat snac.guest
Name: snac
ID:  1
Host:  tcp:192.168.0.103:6633
FlowSpace: readonly:
```

The first configuration file will point to the NOX controller located on 192.168.0.102 listening on port 54321. The FlowSpace: allow: statement is left empty, so that all the flows will match for this controller. The second configuration points to the SNAC controller and defines the same slice (match all traffic) with readonly permissions, so this can be used for monitoring.

The FlowVisor controller is started used the following command to let the controller listen to OpenFlow switches on the default port (6633)

```
./home/openflow/openflow/flowvisor/flowvisor ptcp:
```

The guest controllers are started using the normal command to connect to the FlowVisor controller. To test the setup, we will use the spanning_tree module and the pyswitch module for the controller of the slice nox.guest.

```
./home/openflow/nox/build/src/nox_core -i ptcp:54321 spanning\_tree pyswitch
```

The slice snac.guest controller SNAC will automatically run the nox configuration to monitor the network. Finally, the switches need to connect to the FlowVisor IP address.

Network

To test the infrastructure, we set up various networks. The NOX controller will run the spanning_tree and the pyswitch module to create a loop free topology and to be able to switch the packets over the network.

All the eth0 interfaces of the switches are reserved for the out of band connection to the FlowVisor controller. These interfaces are not drawn in the pictures, because these are used everywhere in the same way. The interfaces of OpenFLow-N uses the IP address 192.168.1.10N. All the interfaces that are not eth0 are added to bridge br0, so OpenFlow can use those interfaces as switch interfaces.

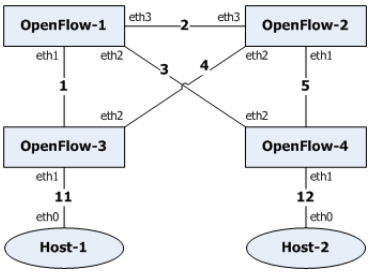

Figure 9: Core and Edge layer

This network in Figure 9 has a core and edge layer to connect the nodes to the core of the network. There is one spanning tree instance over the whole topology, although this can be implemented to use for instance two spanning tree instance for a single broadcast domain.

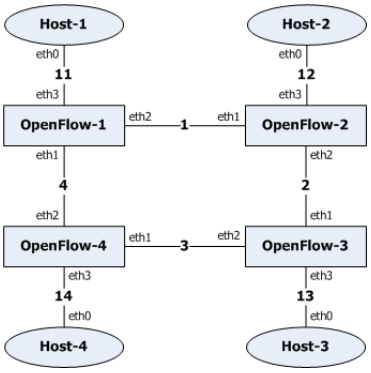

Figure 10: Ring

This ring network in Figure 10 has nodes connected to every switch in the ring. The spanning tree algoritm will cut one connection in the ring to create a loop free topology

The network in Figure 11 is more an experimental topology to see how the spanning tree module reacts on multiple loops in a topology.

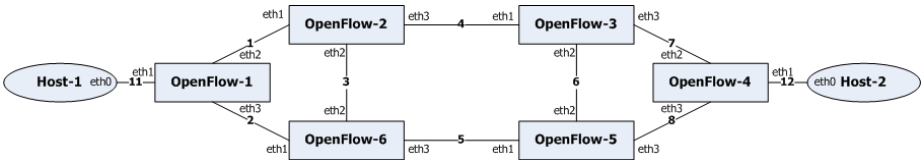

Figure 11: Partitial Mesh

B OpenFlow Flow Entry Counters

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Packets	64
Transmit Bytes	64
Transmit Overrun Errors	64

Table 7: OpenFlow Statistics Counters

C OpenFlow Traffic Analysis

To analyze the OpenFlow protocol, we attached a sniffer (tcpdump) on the network segment that connects all OpenFlow switches to the NOX controller. The NOX controller was started without any applications, the NOX controller will not do anything but accept and setup connections from OpenFlow switches.

TCP Connection Setup

Connections between OpenFlow switches and NOX controller are setup via TCP and are initiated by OpenFlow switches. NOX listens for incoming connections (default TCP port 6633.

The first packets on the wire are Address Resolution Protocol (ARP) packets sent by OpenFlow switches, trying to identify which hardware address (MAC) uses the IP of the controller. After this an TCP connection between OpenFlow switch and NOX controller is established:

```
1 0.000000 be:b9:47:1b:b9:51 Broadcast ARP Who has 192.168.1.1?  Tell 192.168.1.101
2 0.000028 12:34:76:99:01:01 be:b9:47:1b:b9:51 ARP 192.168.1.1 is at 12:34:76:99:01:01
3 0.004013 192.168.1.101 192.168.1.1 TCP 50631 > 6633 [SYN] Seq=0 Win=5840 Len=0 MSS=1460
4 0.004085 192.168.1.1 192.168.1.101 TCP 6633 > 50631 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=
5 0.004800 192.168.1.101 192.168.1.1 TCP 50631 > 6633 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV
```

OF Hello

When the TCP connection is established, both the OpenFlow switch and the NOX controller must start by sending an *OFPT_HELLO* message. This message identifies the highests supported version of the OpenFlow protocol:

```
No.     Time        Source                Destination           Protocol Info
      6 0.005447    192.168.1.101         192.168.1.1           OFP      Hello (SM) (8B)

Frame 6 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51), Dst: 12:34:76:99:01:01 (12:34:76:
Internet Protocol, Src: 192.168.1.101 (192.168.1.101), Dst: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: 50631 (50631), Dst Port: 6633 (6633), Seq: 1, Ack
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Hello (SM) (0)
        Length: 8
        Transaction ID: 894662002
```

In our case, both NOX and Open vSwitch support the OpenFlow 1.0 specification (*Version 0x01*).

Features

Next NOX will ask which *Features* are supported by the switch.

```
No.     Time        Source                Destination           Protocol Info
      9 0.005731    192.168.1.1           192.168.1.101         OFP      Features Request
```

```
Frame 9 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src: 12:34:76:99:01:01 (12:34:76:99:01:01), Dst: be:b9:47:1b:b9:51 (be:b9:47:
Internet Protocol, Src: 192.168.1.1 (192.168.1.1), Dst: 192.168.1.101 (192.168.1.101)
Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 50631 (50631), Seq: 9, Ack
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Features Request (CSM) (5)
        Length: 8
        Transaction ID: 0
```

To which the OpenFlow switch answers with an *Features Reply* message:

```
No.     Time        Source                Destination           Protocol Info
     13 0.010720    192.168.1.101         192.168.1.1           OFP      Features Reply (C

Frame 13 (242 bytes on wire, 242 bytes captured)
Ethernet II, Src: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51), Dst: 12:34:76:99:01:01 (12:34:76:
Internet Protocol, Src: 192.168.1.101 (192.168.1.101), Dst: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: 50631 (50631), Dst Port: 6633 (6633), Seq: 9, Ack
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Features Reply (CSM) (6)
        Length: 176
        Transaction ID: 0
    Switch Features
        Datapath ID: 0x0000002320d1ff7e
        Max packets buffered: 256
        Number of Tables: 2
        Capabilities: 0x00000087
            .... .... .... .... .... .... .... ...1 =   Flow statistics: Yes (1)
            .... .... .... .... .... .... .... ..1. =   Table statistics: Yes (1)
            .... .... .... .... .... .... .... .1.. =   Port statistics: Yes (1)
            .... .... .... .... .... .... .... 0... =   802.11d spanning tree: No (0)
            .... .... .... .... .... .... ...0 .... =   Reserved: No (0)
            .... .... .... .... .... .... ..0. .... =   Can reassemble IP fragments: No (0
            .... .... .... .... .... .... .0.. .... =   Queue statistics: No (0)
            .... .... .... .... .... .... 1... .... =   Match IP addresses in ARP pkts: Ye
        Actions: 0x000007ff
            .... .... .... .... .... .... .... ...1 =   Output to switch port: Yes (1)
            .... .... .... .... .... .... .... ..1. =   Set the 802.1q VLAN id: Yes (1)
            .... .... .... .... .... .... .... .1.. =   Set the 802.1q priority: Yes (1)
            .... .... .... .... .... .... .... 1... =   Strip the 802.1q header: Yes (1)
            .... .... .... .... .... .... ...1 .... =   Ethernet source address: Yes (1)
            .... .... .... .... .... .... ..1. .... =   Ethernet destination address: Yes
            .... .... .... .... .... .... .1.. .... =   IP source address: Yes (1)
            .... .... .... .... .... .... 1... .... =   IP destination address: Yes (1)
            .... .... .... .... .... ...1 .... .... =   Set IP TOS bits: Yes (1)
            .... .... .... .... .... ..1. .... .... =   TCP/UDP source: Yes (1)
```

```
.... .... .... .... .... .1.. .... .... =   TCP/UDP destination: Yes (1)
.... .... .... .... .... 0... .... .... =   Enqueue port queue: No (0)
```

An interesting field in the *Features Reply* message sent by the OpenFlow switch is the 802.11D spanning tree capability. According to the OpenFlow specification, a spanning tree must be constructed by OpenFlow switches before contacting a controller. Open vSwitch does not support 802.11D Spanning Tree. The *actions* field specifies which actions can be used in *flow entries*. Also the *Max packets buffered* switch feature might be of importance in terms of scalability.

Configuration

The NOX controller also sends a configuration message to the OpenFlow switch, that instructs the switch to only send the first 128 bytes of a packet for which it requests a flow.

```
No.     Time        Source                Destination           Protocol Info
     11 0.009742    192.168.1.1           192.168.1.101         OFP      Set Config (CSM)

Frame 11 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 12:34:76:99:01:01 (12:34:76:99:01:01), Dst: be:b9:47:1b:b9:51 (be:b9:47:
Internet Protocol, Src: 192.168.1.1 (192.168.1.1), Dst: 192.168.1.101 (192.168.1.101)
Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 50631 (50631), Seq: 17, Ac
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Set Config (CSM) (9)
        Length: 12
        Transaction ID: 0
    Switch Configuration
        Flags
            .... .... .... ..00 =   Handling of IP fragments: No special fragment handling
        Max Bytes of New Flow to Send to Controller: 128
```

Vendors

OpenFlow also offers support to distinguish between *Vendors*, this can be done by specifying the vendor's IEEE OUI, or (if vendors do not wish to use IEEE OUI's) by obtaining an OUI from the OpenFlow consortium.

```
No.     Time        Source                Destination           Protocol Info
     14 0.010965    192.168.1.1           192.168.1.101         OFP      Vendor (SM) (24B)

Frame 14 (90 bytes on wire, 90 bytes captured)
Ethernet II, Src: 12:34:76:99:01:01 (12:34:76:99:01:01), Dst: be:b9:47:1b:b9:51 (be:b9:47:
Internet Protocol, Src: 192.168.1.1 (192.168.1.1), Dst: 192.168.1.101 (192.168.1.101)
Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 50631 (50631), Seq: 29, Ac
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Vendor (SM) (4)
```

```
        Length: 24
        Transaction ID: 0
Vendor Message Body: 00002320000000080000000000000000
```

Open vSwitch's OpenFlow implementation doesn't understand the vendor extension, and as specified by the OpenFlow 1.0.0 specification responds with an error message:

```
No.     Time        Source                Destination           Protocol Info
     16 0.014930    192.168.1.101         192.168.1.1           OFP      Error (SM) (36B)

Frame 16 (102 bytes on wire, 102 bytes captured)
Ethernet II, Src: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51), Dst: 12:34:76:99:01:01 (12:34:76:
Internet Protocol, Src: 192.168.1.101 (192.168.1.101), Dst: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: 50631 (50631), Dst Port: 6633 (6633), Seq: 185, A
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Error (SM) (1)
        Length: 36
        Transaction ID: 0
    Error Message
        Type: Request was not understood (1)
        Code: Vendor subtype not supported (4)
        Data: 0104001800000000000023200000000800000000000000000
            OpenFlow Protocol
                Header
                    Version: 0x01
                    Type: Vendor (SM) (4)
                    Length: 24
                    Transaction ID: 0
            Vendor Message Body: 00002320000000080000000000000000
```

Flow Modification

Finally, the controller instructs the OpenFlow switch to clear it's Flow Table, which might contain previously installed *flow entries*:

```
No.     Time        Source                Destination           Protocol Info
     17 0.015285    192.168.1.1           192.168.1.101         OFP      Flow Mod (CSM) (7

Frame 17 (138 bytes on wire, 138 bytes captured)
Ethernet II, Src: 12:34:76:99:01:01 (12:34:76:99:01:01), Dst: be:b9:47:1b:b9:51 (be:b9:47:
Internet Protocol, Src: 192.168.1.1 (192.168.1.1), Dst: 192.168.1.101 (192.168.1.101)
Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 50631 (50631), Seq: 53, Ac
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Flow Mod (CSM) (14)
        Length: 72
        Transaction ID: 1342454536
```

```
    Flow Modification
        Match
            Match Types
                .... .... .... .... .... .... .... ...1 =   Input port: Wildcard (1)
                .... .... .... .... .... .... .... ..1. =   VLAN ID: Wildcard (1)
                .... .... .... .... .... .... .... .1.. =   Ethernet Src Addr: Wildcard (1
                .... .... .... .... .... .... .... 1... =   Ethernet Dst Addr: Wildcard (1
                .... .... .... .... .... .... ...1 .... =   Ethernet Type: Wildcard (1)
                .... .... .... .... .... .... ..1. .... =   IP Protocol: Wildcard (1)
                .... .... .... .... .... .... .1.. .... =   TCP/UDP Src Port: Wildcard (1)
                .... .... .... .... .... .... 1... .... =   TCP/UDP Dst Port: Wildcard (1)
                .... .... .... .... ..11 1111 .... .... =   IP Src Addr Mask: /0 (63)
                .... .... .... 1111 11.. .... .... .... =   IP Dst Addr Mask: /0 (63)
                .... .... ...1 .... .... .... .... .... =   VLAN priority: Wildcard (1)
                .... .... ..1. .... .... .... .... .... =   IPv4 DSCP: Wildcard (1)
        Cookie: 0x0000000000000000
        Command: Delete all matching flows (3)
        Idle Time (sec) Before Discarding: 0
        Max Time (sec) Before Discarding: 0
        Priority: 0
        Buffer ID: 0
        Out Port (delete* only): None  (not associated with a physical port)
        Flags
            .... .... .... ...0 = Send flow removed: No (0)
            .... .... .... ..0. = Check for overlap before adding flow: No (0)
            .... .... .... .0.. = Install flow into emergecy flow table: No (0)
        Output Action(s)
            Warning: No actions were specified
```

Echo

In order to monitor the health of a controller-switch connection *echo request* and *echo reply* messages are exchanged periodically. These messages can be used to indicate latency, bandwidth and liveness of the connection.

```
No.     Time        Source                Destination           Protocol Info
     73 4.370907    192.168.1.101         192.168.1.1           OFP      Echo Request (SM)

Frame 73 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51), Dst: 12:34:76:99:01:01 (12:34:76:
Internet Protocol, Src: 192.168.1.101 (192.168.1.101), Dst: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: 50631 (50631), Dst Port: 6633 (6633), Seq: 221, A
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Echo Request (SM) (2)
        Length: 8
        Transaction ID: 0

No.     Time        Source                Destination           Protocol Info
     74 4.371340    192.168.1.1           192.168.1.101         OFP      Echo Reply (SM) (
```

```
Frame 74 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src: 12:34:76:99:01:01 (12:34:76:99:01:01), Dst: be:b9:47:1b:b9:51 (be:b9:47:
Internet Protocol, Src: 192.168.1.1 (192.168.1.1), Dst: 192.168.1.101 (192.168.1.101)
Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 50631 (50631), Seq: 125, A
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Echo Reply (SM) (3)
        Length: 8
        Transaction ID: 0
```

Packet In

In this example, with no active application running at the controller, a ping message is sent out from Host1 to Host2 in the following topology:

```
H1<--->S1<--->S2<--->S3<--->S4<--->H2
```

Each OpenFlow switch, denoted by Sx, is out-of-band connected to the controller. Host1 will start by sending out an *Packet In*, that is sent to the controller:

```
No.     Time        Source                Destination           Protocol Info
    145 33.980382   36:b8:26:15:5f:fc     Broadcast             OFP+ARP  Packet In (AM) (B

Frame 145 (126 bytes on wire, 126 bytes captured)
Ethernet II, Src: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51), Dst: 12:34:76:99:01:01 (12:34:76:
    Destination: 12:34:76:99:01:01 (12:34:76:99:01:01)
        Address: 12:34:76:99:01:01 (12:34:76:99:01:01)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT
    Source: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51)
        Address: be:b9:47:1b:b9:51 (be:b9:47:1b:b9:51)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT
    Type: IP (0x0800)
Internet Protocol, Src: 192.168.1.101 (192.168.1.101), Dst: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: 50631 (50631), Dst Port: 6633 (6633), Seq: 269, A
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Packet In (AM) (10)
        Length: 60
        Transaction ID: 0
    Packet In
        Buffer ID: 256
        Frame Total Length: 42
        Frame Recv Port: 1
        Reason Sent: No matching flow (0)
        Frame Data: FFFFFFFFFFFF36B826155FFC0806000108000604000136B8...
```

```
Ethernet II, Src: 36:b8:26:15:5f:fc (36:b8:26:15:5f:fc), Dst: Broadcast (ff:ff
    Destination: Broadcast (ff:ff:ff:ff:ff:ff)
        Address: Broadcast (ff:ff:ff:ff:ff:ff)
        .... ...1 .... .... .... .... = IG bit: Group address (multicast/broad
        .... ..1. .... .... .... .... = LG bit: Locally administered address (
    Source: 36:b8:26:15:5f:fc (36:b8:26:15:5f:fc)
        Address: 36:b8:26:15:5f:fc (36:b8:26:15:5f:fc)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..1. .... .... .... .... = LG bit: Locally administered address (
    Type: ARP (0x0806)
Address Resolution Protocol (request)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    Sender MAC address: 36:b8:26:15:5f:fc (36:b8:26:15:5f:fc)
    Sender IP address: 172.16.1.100 (172.16.1.100)
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 172.16.1.101 (172.16.1.101)
```

At the NOX controller, this will be received as an *event*. Modules for NOX that enable operators to program the controller, can register for events like *packet_in* and based on the received packet tell an OpenFlow switch what to do with the packet.