NTNU – Trondheim
Norwegian University of
Science and Technology

Evaluating SDN and SDN-based Multicast for Network Intensive Services in UNINETT

Alessandro Menabo

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Otto Wittner, ITEM
Co-supervisor: Lars Landmark, -
Mario Baldi, Politecnico di Torino, Italy

Norwegian University of Science and Technology
Department of Telematics

Problem Description

This project main objective is to investigate to what extent an SDN-based routing architecture for UNINETT (based on available and upcoming standards and HW/SW) may provide more dynamic and flexible routing without sacrificing the level of dependability when compared to the currently applied routing system. Novel designs with respect to utilizing multicast features is of special interest. Controller placement and organization as well as management network operations are also relevant topics.

The thesis will start with an overview of SDN theory, motivations, main concepts, possible applications and current state of the art, comparing SDN to traditional network organization. Typical use cases for research network providers (specifically UNINETT) will be described, as well as how the use of SDN in such scenarios can help making network and service deployment easier.

Special focus will be given to traffic-intensive network services where multicast may be utilized, e.g. videoconferencing and distributed cluster computing. How can SDN make a difference in routing and allocating resources for such use cases, e.g. by taking network statistics collected from physical devices more into account?

Possible network architectures shall be designed and tested with existing simulation tools. Reliability and stability issues will be taken into account. If time allows, network designs will be deployed and evaluated on real hardware.

Abstract

The goal of this thesis is to demonstrate and evaluate how Software-Defined Networking (SDN) techniques can help provision a flexible network service in support of videoconferencing applications using multicasting and network service chaining. Specifically, we show how OpenFlow and the Ryu controller can be used to implement multicasting at network level and route part of the traffic through a middlebox that converts high-quality streams into low-quality ones, in order to accommodate users with limited access bandwidth. After reviewing the main theoretical foundations behind this work, a solution is designed and tested on a sample network topology emulated with Mininet. The results and experience gained from this work confirm that SDN is a promising approach to computer networking that makes service deployment and management easier and allows for better utilization of network resources.

Acknowledgements

I would like to express my most sincere gratitude and admiration towards my supervisors for the time, patience and valuable suggestions they granted me during the work on this thesis. Not only did they help with the technical and organizational aspects of my work, but also they gave me much freedom in choosing the thesis topic and showed me the importance of creativity in technology and innovation.

A special thanks goes to my family for always showing me love and support, teaching me good values and common sense, and encouraging me to pursue an honest and balanced life style. My one-year experience in Norway would have hardly been possible without their help.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software-Defined Networking (SDN) is a computer networking approach in which the control plane (routing) is decoupled from the data plane (packet switching) in order to make network and service management simpler, cheaper and more flexible. This is in contrast to traditional distributed control where intelligence and switching functions coexist within the same physical device, resulting in complex and "ossified" networks. SDN is currently mostly deployed in data centers and controlled environments, where it is used for custom routing and service chaining, but experiments on a large scale are being carried out.

Multicast is a network communication model with a single source and multiple receivers, and is therefore better suited for group communication. As opposed to unicast transmissions, multicast allows efficient usage of network resources and decreased workload on hosts and servers by duplicating traffic only where needed. Unfortunately, multicast today has several security and scalability issues, and is not very well supported by service providers or applications. Typical applications of multicast are multimedia streaming and conferencing.

UNINETT, the Norwegian research network provider, is investigating how SDN may be useful for its service needs. This thesis will look into possible use cases of SDN and multicast for a service provider. We will design and implement a solution for a videoconferencing service based on custom routing using network emulation and SDN software tools.

1.1 Main Objectives

The main goals of this thesis project are:

- gaining a better understanding of the state-of-the-art, opportunities and limitations concerning multicast and SDN;
- investigating if and how combining SDN and multicast can lead to more flexible service deployment in a videoconferencing context;
- gaining expertise with existing network emulation tools and software frameworks for SDN;
- designing, implementing and evaluating a working solution to the selected videoconferencing scenario, enhanced with network service chaining.

1.2 Thesis Organization

The thesis is structured as follows:

- Chapter 1 introduces and motivates the thesis.
- Chapter 2 offers an overview of multicasting in today's networks.
- Chapter 3 presents the main concepts behind Software-Defined Networking and its most popular protocol, OpenFlow.
- Chapter 4 describes the selected videoconferencing scenario and sketches a solution using SDN and OpenFlow.
- Chapter 5 walks through the practical work of implementation and verification of the designed solution.
- Chapter 6 discusses the results and achievements from the implementation with a critical eye.
- Chapter 7 summarizes the content of the thesis and points out possible areas of future work.

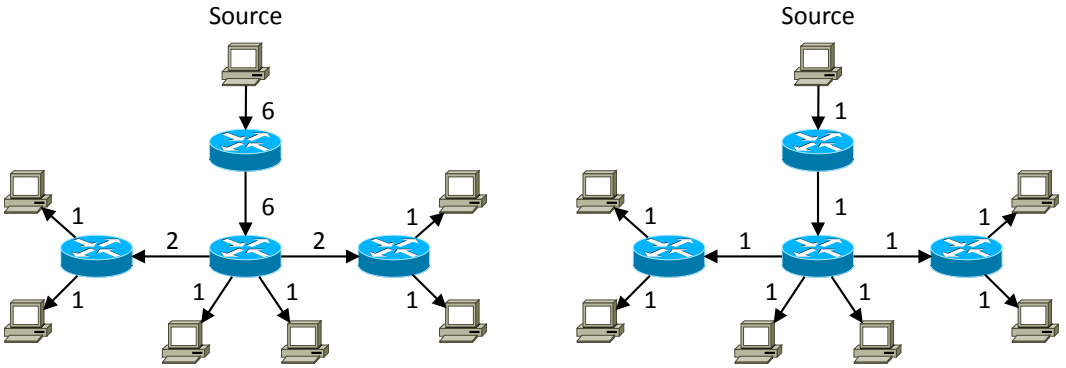
Chapter 2

Multicast

Network communications can be divided into three main categories based on the cardinality of the receivers. At one end we have one-to-one communications, also known as *unicast*, involving exactly one sender and one receiver. At the opposite end we have *broadcast* communications, representing one-to-all interactions in which a sender transmits a message to every other node in the network. In between lie group communications, or *multicast*, where messages are destined to a group of receivers who have previously subscribed to the group (one-to-many and many-to-many).

One naïve way to implement multicast is through multiple unicast streams from the source: the source sends a copy of the same message to each recipient. However, this is highly inefficient and not scalable, because it increases the workload on senders and routers and consumes more bandwidth in the network. A better approach that alleviates the burden on sources and links is to have the source send a single copy of the message, which intermediate nodes will replicate only where needed, namely where the paths towards the various destinations begin to diverge. This requires intermediate nodes to be multicast-aware: in this regard, multicast can exist natively at the network layer, if supported by routers, or as an application-level overlay network, in which case regular hosts perform routing and switching. Figure 2.1 shows the key difference between the naïve and efficient implementations of multicast; the number next to each arrow indicates the number of copies of the same packet sent over the link.

This thesis and the rest of this chapter focus on network-level multicast in IPv4, but the same concepts also apply to IPv6 with some minor modifications.



(a) Unicast-based group communication (b) Multicast-based group communication

Figure 2.1: Comparison of unicast and multicast group communication

2.1 Applications

Multicast is well suited for applications that take advantage of the logical grouping of hosts, especially if large amounts of data need to be transmitted to multiple receivers at the same time. In this case, the benefits over unicast, in terms of bandwidth save and workload reduction, are tangible. Typical examples are multimedia applications like audio/video conferencing and live streaming in IPTV. In the conference example, a group may represent a conversation, and all participants in the same conference belong to the same group. In the IPTV example, each group may represent a TV channel, while members are all customers watching the same channel at a given time.

Multicast is also a valid alternative to broadcast whenever the identity and number of receivers are unknown to the sender, but just a fraction of the total number of hosts. This is the case of discovery protocols such as SSDP and data center monitoring tools [1]. In this sense, multicast implements a limited and controlled form of broadcast in which hosts that are not participating in a given service or protocol do not receive unwanted traffic (broadcast messages must be flooded everywhere and processed by every host, while multicast traffic can be filtered by routers, switches and NICs).

2.2 IP Multicast

Multicast groups in IP are identified by class D addresses. Class D addresses all start with "1110", therefore addresses between 224.0.0.0 and 239.255.255.255 (the range 224.0.0.0/4)

are reserved for multicast [2]. Multicast addresses can only be used as destination, while the source must be unicast. Rather than using a protocol like ARP for unicast, multicast MAC addresses are derived statically from L3 addresses. When hosts subscribe to a multicast group, they reconfigure their NIC to accept frames with the multicast MAC address as destination and deliver them to the operating system kernel.

In contrast to IP unicast, senders may not know in advance who they are transmitting to. Similarly, receivers may receive traffic from any source transmitting to a multicast address. This communication model is called Any-Source Multicast (ASM). The security implications (addressed in Section 2.4) have led to the introduction of a different service model called Source-Specific Multicast (SSM) [3], which allows receivers to explicitly specify the list of sources from which they are willing to accept traffic. This however requires receivers to know the senders in advance, which is not always possible.

In order to receive multicast traffic for a group, end hosts need to join the group by subscribing to its address. The Internet Group Management Protocol (IGMP), originally described in [2] and later revised in [4] (IGMPv2) and [5] (IGMPv3), manages host subscriptions within a single LAN. The actors involved in the protocol are the hosts and the default gateway. The role of the default gateway is to keep track of active groups in the LAN using IGMP and to participate in multicast routing with other routers in the core network.

IGMPv2 defines three types of message:

- *Host Membership Query*: periodically issued by routers to discover if any groups have active members in the LAN;
- *Host Membership Report*: sent by hosts to notify their subscription to a group, either in response to a membership query or in an unsolicited way;
- *Leave Group*: sent by hosts when they gracefully leave a group.

IGMPv3 introduces support for Source-Specific Multicast by extending the format of membership reports.

2.3 Multicast Routing

Edge routers use local membership information learned via IGMP to coordinate with other multicast routers (*mrouters*) and build a distribution tree for each group. It may

happen that not all core routers are also mrouters, in which case multicast packets must be tunneled from one mrouter to another inside unicast packets in order to prevent traditional routers from dropping them.

Not all multicast routing algorithms build an explicit distribution tree. There are two types of trees:

- *source-specific trees*: one tree exists per source per group, and it is rooted at the source; all sources and destinations are connected via the shortest path, but computation of the tree is expensive;
- *shared trees*: each group uses the same tree for all sources; paths may not be optimal but their computation is less expensive.

2.3.1 Flooding schemes

The simplest way to deliver multicast traffic is to flood it throughout the whole network. It is simple and reliable but wastes bandwidth, because packets are sent everywhere, even where there are no receivers. Besides, loops in the network cause packet storms, unless routers are able to recognize previously seen packets. Despite these serious drawbacks, flooding may be appropriate in networks with high density of receivers. Reverse Path Forwarding (RPF) and Reverse Path Broadcasting (RPB) improve the basic flooding scheme with a reverse path check to avoid loops: if packets are received from an interface that is not on the shortest path towards the source, they must have looped and are therefore discarded.

As a further enhancement, Reverse Path Multicasting (RPM) introduces the concept of *pruning*, which removes entire subtrees from the main forwarding tree in case no receivers exist along the subtree. Pruning is initiated by edge routers and notified to the upstream node. Routers must be able to revert the pruning state and rejoin the main tree in case new receivers appear. RPM is featured in DVMRPv3 (Distance Vector Multicast Routing Protocol version 3) [6] with prune state timeouts and in PIM-DM (Protocol Independent Multicast – Dense Mode) [7] with explicit rejoin requests. RPM is best suited in networks with high density of receivers, where prune messages are infrequent.

Flooding schemes produce implicit source-specific trees. However, their optimality is guaranteed only if the reverse path to the source is the same as the forward path from the source, which is not always the case in IP networks due to asymmetric routing.

2.3.2 Link state multicast routing

Link state protocols can be extended to support multicast routing. Every router has a global view of the network and can calculate optimal source-specific trees from every source using Dijkstra's algorithm. This, however, is an expensive process that does not scale well. MOSPF (Multicast Open Shortest Path First), defined in [8] and obsoleted in [9], extends the OSPF link state protocol by adding a new type of LSA without breaking the compatibility with the unicast version of the protocol. It attempts to improve performance by delaying tree calculation until the first packet from a new source is detected, thus implementing on-demand routing, but this is still not enough to make the protocol really scalable.

2.3.3 Core-Based Trees and PIM-SM

Core-Based Trees (CBT) are shared distribution trees centered at one specific router called the *core*. The algorithm finds the shortest path from the core to each destination according to the unicast routing tables. When traffic is sent to a multicast group, the packets first go to the core, which in turn forwards them along the tree. The presence of a common core enables easier and faster construction of the tree. However, even if the paths from any sender to the core and from the core to any receiver are optimal, the end-to-end paths usually are not. Tree construction begins at the edge, and if a new receiver joins the group at a later time, it is attached to the nearest node already in the tree, thus making tree updates faster.

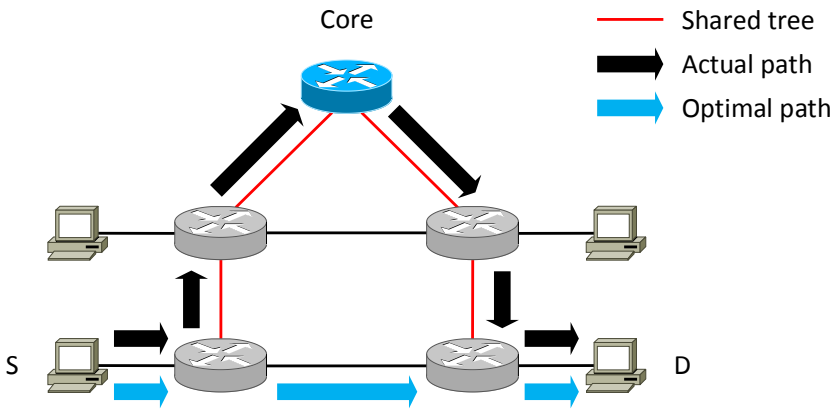


Figure 2.2: Example of a CBT with non-optimal end-to-end paths

PIM-SM (Protocol Independent Multicast – Sparse Mode) [10] is the most successful and widespread multicast routing protocol today. It is an explicit-join protocol: traffic

is forwarded only on those links from which join messages from other routers have been received. This avoids unnecessary flooding and makes the protocol best suited when the receivers are sparse. PIM-SM initially creates a CBT for each group, where the core is called *Rendezvous Point* (RP), but it may switch to an optimal source-specific tree whenever a receiver gets a sufficient amount of traffic from the same source; when this happens, the old path from the receiver to the shared tree is pruned. PIM-SM is thus able to balance network efficiency and computational cost, starting with a shared tree that is easy to compute and switching to a shortest source-specific tree if convenient. It also implicitly supports the SSM service model.

2.4 Relevant Issues

Group communications are substantially different from the unicast case. Therefore, many applications, protocols and common practices do not fit well in a multicast scenario. For instance, IP multicast was not conceived with security in mind. The identity and number of receivers in multicast communications are typically not known in advance to senders and routers. The implications on security and scalability often make service providers reluctant to deploy multicast on a large scale on their networks, unless in a tightly controlled environment.

2.4.1 Scalability, access control and denial of service

The lack of access control in traditional IP multicast allows any host to join any group and send traffic to its members. In addition, group sizes are not fixed and can vary with high dynamicity, and multicast routers need to maintain state and routing information for every group. Any host joining an existing group may cause the distribution tree to expand, thus increasing the amount of traffic flowing in the network. Malicious senders may also inject bogus packets to all members of a group [11]. While the latter issue can be mitigated with the use of SSM, other forms of access control are required to prevent unauthorized hosts from joining a group. These factors seriously limit the scalability of IP multicast and turn it into a possible vector for DoS attacks on both the hosts and the network.

2.4.2 Authentication and privacy

Security mechanisms like IPsec and SSL/TLS are tailored to point-to-point communcations, and adapting them to multicast requires some workarounds.

Security requirements vary from application to application [11]. Sometimes only authentication is required: this is the case of public data that needs to be verified, like stock market updates and routing messages. In other cases, secrecy is more important than authentication, as in pay-per-view services where an IPTV broadcaster wants to prevent non-subscribers to view their channels. Often, authentication and encryption are both desired, such as in multimedia conferencing.

Even SSM, although it restricts the allowed sources to a set of trusted unicast addresses, cannot protect against IP address spoofing or authenticate senders. A proposed extension of IPsec is based on group Security Associations and a centralized trusted entity devoted to the negotiation and distribution of security parameters between members of a group [12].

SSM also does not help when you want to limit the scope of a group, i.e. to restrict the set of authorized receivers. Consider the case of a multicast group used to carry sensitive data within a corporate network: if no scoping mechanisms were implemented, any host outside the corporate network could join the same group and gain access to the sensitive data. Possible solutions are to send packets with a limited time-to-live (*TTL scoping*) or to use scoped group addresses (*administrative scoping*) [13] as defined in [14]. Both solutions require gateways to be configured with appropriate TTL thresholds and scope boundaries.

2.4.3 Reliability

Achieving reliable, connection-oriented transmissions in multicast is not a trivial task. The TCP protocol was designed for point-to-point communications and the mechanics of acknowledgment do not fit well in a multipoint scenario. If every receiver replied to every single TCP segment with an ACK, the traffic thus generated would easily saturate the network and overwhelm the sender, effectively causing a DoS attack. This basically limits the transport protocol to UDP only and makes file transfers and other loss-sensitive applications difficult to adapt to a multicast scenario. Therefore, if required, reliability must be enforced in a different way, either with ad-hoc transport protocols [12] or at the application layer.

Chapter 3

Software-Defined Networking

Computer networks have much evolved through the years. Network switches and routers have been enriched in functionality and performance to handle ever increasing traffic loads; specialized appliances like firewalls, media gateways and load balancers, generically called *middleboxes*, have been introduced to face new application requirements; traffic patterns and market demands are constantly changing and moving towards massive virtualization. As a result, networks have become very complex and delicate, and experimentation is difficult and risky, while at the same time there is an increasing need for flexibility and innovation. However, the software running on network devices is strictly closed-source and new features can only be obtained either through the purchase of new equipment from a different vendor, or by issuing a feature request to the original supplier, which is likely to be very expensive and require a long development cycle. Besides, appliances from different manufacturers are often incompatible when used on the same network due to protocol implementation differences, thus invalidating the first option too and making customers totally dependent on a single vendor (*vendor lock-in*). All these reasons have led to the so-called *ossification* of today's networks. Sofware-Defined Networking (SDN) aims at overcoming these limitations by adopting a radically different network organization, as explained in the next section.

3.1 SDN Architecture

In traditional networking, every device exchanges topology information with all the others in order to build a map of the network and calculate paths between nodes. The software and protocols dedicated to this task constitute the *control plane* of the device.

The results of the control plane are then used to configure the local *data plane*, which is the part of the device that forwards packets between input and output ports to reach the correct destinations. Traditional networks are therefore characterized by a distributed control plane, as every switch or router integrates both control and data planes.

The idea behind SDN is to decouple the control plane from the data plane and move all the intelligence and complexity to a logically centralized entity called the *controller*. The controller runs the relevant algorithms and configures each network element accordingly. In theory, this enables the use of simpler, "dumb" and therefore cheaper hardware that only deals with forwarding packets and has no computing tasks. In practice, though, line-rate switching still requires special purpose and high quality components. The real breakthrough with SDN is that now the intelligence of the network is no more under exclusive control of the equipment manufacturer: network operators and service providers now have the freedom and tools to implement custom routing using common programming languages.

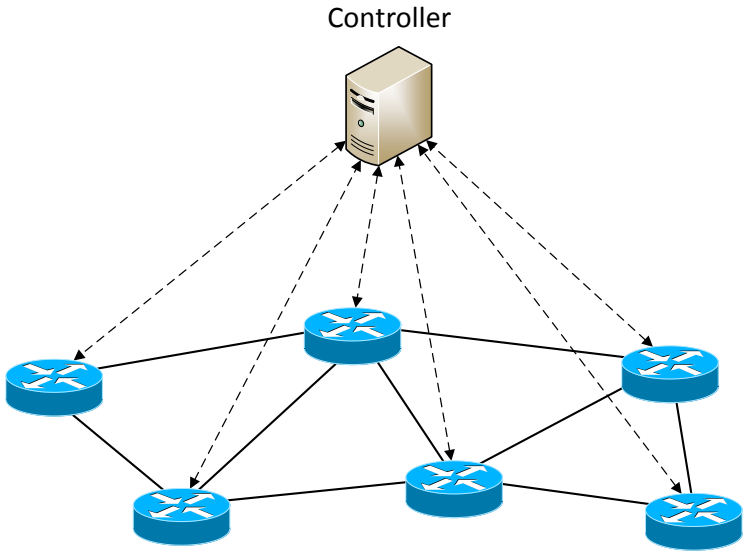


Figure 3.1: Centralized control plane in SDN

The SDN architecture can be broken up into three layers or planes, each with an abstract view of the underlying layers that favours simplicity and modularity. Between them, well-defined interfaces provide the necessary abstraction and communication protocols. The *data plane* consists of forwarding units (switches and routers) and middleboxes that perform switching and packet processing, but do not take on an active role in routing. The *control plane* sits upon and manages the data plane via a logically centralized controller. While in principle the controller can be a single machine, and therefore

a single point of failure, it is often deployed as a cluster of controllers for fault tolerance and load balancing. At the top, the *application plane* hosts a variety of network applications that take advantage of a high-level view of the network. Examples of network applications are policy enforcement, network virtualization, traffic monitoring, and even routing protocols.

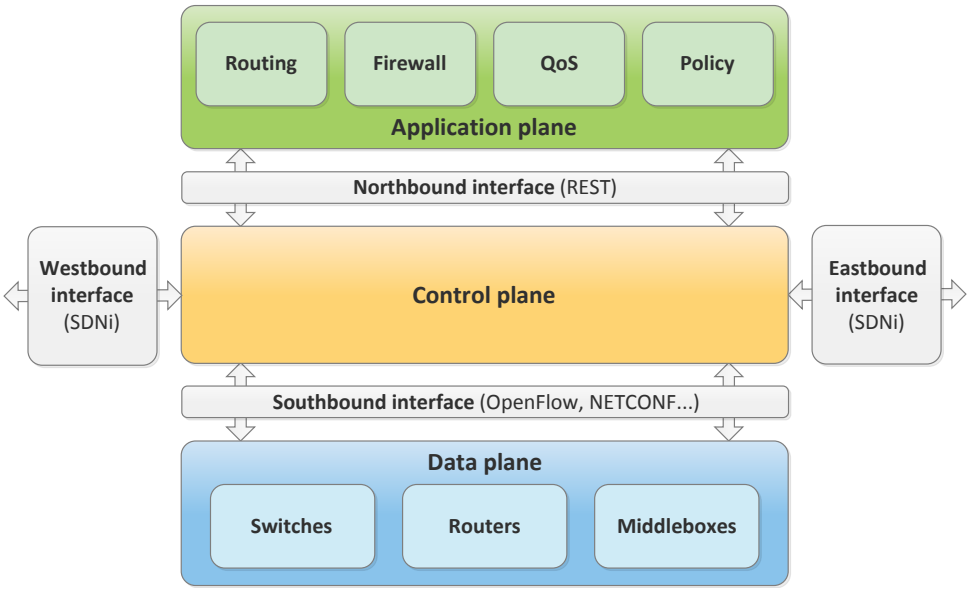


Figure 3.2: SDN domain architecture

The control and data planes communicate through the *southbound interface*, which allows the controller to configure individual devices and query their statistics. The most widespread southbound protocols are NETCONF [15], OpenFlow [16], and the OVSDB management protocol [17] for the Open vSwitch virtual switch [18]. Applications and controllers interact via the *northbound interface*. Currently no standard northbound APIs exist, but REST architectures with XML or JSON data are popular choices. Finally, eastbound and westbound interfaces should allow interoperation and state information exchange between independent SDN domains.

3.2 Applications

It is often the case that network traffic needs to be routed according to other rules than traditional destination-based routing. For example, one may wish to redirect a subset of the incoming traffic through firewalls, content caches and accelerators, deep packet inspectors, or other appliances, creating a *network service chain* (NSC). The reasons behind

this are disparate; a possible use case is a service provider offering firewall and backup services to customers for a fee. As the need for these services varies from customer to customer and over time, SDN can provide the necessary flexibility to implement custom routing policies.

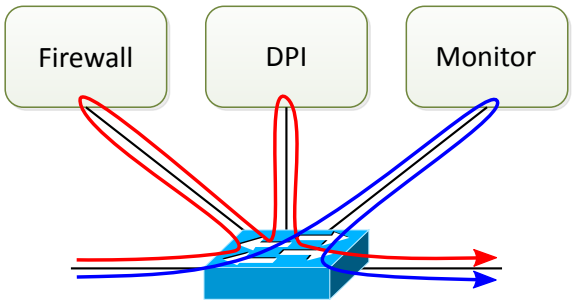
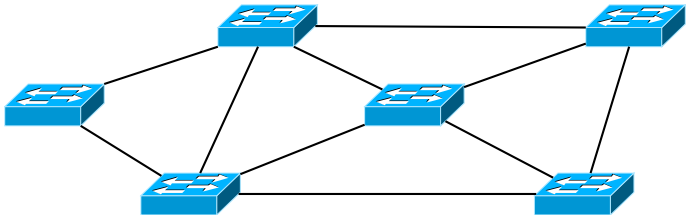


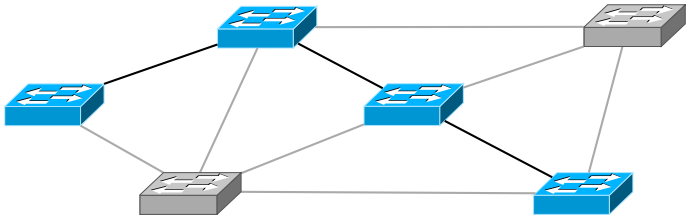
Figure 3.3: Example of network service chaining with two traffic classes

These network services are often virtualized for cost efficiency and easier management and deployment, in what is termed as Network Functions Virtualization (NFV) [19, § 7]. A single general-purpose server can host tens or even hundreds of such virtual machines. The virtualization software on the server provides each virtual appliance with a virtual network interface and implements a software switch (virtual switch) that connect them together and to the data center network via the server's physical interfaces. NFV is independent of SDN, but SDN concepts are often used by the virtual machines orchestration software to dynamically reconfigure physical switches in the network and virtual switches on the servers whenever a service chain is modified or a virtual machine is migrated to a different server. In this sense, the orchestrator takes on the role of SDN controller.

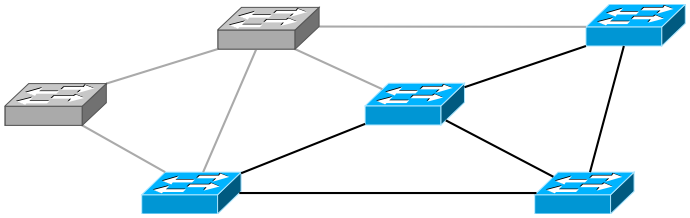
Another recent trend in cloud computing is to sell unused computing, storage and network resources to customers in order to balance the costs of running an often over-provisioned data center infrastructure [19, § 6]. This leads to multi-tenant data centers where the same physical infrastructure is shared among several users, but each user maintains full control over their own virtualized subset of resources in complete isolation from every other tenant. This is referred to as *network slicing* and can be easily achieved using SDN techniques. In the same context, tenants can configure custom routing policies and run their own network applications on their slice without intervention from data center operators, e.g. via a web-based interface towards the underlying SDN platform.



(a) Physical network



(b) First tenant's slice (blue)



(c) Second tenant's slice (blue)

Figure 3.4: Example of slicing with two tenants sharing the same infrastructure

3.3 OpenFlow

OpenFlow is an open protocol for programming the forwarding plane of network elements from a logically centralized controller. OpenFlow was initially developed at Stanford University in 2008 as a way to experiment with custom protocols on existing networks [16]. It is now maintained by the Open Networking Foundation (ONF) since 2011 and has become the de facto standard southbound interface in SDN. This section is based on the OpenFlow 1.3.1 specification described in [20].

An OpenFlow switch, also called *datapath*, is a forwarding unit consisting of a set of ports and a pipeline of *flow tables* that contain the forwarding rules of the switch. Each flow table entry is a set of packet header fields and values to match, associated actions to execute and counters for statistics. Upon receiving a packet, the switch searches the flow tables for a match and, if found, applies the specified actions and updates the counters. If a match is not found, the packet is simply dropped. An OpenFlow switch abstracts common switching components like ports and queues and provides statistics such as

the number of packets matching a flow, the number of transmission and receive errors, and more. Controllers can leverage these statistics to implement more intelligent and flexible routing.

3.3.1 Flow tables

OpenFlow supports a number of header fields ranging from layer 1 to layer 4 of the OSI reference stack. Possible fields include, but are not limited to, the input port on the switch, Ethernet source and destination addresses, IPv4/IPv6 source and destination addresses, UDP/TCP source and destination ports, VLAN tags, MPLS labels, IP ToS (Type of Service) field, ICMP type and code and ARP opcode. Some of these fields, notably Ethernet and IP addresses, are maskable, meaning that it is possible to partially match those fields. A match field whose value is not specified is said to be a *wildcard* and effectively matches any value. Because a packet can match several wildcard flow entries, it is possible to assign a priority value to each entry. Exact matches, on the other hand, always have precedence over any wildcard entry.

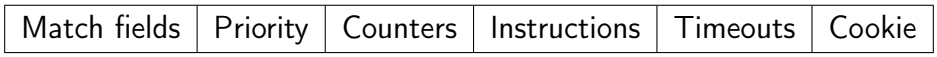


Figure 3.5: Structure of a flow table entry

Each flow entry has an associated set of instructions that are executed for each matching packet. Flow entries may expire after a certain amount of time without any matches (*idle timeout*) or regardless of this (*hard timeout*), whichever comes first. Flow counters measure the number of packets and bytes that matched the flow as well as the total age of the flow. The cookie is an opaque value that can be set by the controller to easily identify a single entry.

A special entry, called *table-miss flow entry*, can be used to handle the case where no other flow entry is matched. This entry is defined as all wildcards with zero (lowest) priority. Although it does not need to be installed in any flow table, the switch must support it and at least be able to drop the packet or send it to the controller.

3.3.2 Instructions and actions

The standard requires that OpenFlow switches support at least the following two instructions:

- *Write-Actions* [action-list]: the enclosed actions are added to the current action set of the packet, possibly overwriting previous actions of the same type;
- *Goto-Table* [table-id]: the packet and its current action set are sent to the specified flow table where the lookup and matching process starts over.

The *Goto-Table* instruction allows to arrange several flow tables in a pipeline. To avoid loops, the target table identifier must be greater than the current one. The main advantage of a pipeline over a single table is the possibility to group different classes of rules (e.g. forwarding and filtering) in separate tables and keep the total number of flow entries small while still covering all possible cases.

An OpenFlow-compliant switch must support at least the following actions:

- *Output* [port-id]: the packet is sent out of the specified port, which can be a physical port, a logical port (vendor-dependent, not defined by OpenFlow, e.g. a tunnel endpoint or link aggregation group) or a reserved port defined by OpenFlow;
- *Group* [group-id]: the packet is processed through the specified group table entry (see Section 3.3.3);
- *Drop*: the packet is dropped (this is the default behaviour when no match is found or no output or group action is specified).

Among the optional actions, the most relevant are:

- *Push-Tag/Pop-Tag*: the switch may be able to push and pop the outermost element of the VLAN, PBB and/or MPLS stack;
- *Set-Field*: the switch may be able to rewrite some of the packet header fields, such as addresses and port numbers.

3.3.3 Group table

Group tables, introduced in OpenFlow 1.1, allow for easier and more efficient flow action managament and richer switch behaviour. Each group entry contains an identifier, a group type, and counters and actions similar to flow counters and actions. Actions are arranged in a list of *action buckets*, where each bucket is a set with at most one action of each type (e.g. at most one output action per bucket). Groups can be used to group

common flow actions together: with several flow entries pointing to the same group, it is possible to change the actions of all those flows by simply modifying the common group.

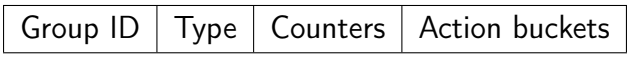


Figure 3.6: Structure of a group table entry

The group type determines the semantics of the group and how the buckets are executed. Possible group types are:

- *All* (required): all buckets are executed and the packet is cloned for each of them; used to implement multicasting and broadcasting;
- *Indirect* (required): only one bucket is allowed; equivalent to an All group with a single bucket;
- *Select* (optional): multiple buckets are allowed, but only one is executed; the selection algorithm is up to the switch implementation and out of scope of the OpenFlow standard, however a weighted load balancing algorithm is suggested;
- *Fast Failover* (optional): each bucket is associated with a port or group whose liveness is monitored; this group executes the first live bucket available and can be used to react to failures without controller intervention.

Chapter 4

Design

This chapter begins with a description of possible use cases for UNINETT in which SDN and multicast may be relevant. We will then select one of these use cases and design a solution to a proposed test scenario. Goals and requirements shall be clearly stated and related to the concepts and features discussed in the previous chapters.

4.1 Use Cases for UNINETT

UNINETT [21] is Norway's National Research and Education Network provider (NREN) and offers connectivity and other network-related services to Norwegian universities, educational institutions and research centers over a high-capacity network infrastructure. As part of the thesis work, we analyzed UNINETT's service offer in search of suitable applications for SDN and SDN-based multicast.

UNINETT offers backup and storage services in the cloud environment. In this context, data are often replicated and spread out over different locations in the data center in order to achieve fault-tolerance through redundancy. As stated in the introduction of Chapter 2, one of the advantages of multicast over unicast is that packets to multiple receivers need only be duplicated at some points in the network, thus limiting bandwidth consumption. Bandwidth saving becomes evident with high volumes of data, as is the case for file system and database replication. SDN can also help in finding the least loaded paths at any given time by keeping track of link allocation and performing traffic engineering.

Another set of services offered by UNINETT belongs to the multimedia domain and

includes live video streaming, IPTV and audio/video conferencing. The multimedia conferencing service supports both H.323 and SIP clients and provides a Multipoint Control Unit (MCU) for both signaling and media processing. The MCU acts as the "meeting room" for all participants, mixes media streams from different sources and delivers a single stream to each participant. As such, all communications between participants are actually done in unicast to and from the MCU. In addition, some of the participants may be located in parts of the network with unstable connections or limited access bandwidth, such as radio links in cellular networks. The currently deployed service does not take this into account, and improvements can be made with respect to stream delivery (replacing unicast with multicast) and custom routing through high-capacity links or dedicated appliances via SDN.

4.2 Description of Scenario

The selected use case for this project is the videoconferencing scenario, enhanced with service chaining. For our purposes, hosts are divided into two groups, according to their access bandwidth and capabilities. *Full-capacity hosts* connect to the network via reliable high-speed links and can therefore handle multimedia streams at high quality. In contrast, *low-capacity hosts* cannot handle streams at the same quality level and would greatly benefit from a downgrade. *Transcoders* are dedicated network appliances that perform quality downgrading, compression and re-encoding of multimedia content in order to accommodate the limited capabilities of low-capacity hosts.

The control plane should take this distinction into consideration and apply two different routing schemes:

- full-capacity hosts should communicate directly among them, without a central MCU, using multicast;
- low-capacity hosts should also send traffic directly to other participants, but should receive only from a transcoder.

The selected network topology is shown shown in Figure 4.1 and comprises six OpenFlow switches, one transcoder and five hosts. Among these, h4 is the only low-capacity host and should receive traffic from transcoder T1. All hosts belong to the same logical group, which is identified with the multicast address 239.192.0.1, in compliance with the guidelines for administratively scoped addresses found in [14].

The SDN application must be able to calculate source-specific trees from every potential source of traffic (hosts and transcoders) and automatically install the corresponding flow and group entries on traversed switches. In order to implement service chaining, outgoing traffic from each host must additionally be routed to the transcoder, which will then perform media processing and inject compressed traffic back into the network towards h4. We will refer to the source-specific tree from the transcoder towards all low-capacity hosts as the *low-capacity tree*. Low-capacity trees must be distinguished from regular trees in order to avoid inconsistencies and loops; we choose to mark packets coming from the transcoder with a special DSCP value (63) in the IP ToS field.

We assume that all necessary signaling procedures have already been completed and the controller has a stable view of the network. This simplifies the design and helps focus on service chaining and switch configuration. We can imagine that the controller and the application learn about end host capabilities during the signaling phase, e.g. via extensions to the SIP signaling protocol.

The final distribution trees, as well as flow and group entries on each switch, shall not be defined here, as these will be dynamically calculated by the SDN application based on whatever topology is chosen. Details on how topology data is used by the application to derive trees and forwarding rules for switches are given in Section 5.2.2. Here we shall instead describe how the designed forwarding behaviour can be achieved using features of the OpenFlow protocol (see Section 4.2.2).

4.2.1 Objectives

We will now summarize the main requirements for our test scenario:

- *source-specific trees*: the application should be able to build optimal distribution trees rooted at each traffic source (hosts and transcoders);
- *automatic switch configuration*: the application should automatically convert the source-specific trees into flow and group entries and install them on all switches along each tree;
- *ToS-marking*: switches should be able to distinguish low-capacity trees from the others by means of a special value in the IP ToS field;
- *multicasting*: packet forwarding should take advantage of multicasting in order to limit bandwidth usage in the network and duplicate packets only where needed;

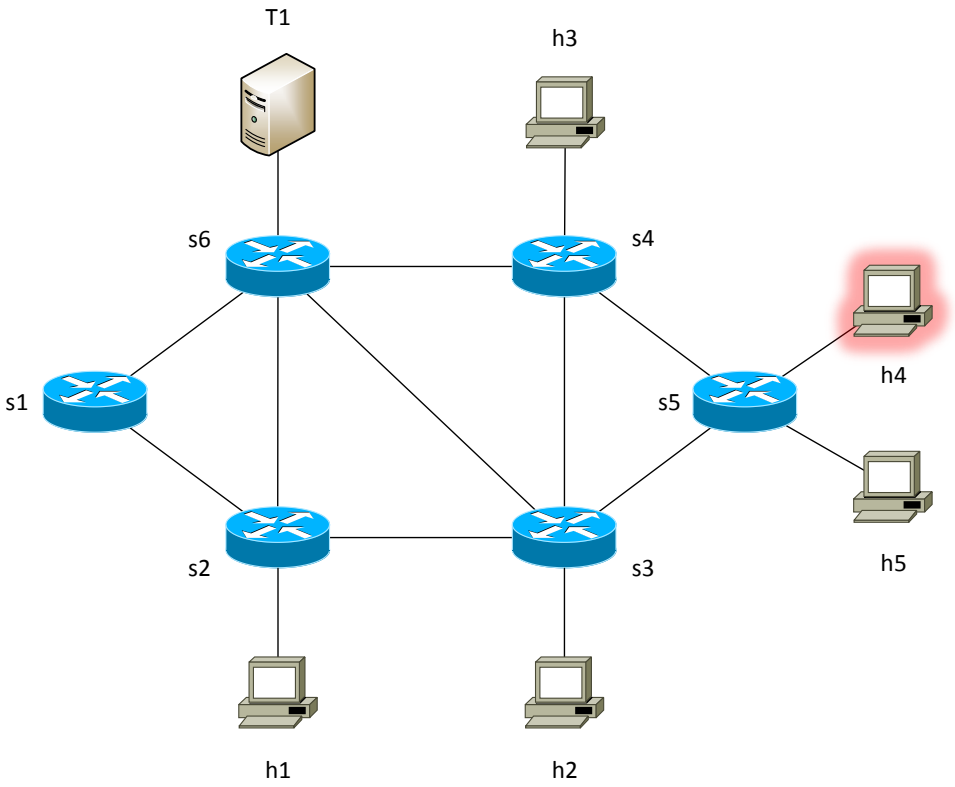


Figure 4.1: Tested network topology

- *service chaining*: low-capacity hosts should receive traffic indirectly through the transcoder, therefore outgoing traffic from every host must be delivered not only to full-capacity hosts, but also to the transcoders, and from there to low-capacity hosts; the first switch along the low-capacity tree will change the ToS field of all packets coming from the transcoder.

In addition, we would also like to meet three additional requirements:

- *transparency*: end hosts should be, as far as possible, unaware of custom routing and chaining, and our solution should have little or no impact on existing applications (this includes maintaining the original source address in packets coming back from the transcoder);

- *access control*: the set of hosts that are allowed to send traffic to the multicast group should be restricted in order to avoid denial of service attacks;

- *spoofing protection*: IP address spoofing can be easily mitigated with simple reverse path checks.

4.2.2 Solution with OpenFlow

Multicasting is supported by all OpenFlow versions, with a slight difference between version 1.0 and 1.1+. In OpenFlow 1.0 it is possible to specify multiple OUTPUT actions in a single flow rule. Group tables, introduced in OpenFlow 1.1, allow the same behaviour via the GROUP action with multiple action buckets, each containing a single OUTPUT action. Group tables also enable grouping of flow actions (hence the name) so that multiple flows can point to the same group, and modifying a single group results in all flows being updated in a single transaction. For this reason, an OpenFlow version greater than 1.0 was preferred. We decided to use the highest version available that started to see some vendor support at the time the experiments were carried out, so our choice fell on OpenFlow 1.3. All versions of OpenFlow silently prevent sending a packet back over the interface it was received (unless the reserved port IN_PORT is explicitly used), so it is perfectly safe to create a single group entry to be executed for all packets coming from any port along the multicast tree.

Multiple source-specific trees can easily coexist on the same switch by creating a flow rule and a group for every source whose tree includes that switch, and matching flows on source address. This also implicitly enables us to perform access control, because any packet that does not match on any known source address is automatically dropped. Relevant match fields for low-capacity trees are input port, source IP address (unicast), destination IP address (multicast), and ToS value. For standard trees, only input port and IP addresses need to be matched. Explicitly matching on input ports implements a simple reverse path check and protects us from address spoofing.

Finally, the SET_FIELD action is used to mark traffic coming from the transcoder before sending it down the low-capacity tree. This action is only performed by the switch where the transcoder is attached. In summary, switches need to be configured according to the rules listed below.

- For every source-specific tree from a host traversing the switch:
  - Flow match: input port towards host, source host IP address, destination multicast address.
  - Flow action: GROUP.

  - Group: type ALL, one bucket with OUTPUT action for every port towards full-capacity hosts and transcoders.

- For every low-capacity tree from a transcoder traversing the switch (and if the switch is not the attachment point for that transcoder):
  - Flow match: input port towards transcoder, source transcoder IP address, destination multicast address, IP ToS.
  - Flow action: GROUP.
  - Group: type ALL, one bucket with OUTPUT action for every port towards low-capacity hosts.

- For every transcoder attached to the switch:
  - Flow match: input port towards transcoder, destination multicast address.
  - Flow actions: SET_FIELD ToS, GROUP.
  - Group: type ALL, one bucket with OUTPUT action for every port towards low-capacity hosts.

Chapter 5

Implementation and Testing

Following the description of the test scenario given in the previous chapter, it is now time to show how they have been implemented and evaluated. First we present the tools chosen to accomplish the tasks. We then show how these tools were combined to build and verify the application, and we discuss the difficulties encountered during implementation and the results obtained from the testing phase.

5.1 Tools

This sections gives an overview of the software tools used in the implementation phase and motivates their choice over possible alternatives.

5.1.1 Network emulation

In order to achieve the maximum degree of flexibility and take advantage of the features of OpenFlow 1.3 without depending on the availability of physical devices or vendor support for OpenFlow, all experiments were carried out in a virtualized network environment created with Mininet [22].

Mininet is a network emulation tool written mostly in Python and based on Linux network namespaces. Hosts are modeled as regular processes running in user space; as such, they all share the same filesystem and can run any programs and scripts that are available on the host (this also helps with management and testing). Every host runs in its own network namespace and is connected to a switch via a virtual Ethernet pair pro-

vided by the namespace. Mininet exposes a Python API that can be used to build custom topologies, configure hosts addresses, send commands to hosts and invoke a command-line interface (CLI) that allows real-time interaction with the emulated network. Mininet supports several software switches, the default being Open vSwitch running in kernel mode. Version 2.2.1 was used.

Open vSwitch (OVS) [23, 24] is a multilayer software switch typically used to interconnect virtual machines. The Linux kernel includes OVS by default since version 3.3. Open vSwitch supports OpenFlow 1.3; support for versions 1.4 and 1.5 is currently under development. OVS performs fast flow lookup and packet switching as a kernel module, while the OpenFlow protocol and various management utilities run in user space. The most relevant utility is ovs-ofctl, the OpenFlow management tool. Open vSwitch 2.3.1 was used in this project.

As both Mininet and Open vSwitch require a Linux operating system, and all tutorials and guides found on the web refer to the Ubuntu distribution, this was also the choice for this project. Ubuntu 14.10 (later upgraded to 15.04) was hosted on a Windows 8.1 machine through VirtualBox.

5.1.2 Choice of controller platform

The choice of the controller was driven mainly by two factors. First of all, as explained in the previous chapter, one of the requirements was to take advantage of group tables and other features available in OpenFlow 1.3. This quickly narrowed down the list of suitable controllers to a very small set, in which the most promising options seemed to be OpenDaylight [25] and Ryu [26]. Both provide an SDN framework consisting of an OpenFlow controller and built-in modules that implement some common network functions, such as topology discovery, L2 switching and simple L3 routing, and expose APIs to external applications.

The second motivating factor was the degree of complexity of the framework along with the availability and quality of documentation and online support. OpenDaylight is a very lively community-led project with contributions from several major vendors as well. This has led to a very complex and feature-rich framework which is somewhat difficult to master. Besides, many of its built-in features were not needed in our small-scale project. Ryu, on the other hand, has a simpler architecture, yet provides a fairly complete SDN solution that covers many use cases. The online documentation is lacking in some parts, but it is backed up by an excellent book full of examples with source code explained [27]. Another source of documentation is the code itself, which is not always

well commented but can be inspected rather easily.

For all these reasons the choice fell on Ryu, version 3.18. At the time when this choice was made, it was not yet entirely clear whether the solution would be implemented as a module inside the controller, or as an application on top of it. Controller modules can interact with the network directly via the controller's native OpenFlow API for better performance, but this makes the application tightly coupled to the specific platform. On the other hand, external applications use the simpler northbound API, which adds some overhead but is easier to port. Both frameworks are suited for the second option, but the first thought was to use the built-in module approach, and Ryu was chosen for its simplicity. Later, when the decision was taken to switch to an external application, Ryu was kept as the controller of choice because enough familiarity had already been gained with it. The Ryu module chosen to interact with the network is the ofctl_rest.py module, which provides a REST OpenFlow API for modifying flow and group entries, get statistics and change port behaviour.

5.1.3 Generating multicast traffic

The first choice for generating (and receiving) sample multicast traffic was iperf [28], a client-server performance measurement tool capable of listening and sending to unicast and multicast addresses. When used in server mode with a multicast address, iperf generates an IGMP Membership Report at startup and an IGMP Leave Group message on termination. However, it comes with options and limitations that are specific to the field of network performance evaluation, so a general-purpose tool was preferred.

Netcat [29] is a very popular and flexible utility designed for simplicity and ease of integration in commands and scripts. It allows to read and write files, receive data from standard input, execute commands received from a peer, create simple proxies and port forwarders, and more. Unfortunately, netcat's big limitation with respect to this thesis project is its inability to listen to a multicast address. Despite this, it is still a valid choice for generating multicast traffic.

Finally, socat [30] was chosen as the tool for listening to a multicast address. Socat can be thought of as an augmented version of netcat, as it provides a much richer set of functionalities and configuration options, even down to the socket level. One of these options allows to disable the loopback of multicast traffic over the interface they are sent out, preventing applications from receiving their own outgoing traffic. Socat can also be configured to generate IGMP messages like iperf.

5.2 Implementation

Two major functional blocks were implemented in order to test and validate our design. The first component, consisting of a topology file and a Mininet script, recreates and emulates the sample network described in the previous chapter. The second component is the SDN application proper, running on top of the Ryu framework and its REST OpenFlow API, and is in charge of configuring the switches and enforcing the correct forwarding rules to achieve service chaining. The rest of this section describes the high-level structure of these components.

5.2.1 Mininet script

The topology of the network is stored in a JSON file that contains the list of switches, hosts and transcoders. The JSON format allows portability and easy parsing via Python's JSON package. The network graph is represented as an adjacency list of switches and their remote interface identifier. For each host and transcoder, the attachment switch is specified, along with its IP address and switch port. The file is shown in Listing A.1 and the resulting topology, with IP addresses and switch port numbers, can be seen in Figure 5.1.

The Mininet script parses the topology file and adds each node and link to the emulated network environment. Since links in Mininet are bidirectional by default, it is necessary to keep track of those added so far, otherwise an error occurs when a link is inserted twice. It is worth noting that Mininet makes no distinction between low-capacity hosts, full-capacity hosts or transcoders; this must be handled within the network application. Finally, the script adds the controller, starts the emulation and invokes the CLI. The complete script is shown in Listing A.2.

5.2.2 Network application

The network application is written in Python and uses the OpenFlow REST API provided by Ryu's ofctl_rest.py module to configure flows and groups on the switches.

The application reads the same topology file used by the Mininet script and runs Dijkstra's algorithm to calculate the shortest paths from every switch to all the others, thus obtaining optimal source-specific trees. The set of switch ports along each tree is then retrieved by backward traversal of the tree until either the root, or an already visited switch is encountered. The port sets are stored in two similar data structures,

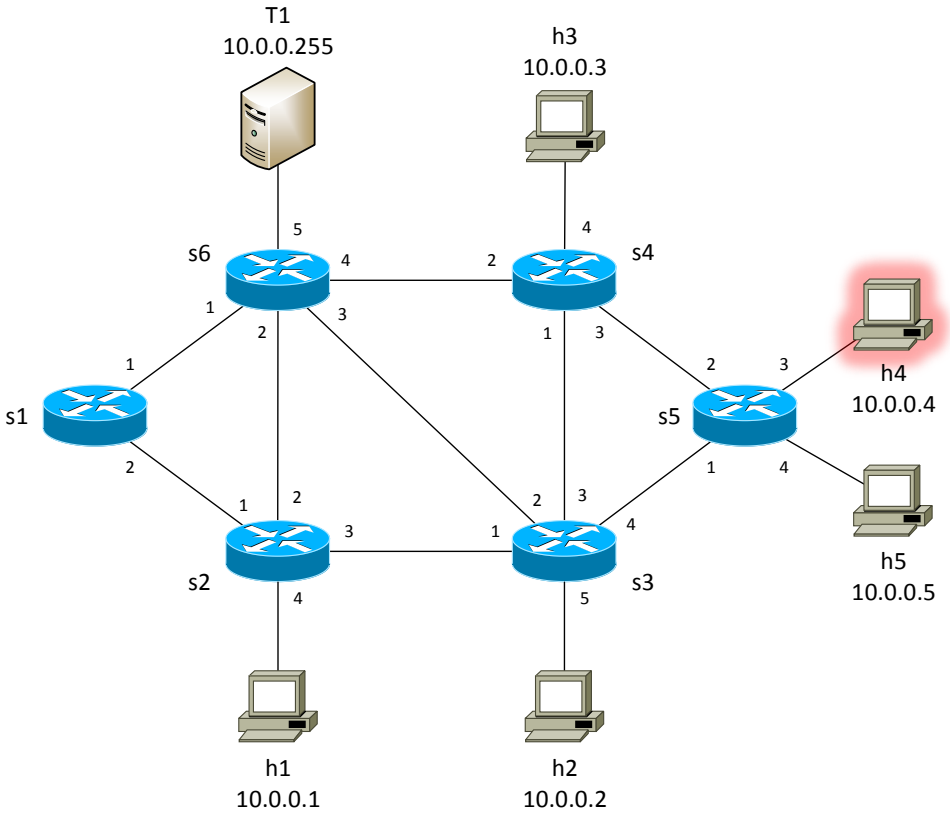


Figure 5.1: Detailed network topology as specified in the topology file

one for normal trees and indexed by source host, the other for low-capacity trees and indexed by transcoder. Group entries and flow rules are derived directly from port sets: on each switch along a tree, a group is created for each set of ports on that tree, and a flow pointing to that group is installed; this may result in several identical groups on the same switch, but it was easier to code and maintain and does not lead to incorrect behaviour. A simple menu is available for debugging purposes, with options to print shortest paths and trees. The full code of the network application is available in Listing B.1.

Switch configuration is done through the classes and methods defined in a small helper file, shown in Listing B.2, which is a very simple wrapper around the flow and group commands of the REST API. To install a flow on a switch, flow attributes (matches, priority and actions) are formatted in JSON and sent as the body of an HTTP POST request to the controller, at the /stats/flowentry/add URI. Similarly, groups are installed through a POST request to the /stats/groupentry/add URI.

In summary, the application performs, in order, the following:

- reads the topology file;
- computes shortest paths;
- converts shortest paths into port sets;
- installs group entries;
- installs flow entries.

5.3 Testing

The testing phase aims at verifying the correctness of the implementation with respect to the goals set at design time. We now show how the tools described earlier in this chapter are used for this purpose.

5.3.1 Emulated network setup

The following command launches the Mininet script that builds the network from the topology file and invokes the CLI:

```
$ sudo python net.py
```

One can verify that the network was created as expected by executing the net command in Mininet:

```
mininet> net

h2 h2-eth0:s3-eth5
h3 h3-eth0:s4-eth4
h1 h1-eth0:s2-eth4
h4 h4-eth0:s5-eth3
h5 h5-eth0:s5-eth4
T1 T1-eth0:s6-eth5
s3 lo:  s3-eth1:s2-eth3 s3-eth2:s6-eth3 s3-eth3:s4-eth1 s3-eth4:s5-
    eth1 s3-eth5:h2-eth0
s2 lo:  s2-eth1:s1-eth2 s2-eth2:s6-eth2 s2-eth3:s3-eth1 s2-eth4:h1-
    eth0
s1 lo:  s1-eth1:s6-eth1 s1-eth2:s2-eth1
s6 lo:  s6-eth1:s1-eth1 s6-eth2:s2-eth2 s6-eth3:s3-eth2 s6-eth4:s4-
    eth2 s6-eth5:T1-eth0
```

```
s5 lo:  s5-eth1:s3-eth4 s5-eth2:s4-eth3 s5-eth3:h4-eth0 s5-eth4:h5-
    eth0
s4 lo:  s4-eth1:s3-eth3 s4-eth2:s6-eth4 s4-eth3:s5-eth2 s4-eth4:h3-
    eth0
c0
```

The output shows the name of each node and the list of interfaces and connected nodes. For example, switch s1 has a loopback interface and two links, one from local port s1-eth1 to remote port s6-eth1 of switch s6, and the other from local port s1-eth2 to remote port s2-eth1 of switch s2. The controller is shown as c0.

5.3.2 Installation of forwarding rules

Before showing how the application was tested, it is important to mention that the correctness of the designed forwarding rules was first verified by statically pushing group and flow entries using the ovs-ofctl utility mentioned in Section 5.1.1. This helped discovering design flaws before coding the SDN application. Here is an example of how entries can be installed with ovs-ofctl from a Linux terminal after launching the Mininet script:

```
$ sudo ovs-ofctl -O OpenFlow13 add-group s2 "group_id=1 type=all
    bucket=output:2 bucket=output:3"

$ sudo ovs-ofctl -O OpenFlow13 add-flow s2 "ip in_port=1 priority=1
    actions=group:1"
```

This example (which is not part of our implementation) replicates incoming IP packets from interface 1 of switch s2 and sends them out of interfaces 2 and 3, demonstrating how multicasting can be achieved with OVS and OpenFlow 1.3.

The Ryu controller and its REST module can be started either before or after launching Mininet, with the following command run from a different terminal:

```
$ ryu-manager ryu/ryu/app/ofctl_rest.py
```

This launches Ryu on the default TCP port 6633 and loads the ofctl_rest.py module, which starts an HTTP server on port 8080 and logs every request to the console.

The next step is to run the network application that dynamically installs groups and flow rules on switches based on the topology file and shortest path calculations:

```
$ python app.py
```

The ovs-ofctl tool can be used once again to verify how the switches have been configured by the application. To retrieve the flow rules from, say, switch s6, the following command sends a flow statistics request to the switch:

```
$ sudo ovs-ofctl -O OpenFlow13 dump-flows s6

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=7.792s, table=0, n_packets=0, n_bytes=0,
     priority=1,ip,in_port=3,nw_src=10.0.0.4,nw_dst=239.192.0.1
     actions=group:4
 cookie=0x0, duration=7.868s, table=0, n_packets=0, n_bytes=0,
     priority=1,ip,in_port=4,nw_src=10.0.0.3,nw_dst=239.192.0.1
     actions=group:2
 cookie=0x0, duration=7.757s, table=0, n_packets=0, n_bytes=0,
     priority=1,ip,in_port=3,nw_src=10.0.0.5,nw_dst=239.192.0.1
     actions=group:5
 cookie=0x0, duration=7.838s, table=0, n_packets=0, n_bytes=0,
     priority=1,ip,in_port=2,nw_src=10.0.0.1,nw_dst=239.192.0.1
     actions=group:3
 cookie=0x0, duration=7.922s, table=0, n_packets=0, n_bytes=0,
     priority=1,ip,in_port=3,nw_src=10.0.0.2,nw_dst=239.192.0.1
     actions=group:1
 cookie=0x0, duration=7.729s, table=0, n_packets=0, n_bytes=0,
     priority=1,ip,in_port=5,nw_dst=239.192.0.1 actions=set_field
     :63->ip_dscp,group:6
```

The reply contains a list of flow entries along with their duration, packet and byte counts, match fields and actions. Recall that s6 is the attachment switch for the transcoder. As can be seen, six flow rules were retrieved. The first five rules match packets from the hosts and forward them along their source-specific trees towards the transcoder and the other hosts. The last rule matches traffic from the transcoder, sets the DSCP field to 63 (Tos value 252), and sends it down the low-capacity tree towards host h4. Group details are retrieved as follows:

```
$ sudo ovs-ofctl -O OpenFlow13 dump-groups s6
```

```
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=6,type=all,bucket=weight:0,actions=output:3
 group_id=4,type=all,bucket=weight:0,actions=output:5
 group_id=1,type=all,bucket=weight:0,actions=output:5
 group_id=5,type=all,bucket=weight:0,actions=output:5
 group_id=2,type=all,bucket=weight:0,actions=output:5
 group_id=3,type=all,bucket=weight:0,actions=output:5
```

In order to show that groups are set up properly and packets are indeed sent out of the correct interfaces, it is necessary to examine the configuration of all the other switches (see Appendix C). Also notice how most groups are identical, as expected. This may be a little inefficient but makes topology updates easier to manage. The configuration of other switches can be verified in a similar way.

5.3.3 Delivery of multicast traffic

The final test consists in generating sample multicast traffic and checking that it truly reaches every host according to design goals and switch configuration. To do so, we need to send commands to hosts and the transcoder through Mininet's CLI. We can open terminal windows (xterm) on hosts as follows:

```
mininet> xterm h1 h2 h3 h4 h5 T1
```

Using the tools described in Section 5.1.3, we want hosts to listen to the selected multicast address and port (239.192.0.1:1234) with the socat command and print what they receive to standard output:

```
# socat UDP4-RECVFROM:1234,ip-add-membership=239.192.0.1:10.0.0.1,
    fork STDOUT
```

This command subscribes the host (h1 in this case) to the multicast address, binds it to the virtual interface with address 10.0.0.1 and forks a new process to print the payload of every received packet to standard output. Similarly, the transcoder pipes two socat commands together so that received traffic is reflected back over the same interface it was received:

```
# socat UDP4-RECVFROM:1234,ip-add-membership=239.192.0.1:10.0.0.255,
    fork STDOUT | socat STDIN UDP4-DATAGRAM:239.192.0.1:1234,ip-
```

```
    multicast-if=10.0.0.255,ip-multicast-loop=0
```

The ip-multicast-loop option is explicitly disabled to prevent outgoing packets from being received again by the transcoder. This is obviously not real transcoding at all, but merely serves the purpose of demonstrating that it is possible to reroute traffic through a middlebox to achieve service chaining. A better way would be to have the transcoder perform some kind of basic compression (e.g. removing random bytes from the payload) by piping a third command between the two socat commands. Unfortunately, for some reason we were not able to pipe three commands together with socat. Finally, we need to pick a host as the sender (h1 in this case), open a new xterm window from Mininet and type the following:

```
# nc -u 239.192.0.1 1234
```

The netcat command shown above should prompt the user for data to be sent to the multicast address on UDP port 1234. However, the program terminates immediately and nothing happens. This is easily fixed by adding a default route to h1's IP configuration and trying again:

```
# ip route add default via 10.0.0.1

# nc -u 239.192.0.1 1234
```

Alternatively, we can use socat again:

```
# socat STDIN UDP4-DATAGRAM:239.192.0.1:1234,ip-multicast-if
    =10.0.0.1,ip-multicast-loop=0
```

We can now type some test data into h1's terminal. Upon pressing enter, the input is sent as payload of a UDP packet to the multicast address and will appear in the xterm windows of the other hosts. In order to check the actual path of the packet through the network we can look at the flow counters on each switch along h1's and T1's trees using the ovs-ofctl utility. The complete flow and group dumps for every switch, retrieved after testing, are shown in Appendix C. Results related to the traffic from h1 are summarized in Table 5.1 and Figure 5.2.

Switch	Match fields				Actions	
	Input port	Source	Destination	ToS	Set field	Output
s2	4	10.0.0.1	239.192.0.1	-	-	2, 3
s3	1	10.0.0.1	239.192.0.1	-	-	3, 4, 5
	2	10.0.0.255	239.192.0.1	252	-	4
s4	1	10.0.0.1	239.192.0.1	-	-	4
s5	1	10.0.0.1	239.192.0.1	-	-	4
	1	10.0.0.255	239.192.0.1	252	-	3
s6	2	10.0.0.1	239.192.0.1	-	-	5
	5	-	239.192.0.1	-	ToS: 252	3

Table 5.1: Summary of forwarding rules for traffic from h1

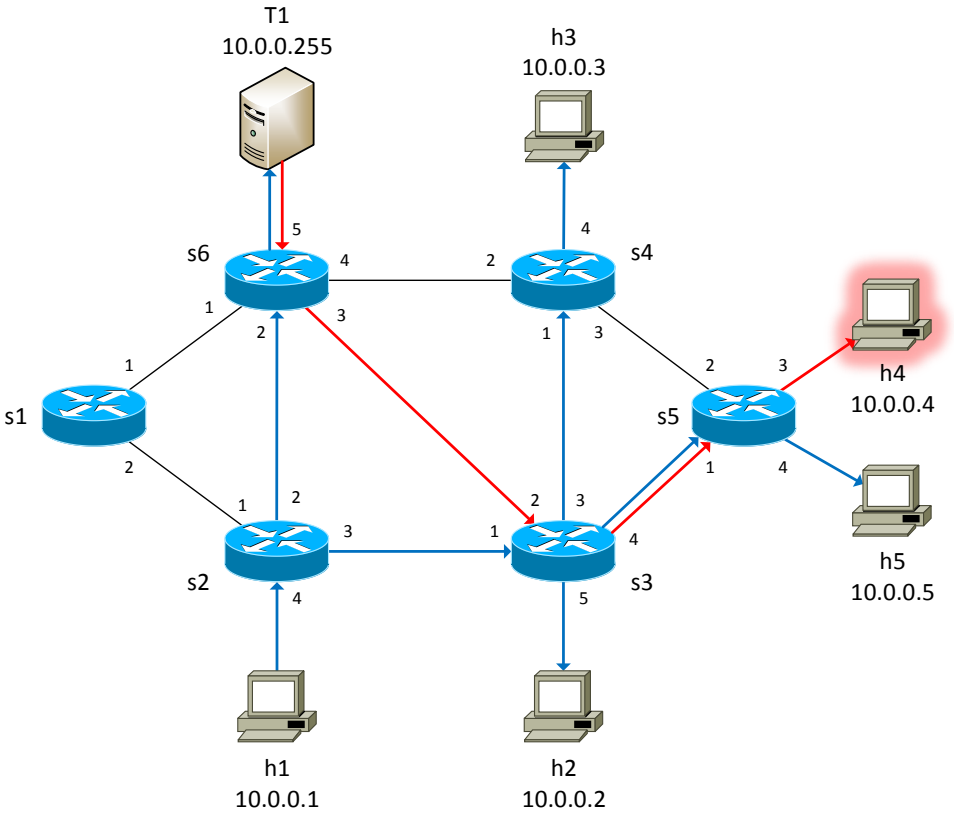


Figure 5.2: Path of traffic from h1 through the network (low-capacity tree in red)

Chapter 6

Discussion

After demonstrating how the original problem can be solved using a combination of multicast and SDN, we need to draw some conclusions and comment on the results. This chapter points out achievements and possible areas of improvement and will hopefully inspire further investigation on the problem.

SDN in general brings in both advantages and drawbacks. On the positive side, the ability to drive networks and deploy services using generic programming tools is perhaps the most promising feature of SDN and provides a high degree of flexibility that is not possible with proprietary and closed-source network architectures. The separation of the control plane from the data plane and the unified southbound interface provided by OpenFlow make network management easier, but at the same time add some complexity in the control and application planes (this is espcially apparent in OpenDaylight). The additional layers of abstraction can make high-performance and scalable application development more difficult. Besides, unlike traditional distributed control, a centralized controller is very intuitively a single point of failure.

Only proactive configuration of switches has been tested, i.e. flows and groups were installed before running the experiments, with the network "at rest". Reactive configuration in response to network events and host activity is another possibility, but requires careful design to ensure acceptable performance, and programming at controller level rather than application level, thus increasing the overall complexity of the solution.

The lack of a real transcoding middlebox made evaluation somewhat difficult and imprecise. For this reason we could not rely on flow byte counters for measuring bandwidth saving, so we resorted to simply counting the number of packets in order to verify that

forwarding was carried out properly. In addition, it was recalled quite late that Mininet supports traffic shaping and virtual link bandwidth limiting, which could be useful to evaluate the usefulness of service chaining by comparing the number of packet drops with and without the transcoder.

We do not believe that SDN is the definitive solution to all networking problems, but it sure has a lot of potential. It appears from this thesis work that SDN is well suited to build simple custom forwarding and service chaining, although its performance in highly dynamic environments with near real-time requirements should be studied and evaluated in much greater detail.

We are conscious of the fact that the application developed in this project has many imperfections and is still far from being a complete solution. We are, however, reasonably satisfied with our results.

6.1 Fulfillment of Objectives

Most of the objectives listed in Section 4.2.1 have been succesfully achieved, as shown in Table 6.1. The only exception is transparency, which stated that the original IP addresses should preferably be maintained after packet processing by the transcoder. Unfortunately, too much time and effort were required in order to address the problem, and this requirement, being not among the primary goals, was eventually dropped. It has to be said that the most difficult part of the project was not the implementation, but rather figuring out the necessary flow rules at design time.

Objective	Fulfilled
Source-specific trees	Yes
Automatic switch configuration	Yes
ToS-marking	Yes
Multicasting	Yes
Service chaining	Yes
Transparency	No
Access control	Yes
Spoofing protection	Yes

Table 6.1: Summary of objectives and their fulfillment

6.2 Middlebox Placement

In our design, the transcoder was placed in an arbitrary position without any special requirement: the main objective was not to find the best location for the middlebox, but rather to correctly setup forwarding rules for service chaining. Now that we have shown that this is possible with SDN and OpenFlow, some words should be spent on how to determine the appropriate location of such middlebox. This problem is known in the literature as the facility location problem. Even without addressing the problem in a complete and formal way (this would likely require a whole book on its own, and is definitely out of scope here) we can still draw some empyrical guidelines.

The number and distribution of low-capacity hosts surely affects transcoder placement. This is also highly dependent on the network topology. In general, the most reasonable solution is to place the controller as close to the low-capacity receivers as possible. However, if all low-capacity hosts reside in the same region, locating the transcoder next to the sources may be more appropriate in order to avoid unnecessary bandwidth occupancy. Other variables influence this choice, for instance the number of available middleboxes and the cost of installation. Another interesting factor is whether the appliance is virtualized or not: if so, it may be possible to move it across the network on demand and leverage the full potential of Network Functions Virtualization. These considerations show how difficult it is to derive a general rule for every case.

Chapter 7

Conclusions

This thesis looked into how custom routing and multicasting are possible with Software-Defined Networking and OpenFlow. Decoupling the control plane from the data plane and unifying the network management interface allows configuration of the network using standard, general-purpose programming tools, and paves the way for a whole new range of services and applications. We have also shown how networks can be designed and evaluated easily with Mininet and Open vSwitch without physical deployment and interconnection of hardware.

Among the possible use cases of multicast and SDN for a research network provider, we focused on videoconferencing. It is common today to access the Internet from a variety of devices and locations, such as smartphones connected over wireless networks with low bandwidth or poor coverage. Our design aimed at providing a flexible solution that allows end-users with limited capabilities to participate in a videoconferencing experience together with other users that do not suffer from such limitations. This was accomplished through network service chaining, in which traffic is re-routed through a transcoding middlebox that delivers a compressed, low-quality multimedia stream only to those struggling hosts.

Following the description of the implementation and testing procedure, possible advantages, drawbacks and middlebox location were discussed. Although SDN can provide the required degree of flexibility, a complete solution needs careful planning, taking into account scalability, availability and security issues.

7.1 Future Work

During the work on this thesis, several interesting ideas emerged but did not make it into the final design due to lack of time. Possible continuations of this work include:

- automatic topology discovery in the controller;
- load balancing between multiple transcoders;
- dynamic optimization of transcoder placement;
- integration with multimedia signaling and group management;
- traffic engineering with load balancing across links;
- dynamic/predictive fault handling with periodic collection of statistics;
- testing on real hardware.

References

[1] M. McBride and H. Lui. Multicast in the Data Center Overview. Internet draft, July 2012. http://tools.ietf.org/html/draft-mcbride-armd-mcast-overview-02.

[2] S. Deering. Host Extensions for IP Multicasting. RFC 1112, August 1989. http://tools.ietf.org/html/rfc1112.

[3] H. Holbrook and B. Cain. Source-Specific Multicast for IP. RFC 4607, August 2006. http://tools.ietf.org/html/rfc4607.

[4] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236, November 1997. http://tools.ietf.org/html/rfc2236.

[5] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376, October 2002. http://tools.ietf.org/html/rfc3376.

[6] T. Pusateri. Distance Vector Multicast Routing Protocol. Internet draft, October 2003. http://tools.ietf.org/html/draft-ietf-idmr-dvmrp-v3-11.

[7] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). RFC 3973, January 2005. http://tools.ietf.org/html/rfc3973.

[8] J. Moy. Multicast Extensions to OSPF. RFC 1584, March 1994. http://tools.ietf.org/html/rfc1584.

[9] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. OSPF for IPv6. RFC 5340, July 2008. http://tools.ietf.org/html/rfc5340.

[10] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 4601, August 2006. http://tools.ietf.org/html/rfc4601.

[11] T. Hardjono and G. Tsudik. IP Multicast Security: Issues and Directions. *Annales Des Télécommunications*, 55(7-8), July-August 2000.

[12] T. Bachert. IPv4 Multicast Security: A Network Perspective. White paper, SANS Institute, 2002. http://www.sans.org/reading-room/whitepapers/networkdevs/ipv4-multicast-security-network-perspective-246.

[13] M. Handley and J. Crowcroft. Internet Multicast Today. *The Internet Protocol Journal*, 2(4), December 1999.

[14] D. Meyer. Administratively Scoped IP Multicast. RFC 2365, July 1998. http://tools.ietf.org/html/rfc2365.

[15] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). RFC 6241, June 2011. http://tools.ietf.org/html/rfc6241.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. White paper, March 2008. http://www.sans.org/reading-room/whitepapers/networkdevs/ipv4-multicast-security-network-perspective-246.

[17] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047, December 2013. http://tools.ietf.org/html/rfc7047.

[18] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. Technical report, October 2009. http://openvswitch.org/papers/hotnets2009.pdf.

[19] T.D. Nadeau and K. Gray. *SDN: Software Defined Networks*. O'Reilly, August 2013.

[20] Open Networking Foundation. OpenFlow Switch Specification, Version 1.3.1 (Wire Protocol 0x04). Technical report, September 2012. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[21] UNINETT. https://www.uninett.no/en.

[22] Mininet Team. Mininet Overview. http://mininet.org/overview/.

[23] Open vSwitch. https://github.com/openvswitch/ovs/blob/master/README.md.

[24] Why Open vSwitch? https://github.com/openvswitch/ovs/blob/master/WHY-OVS.md.

[25] OpenDaylight. http://www.opendaylight.org/.

[26] Ryu SDN Framework Community. Ryu SDN Framework. http://osrg.github.io/ryu/.

[27] Ryu project team. *Ryu SDN Framework*. http://osrg.github.io/ryu-book/en/Ryubook.pdf.

[28] Iperf - The TCP/UDP Bandwidth Measurement Tool. https://iperf.fr/.

[29] G. Giacobbi. The GNU Netcat. http://netcat.sourceforge.net/.

[30] socat - Multipurpose relay. http://www.dest-unreach.org/socat/.

Appendix A

Topology File and Mininet Script

Listing A.1: Topology file: topo1.json

```
{
    "__COMMENT": "Network topology description used by network application and
        Mininet script",

    "__COMMENT": "switches: each switch is a dictionary of adjacent switch name
        : local port switch",
    "__COMMENT": "dpids: mapping from switch name to datapath id (only for app.
        py)",
    "__COMMENT": "hosts: each host is a dictionary of attached switch, switch
        port and host IP address",
    "__COMMENT": "low_hosts: array of low-capacity hosts, each as key in hosts
        dictionary",
    "__COMMENT": "tees: transcoders, same format as hosts",


    "switches": {
        "s1": { "s2": 2, "s6": 1 },
        "s2": { "s1": 1, "s3": 3, "s6": 2 },
        "s3": { "s2": 1, "s4": 3, "s5": 4, "s6": 2},
        "s4": { "s3": 1, "s5": 3, "s6": 2 },
        "s5": { "s3": 1, "s4": 2 },
        "s6": { "s1": 1, "s2": 2, "s3": 3, "s4": 4 }
    },

    "dpids": {
        "s1": 1,
        "s2": 2,
```

```
        "s3": 3,
        "s4": 4,
        "s5": 5,
        "s6": 6
    },

    "hosts": {
        "h1": { "switch": "s2", "port": 4, "ip": "10.0.0.1" },
        "h2": { "switch": "s3", "port": 5, "ip": "10.0.0.2" },
        "h3": { "switch": "s4", "port": 4, "ip": "10.0.0.3" },
        "h4": { "switch": "s5", "port": 3, "ip": "10.0.0.4" },
        "h5": { "switch": "s5", "port": 4, "ip": "10.0.0.5" }
    },

    "low_hosts": [
        "h4"
    ],

    "tees": {
        "T1": { "switch": "s6", "port": 5, "ip": "10.0.0.255" }
    }
}
```

Listing A.2: Mininet script: net.py

```
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.node import OVSSwitch, RemoteController
import json


def net():

    net = Mininet()

    # read topology file
    filejson = open("topo/topo1.json")
    topojson = json.load(filejson)

    # create topology
    link_exists = {}

    for name in topojson['switches']:
        net.addSwitch(name, cls=OVSSwitch, protocols="OpenFlow13")
        link_exists[name] = {}

```

```
    for name in topojson['hosts']:
        net.addHost(name, ip=topojson['hosts'][name]['ip'])

    for name in topojson['tees']:
        net.addHost(name, ip=topojson['tees'][name]['ip'])

    # connect switches
    for swname in topojson['switches']:
        adjsw = topojson['switches'][swname]
        for adjswname in adjsw:
            # links are bidirectional, error if added twice
            if adjswname not in link_exists[swname]:
                local_if = adjsw[adjswname]
                remote_if = topojson['switches'][adjswname][swname]
                net.addLink(swname, adjswname, port1=local_if, port2=remote_if)
                # mark both as created
                link_exists[swname][adjswname] = True
                link_exists[adjswname][swname] = True

    # connect hosts and transcoders to switches
    for name in topojson['hosts']:
        hostdata = topojson['hosts'][name]
        net.addLink(name, hostdata['switch'], port2=hostdata['port'])

    for name in topojson['tees']:
        hostdata = topojson['tees'][name]
        net.addLink(name, hostdata['switch'], port2=hostdata['port'])

    # add controller and start network
    net.addController(controller=RemoteController, port=6633)
    net.start()

    # start CLI
    CLI(net)

    # done
    net.stop()


if __name__ == '__main__':
    net()
```

Appendix B

Network Application Files

Listing B.1: Network application: app.py

```
from ofhelper import FlowEntry, GroupEntry
import json


# allocate variable names (see topology files in common dir for format)
switches = {}   # switches
hosts = {}      # all hosts, including low-capacity hosts but not transcoders
low_hosts = []  # low-capacity hosts
tees = {}       # transcoders
dpids = {}      # datapath id for each switch

# for each source host, store the list of output ports for each switch in tree
# used to build and track group entries
ports = {}
ports_lq = {}   # rooted at T

# shortest paths, from each switch
sp = {}

# different groups may be installed on each switch (one for each source-
    specific
# tree traversing the switch): keep track of the next available group id
gid = {}

# the multicast address reserved to this group
MCAST_ADDR = "239.192.0.1"
DSCP_VALUE = 63
```

```
27
28
29  def load_json_topology (filename):
30
31      global switches
32      global hosts
33      global low_hosts
34      global tees
35      global dpids
36      global gid
37
38      filejson = open(filename)
39      topojson = json.load(filejson)
40
41      switches = topojson['switches']
42      hosts = topojson['hosts']
43      low_hosts = topojson['low_hosts']
44      tees = topojson['tees']
45      dpids = topojson['dpids']
46
47      for sw in switches:
48          gid[sw] = 1
49
50
51  def get_next_gid (sw):
52      g = gid[sw]
53      gid[sw] += 1
54      return g
55
56
57  # Dijkstra's algorithm from switch src
58  def shortest_paths (src):
59
60      dist = {}
61      prev = {}
62
63      tovisit = switches.keys()
64      for node in tovisit:
65          dist[node] = float('inf')
66          prev[node] = None
67      dist[src] = 0
68
69      while len(tovisit) > 0:
70          # extract node u closest to the set of visited nodes
71          tovisit.sort(key = lambda x: dist[x])
72          u = tovisit.pop(0)
```

```
        # for each neighbor v of u still unvisited, update distances
        for v in switches[u]:
            if v in tovisit:
                tmp = dist[u] + 1
                if tmp < dist[v]:
                    dist[v] = tmp
                    prev[v] = u

    return prev


def shortest_paths_all():
    for s in switches:
        sp[s] = shortest_paths(s)


def tree_ports_hq (sh): # source host
    done = set() # switches already part of the tree
    treeports = {}
    src = hosts[sh]['switch'] # source switch
    for dh in hosts: # high-capacity destination hosts
        if dh != sh and dh not in low_hosts:
            dst = hosts[dh]['switch'] # destination switch
            # walk back towards source until root (pre is None)
            # or another switch is found that is already part of the tree
            cur = dst # current switch
            pre = sp[src][cur] # parent of current switch
            while pre is not None and cur not in done:
                port = switches[pre][cur]
                if pre not in treeports:
                    treeports[pre] = set()
                treeports[pre].add(port)
                # next iteration
                done.add(cur) # mark current switch as added to the tree
                cur = pre
                pre = sp[src][cur]
            # add destination host
            if dst not in treeports:
                treeports[dst] = set()
            treeports[dst].add(hosts[dh]['port'])
    for t in tees: # transcoders (also part of multicast group)
        dst = tees[t]['switch'] # destination switch
        # walk back towards source until root (pre is None)
        # or another switch is found that is already part of the tree
        cur = dst # current switch
        pre = sp[src][cur] # parent of current switch
```

```
        while pre is not None and cur not in done:
            port = switches[pre][cur]
            if pre not in treeports:
                treeports[pre] = set()
            treeports[pre].add(port)
            # next iteration
            done.add(cur) # mark current switch as added to the tree
            cur = pre
            pre = sp[src][cur]
        # add destination host
        if dst not in treeports:
            treeports[dst] = set()
        treeports[dst].add(tees[t]['port'])
    return treeports


def tree_ports_hq_all():
    for sh in hosts: # source host
        ports[sh] = tree_ports_hq(sh)


def tree_ports_lq (t): # source transcoder
    done = set() # switches already part of the tree
    treeports = {}
    src = tees[t]['switch'] # source switch
    for dh in low_hosts: # low-capacity destination hosts
        dst = hosts[dh]['switch'] # destination switch
        # walk back towards source until root (pre is None)
        # or another switch is found that is already part of the tree
        cur = dst # current switch
        pre = sp[src][cur] # parent of current switch
        while pre is not None and cur not in done:
            port = switches[pre][cur]
            if pre not in treeports:
                treeports[pre] = set()
            treeports[pre].add(port)
            # next iteration
            done.add(cur) # mark current switch as added to the tree
            cur = pre
            pre = sp[src][cur]
        # add destination host
        if dst not in treeports:
            treeports[dst] = set()
        treeports[dst].add(hosts[dh]['port'])
    return treeports

```

```
165
166 def tree_ports_lq_all():
167     for t in tees:
168         ports_lq[t] = tree_ports_lq(t)
169
170
171 def reverse_path_port (host, switch):
172     root = host['switch'] # root switch of h's tree
173     pre = sp[root][switch] # parent switch of current switch
174     if pre is None: # current switch is root switch
175         return host['port'] # local port towards host
176     else:
177         return switches[switch][pre] # local port towards parent switch
178
179
180 def install_hq_flows():
181     for h in ports: # for each high-capacity source host
182         for sw in ports[h]: # for each switch in the tree
183             # group entry
184             newgid = get_next_gid(sw)
185             g = GroupEntry(dpids[sw], newgid, "ALL")
186             i = 0
187             for p in ports[h][sw]: # for each switch port in the tree
188                 g.addBucket()
189                 g.addAction(i, "OUTPUT", port=p)
190                 i += 1
191             g.install()
192             # flow entry (also match on in_port for reverse path check)
193             f = FlowEntry(dpids[sw])
194             f.addMatch("in_port", reverse_path_port(hosts[h],sw))
195             f.addMatch("dl_type", 0x800)
196             f.addMatch("nw_src", hosts[h]['ip'])
197             f.addMatch("nw_dst", MCAST_ADDR)
198             f.addAction("GROUP", group_id=newgid)
199             f.install()
200
201
202 def install_lq_flows():
203     for t in ports_lq: # for each transcoder as source
204         for sw in ports_lq[t]: # for each switch in the tree
205             # group entry
206             newgid = get_next_gid(sw)
207             g = GroupEntry(dpids[sw], newgid, "ALL")
208             i = 0
209             for p in ports_lq[t][sw]: # for each switch port in the tree
210                 g.addBucket()
```

```
211                 g.addAction(i, "OUTPUT", port=p)
212                 i += 1
213             g.install()
214             # flow entry (also match on in_port for reverse path check)
215             # do not install on transcoder switch, tos is not set by T
216             if not sw == tees[t]['switch']:
217                 f = FlowEntry(dpids[sw])
218                 f.addMatch("in_port", reverse_path_port(tees[t],sw))
219                 f.addMatch("dl_type", 0x800)
220                 f.addMatch("ip_dscp", DSCP_VALUE)
221                 f.addMatch("nw_src", tees[t]['ip'])
222                 f.addMatch("nw_dst", MCAST_ADDR)
223                 f.addAction("GROUP", group_id=newgid)
224                 f.install()
225         # set ip_dscp when coming from T
226         # the last group added to T's switch refers to the low-capacity tree
227         tsw = tees[t]['switch']
228         lastgid = gid[tsw]-1
229         # flow entry (match on in_port, not nw_src, because original IP address
230         # should be kept)
231         f = FlowEntry(dpids[tsw])
232         f.addMatch("in_port", tees[t]['port'])
233         f.addMatch("dl_type", 0x800)
234         f.addMatch("nw_dst", MCAST_ADDR)
235         f.addAction("SET_FIELD", field="ip_dscp", value=DSCP_VALUE)
236         f.addAction("GROUP", group_id=lastgid)
237         f.install()
238 
239 
240 def dump_sp():
241     for s in sp:
242         print "sp[%s]:" % (s, sp[s])
243     print #newline
244 
245 
246 def dump_ss_trees():
247     for sh in hosts: # source host
248         src = hosts[sh]['switch'] # source switch
249         print "source: %s (%s)" % (sh,src)
250         for dh in hosts: # destination hosts
251             if dh != sh:
252                 dst = hosts[dh]['switch'] # destination switch
253                 print "  dest: %s (%s)" % (dh,dst)
254                 if dh not in low_hosts:
255                     print "    pre[%s]=%s, port=%d" % (dh,dst,hosts[dh]['port
                            '])
```

```
256                 # walk back until root (pre is None)
257                 cur = dst # current switch
258                 pre = sp[src][cur] # parent of current switch
259                 while pre is not None:
260                     port = switches[pre][cur]
261                     print "    pre[%s]=%s, port=%d" % (cur,pre,port)
262                     cur = pre
263                     pre = sp[src][cur]
264 
265         for t in tees: # transcoders (also part of multicast group)
266             dst = tees[t]['switch'] # destination switch
267             print "  dest: %s (%s)" % (t,dst)
268             # walk back towards source until root (pre is None)
269             cur = dst # current switch
270             pre = sp[src][cur] # parent of current switch
271             while pre is not None:
272                 port = switches[pre][cur]
273                 print "    pre[%s]=%s, port=%d" % (cur,pre,port)
274                 cur = pre
275                 pre = sp[src][cur]
276 
277         portbuf = "ports:"
278         for sw in ports[sh]:
279             for port in ports[sh][sw]:
280                 portbuf += " %s-eth%d" % (sw,port)
281         print portbuf
282         print #newline
283 
284 
285 def dump_low_trees():
286     for t in tees: # source transcoder
287         src = tees[t]['switch'] # source switch
288         print "source: %s (%s)" % (t,src)
289         for dh in low_hosts: # destination low-capacity hosts
290             dst = hosts[dh]['switch'] # destination switch
291             print "  dest: %s (%s)" % (dh,dst)
292             print "    pre[%s]=%s, port=%d" % (dh,dst,hosts[dh]['port'])
293             # walk back until root (pre is None)
294             cur = dst # current switch
295             pre = sp[src][cur] # parent of current switch
296             while pre is not None:
297                 port = switches[pre][cur]
298                 print "    pre[%s]=%s, port=%d" % (cur,pre,port)
299                 cur = pre
300                 pre = sp[src][cur]
301         portbuf = "ports:"
```

```
302         for sw in ports_lq[t]:
303             for port in ports_lq[t][sw]:
304                 portbuf += " %s-eth%d" % (sw,port)
305         print portbuf
306         print #newline
307 
308 
309 def menu():
310 
311     options = [
312         {'str': "Quit", 'action': None},
313         {'str': "Dump shortest paths", 'action': dump_sp},
314         {'str': "Dump source-specific trees", 'action': dump_ss_trees},
315         {'str': "Dump low-capacity trees", 'action': dump_low_trees}
316     ]
317 
318     while True: # until quit
319         while True: # while bad input
320 
321             for i in range(len(options)):
322                 print "  %d - %s" % (i, options[i]['str'])
323             print #newline
324 
325             try:
326                 choice = int(raw_input("Choose an option: "))
327                 if choice < 0 or choice >= len(options):
328                     raise ValueError
329                 break
330             except ValueError:
331                 print "Invalid choice: enter a number between 0 and %d" \
332                   % (len(options)-1)
333             except (EOFError, KeyboardInterrupt):
334                 print #newline
335                 choice = 0
336                 break
337 
338         print #newline
339 
340         if choice == 0: # quit
341             break
342 
343         if not options[choice]['action'] is None:
344             options[choice]['action']()
345 
346 
347 if __name__ == "__main__":
```

```
348     print "** Loading topology **"
349     load_json_topology("../topo/topo1.json")
350     print "** Generating shortest paths (source-specific trees) **"
351     shortest_paths_all()
352     print "** Generating port sets for high-capacity trees **"
353     tree_ports_hq_all()
354     print "** Generating port sets for low-capacity trees **"
355     tree_ports_lq_all()
356     print "** Installing flows for high-quality traffic **"
357     install_hq_flows()
358     print "** Installing flows for low-quality traffic **"
359     install_lq_flows()
360     menu()
```

Listing B.2: Helper file: ofhelper.py

```
1  import json
2  import httplib
3
4
5  class FlowEntry():
6
7      def __init__ (self, dpid, priority=1):
8          self._dpid = dpid
9          self._priority = priority
10         self._matches = {}
11         self._actions = []
12
13     def addMatch (self, field, value):
14         self._matches[field] = value
15
16     def addAction (self, action, **params):
17         action = {'type': action}
18         for key in params:
19             action[key] = params[key]
20         self._actions.append(action)
21
22     def install (self):
23         body = self._make_request_body()
24         res = self._send_request("POST", "/stats/flowentry/add", body)
25
26     def delete (self):
27         body = self._make_request_body()
28         res = self._send_request("POST", "/stats/flowentry/delete", body)
29
30     def _make_request_body (self):
```

```
        obj = {}
        obj['dpid'] = self._dpid
        obj['priority'] = self._priority
        obj['match'] = self._matches
        obj['actions'] = self._actions
        return json.dumps(obj)

    def _send_request (self, method, url, body):
        conn = httplib.HTTPConnection("127.0.0.1", 8080)
        conn.request(method, url, body)
        res = conn.getresponse()
        return res


class GroupEntry():

    def __init__ (self, dpid, grpid, grptype):
        self._dpid = dpid
        self._grpid = grpid
        self._type = grptype
        self._buckets = []

    def addBucket (self, weight=0):
        self._buckets.append({'weight': weight, 'actions': []})

    def addAction (self, bucket, action, **params):
        if not bucket < len(self._buckets):
            print "** Bucket %d does not exist **" % bucket
            return
        action = {'type': action}
        for key in params:
            action[key] = params[key]
        self._buckets[bucket]['actions'].append(action)

    def install (self):
        body = self._make_request_body()
        res = self._send_request("POST", "/stats/groupentry/add", body)

    def delete (self):
        body = self._make_request_body()
        res = self._send_request("POST", "/stats/groupentry/delete", body)

    def _make_request_body (self):
        obj = {}
        obj['dpid'] = self._dpid
        obj['group_id'] = self._grpid
```

```
77          obj['type'] = self._type
78          obj['buckets'] = self._buckets
79          return json.dumps(obj)
80
81      def _send_request (self, method, url, body):
82          conn = httplib.HTTPConnection("127.0.0.1", 8080)
83          conn.request(method, url, body)
84          res = conn.getresponse()
85          return res
```

Appendix C

Flow and Group Table Entries after Testing

Listing C.1: Flows and groups on switch s1

```
$ sudo ovs-ofctl -O openflow13 dump-flows s1 --sort=nw_src

OFPST_FLOW reply (OF1.3) (xid=0x2):

$ sudo ovs-ofctl -O openflow13 dump-groups s1

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
```

Listing C.2: Flows and groups on switch s2

```
$ sudo ovs-ofctl -O openflow13 dump-flows s2 --sort=nw_src

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=438.477s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=4,nw_src=10.0.0.1,nw_dst=239.192.0.1 actions=group:3
 cookie=0x0, duration=438.522s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.2,nw_dst=239.192.0.1 actions=group:1
 cookie=0x0, duration=438.498s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.3,nw_dst=239.192.0.1 actions=group:2
 cookie=0x0, duration=438.455s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.4,nw_dst=239.192.0.1 actions=group:4
 cookie=0x0, duration=438.430s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.5,nw_dst=239.192.0.1 actions=group:5
```

```
$ sudo ovs-ofctl -O openflow13 dump-groups s2

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=4,type=all,bucket=weight:0,actions=output:4
 group_id=1,type=all,bucket=weight:0,actions=output:4
 group_id=5,type=all,bucket=weight:0,actions=output:4
 group_id=2,type=all,bucket=weight:0,actions=output:4
 group_id=3,type=all,bucket=weight:0,actions=output:2,bucket=weight:0,actions=
     output:3
```

Listing C.3: Flows and groups on switch s3

```
$ sudo ovs-ofctl -O openflow13 dump-flows s3 --sort=nw_src

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=460.727s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=1,nw_src=10.0.0.1,nw_dst=239.192.0.1 actions=group:3
 cookie=0x0, duration=460.777s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=5,nw_src=10.0.0.2,nw_dst=239.192.0.1 actions=group:1
 cookie=0x0, duration=460.706s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=4,nw_src=10.0.0.4,nw_dst=239.192.0.1 actions=group:4
 cookie=0x0, duration=460.749s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.3,nw_dst=239.192.0.1 actions=group:2
 cookie=0x0, duration=460.683s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=4,nw_src=10.0.0.5,nw_dst=239.192.0.1 actions=group:5
 cookie=0x0, duration=460.655s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=2,nw_src=10.0.0.255,nw_dst=239.192.0.1,nw_tos=252 actions=group:6

$ sudo ovs-ofctl -O openflow13 dump-groups s3

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=6,type=all,bucket=weight:0,actions=output:4
 group_id=4,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:2,bucket=weight:0,actions=output:5
 group_id=1,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:2,bucket=weight:0,actions=output:3,bucket=weight:0,actions=output:4
 group_id=5,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:2,bucket=weight:0,actions=output:5
 group_id=2,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:5
 group_id=3,type=all,bucket=weight:0,actions=output:3,bucket=weight:0,actions=
     output:4,bucket=weight:0,actions=output:5
```

Listing C.4: Flows and groups on switch s4

```
$ sudo ovs-ofctl -O openflow13 dump-flows s4 --sort=nw_src

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=481.078s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=1,nw_src=10.0.0.1,nw_dst=239.192.0.1 actions=group:3
 cookie=0x0, duration=481.122s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=1,nw_src=10.0.0.2,nw_dst=239.192.0.1 actions=group:1
 cookie=0x0, duration=481.099s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=4,nw_src=10.0.0.3,nw_dst=239.192.0.1 actions=group:2
 cookie=0x0, duration=481.056s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.4,nw_dst=239.192.0.1 actions=group:4
 cookie=0x0, duration=481.028s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.5,nw_dst=239.192.0.1 actions=group:5

$ sudo ovs-ofctl -O openflow13 dump-groups s4

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=4,type=all,bucket=weight:0,actions=output:4
 group_id=1,type=all,bucket=weight:0,actions=output:4
 group_id=5,type=all,bucket=weight:0,actions=output:4
 group_id=2,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:2,bucket=weight:0,actions=output:3
 group_id=3,type=all,bucket=weight:0,actions=output:4
```

Listing C.5: Flows and groups on switch s5

```
$ sudo ovs-ofctl -O openflow13 dump-flows s5 --sort=nw_src

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=495.813s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=1,nw_src=10.0.0.1,nw_dst=239.192.0.1 actions=group:3
 cookie=0x0, duration=495.858s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=1,nw_src=10.0.0.2,nw_dst=239.192.0.1 actions=group:1
 cookie=0x0, duration=495.835s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=2,nw_src=10.0.0.3,nw_dst=239.192.0.1 actions=group:2
 cookie=0x0, duration=495.793s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.4,nw_dst=239.192.0.1 actions=group:4
 cookie=0x0, duration=495.765s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=4,nw_src=10.0.0.5,nw_dst=239.192.0.1 actions=group:5
 cookie=0x0, duration=495.749s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=1,nw_src=10.0.0.255,nw_dst=239.192.0.1,nw_tos=252 actions=group:6

$ sudo ovs-ofctl -O openflow13 dump-groups s5

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=6,type=all,bucket=weight:0,actions=output:3
```

```
 group_id=4,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:2,bucket=weight:0,actions=output:4
 group_id=1,type=all,bucket=weight:0,actions=output:4
 group_id=5,type=all,bucket=weight:0,actions=output:1,bucket=weight:0,actions=
     output:2
 group_id=2,type=all,bucket=weight:0,actions=output:4
 group_id=3,type=all,bucket=weight:0,actions=output:4
```

Listing C.6: Flows and groups on switch s6

```
$ sudo ovs-ofctl -O openflow13 dump-flows s6 --sort=nw_src

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=517.216s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=2,nw_src=10.0.0.1,nw_dst=239.192.0.1 actions=group:3
 cookie=0x0, duration=517.260s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.2,nw_dst=239.192.0.1 actions=group:1
 cookie=0x0, duration=517.237s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=4,nw_src=10.0.0.3,nw_dst=239.192.0.1 actions=group:2
 cookie=0x0, duration=517.194s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.4,nw_dst=239.192.0.1 actions=group:4
 cookie=0x0, duration=517.169s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
     in_port=3,nw_src=10.0.0.5,nw_dst=239.192.0.1 actions=group:5
 cookie=0x0, duration=517.140s, table=0, n_packets=1, n_bytes=49, priority=1,ip
     ,in_port=5,nw_dst=239.192.0.1 actions=set_field:63->ip_dscp,group:6

$ sudo ovs-ofctl -O openflow13 dump-groups s6

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=6,type=all,bucket=weight:0,actions=output:3
 group_id=4,type=all,bucket=weight:0,actions=output:5
 group_id=1,type=all,bucket=weight:0,actions=output:5
 group_id=5,type=all,bucket=weight:0,actions=output:5
 group_id=2,type=all,bucket=weight:0,actions=output:5
 group_id=3,type=all,bucket=weight:0,actions=output:5
```