INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Deployment of MEADcast in Stub Software-Defined Networks

Duc Minh Nguyen

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Deployment of MEADcast in Stub Software-Defined Networks

Duc Minh Nguyen

Aufgabensteller:	PD Dr. rer. nat. Vitalian Danciu
Betreuer:	Cuong Ngoc Tran
Abgabetermin:	31. März 2019

Zusammenfassung

Multicasting ist eine Form der simultanen Datenübertragung von einem Sender zu einer Gruppe von Empfängern. Während Unicast ein Datenpaket an jeden Empfänger senden muss, reduziert Multicast die benötigte Anzahl von Datenpaketen vom Sender zu den Empfängern. Dies is besonders vorteilhaft für große Rechnernetze, in denen ein oder mehrere Sender gleichzeitig Daten an viele Empfänger schickt. Video-Streaming-Dienste können beispielsweise von Multicast profitieren. Obwohl Multicast in der Theorie sehr gut funktioniert, ist zusätzliche Arbeit nötig um Multicast zu benutzen. Es bedarf beispielsweise die Einrichtung von Multicast-Routern durch Netzwerkadministratoren oder die Installation von zusätzlichen Multicast-Applikationen durch den Benutzer. Dies führt zu mehr Komplexität und Aufwand und lohnt möglicherweise den Arbeitsaufwand nicht.

Privacy-Preserving Multicast to Explicit Agnostic Destinations protocol (MEADcast) - bietet eine Sender-zentrierte Multicast Lösung. Der Sender besitzt alle nötigen Informationen, wie die Netzwerktopologie, die Anzahl der Router im Netz die MEADcast unterstützen, und die Anzahl der Empfänger. Basierend auf diesen Informationen entscheidet der Sender, ob die Daten über MEADcast oder Unicast übertragen werden.

Das Ziel dieser Arbeit ist es, MEADcast in einem Software-Defined network (SDN) zu implementieren. Das SDN wird mit Hilfe von OpenFlow eingerichtet. Das MEADcast-Protokoll wird in Kombination mit Internet Protocol version 6 (IPv6) und Raw Sockets implementiert. Raw Sockets ermöglichen das Erstellen von eigenen packet headers (zu deutsch: Kopfdatenbereichen) und ist notwendig um MEADcast in IPv6-Pakete einzubinden. Die Implementierung wird anschließend in verschiedenen Netzen getestet.

Anstatt MEADcast jedoch in einem Netzwerk-Simulator wie ns-2 zu testen, wird eine realistischere Testumgebung benutzt. Die Implementierung wird durch verschiedene Tests evaluiert, in denen MEADcast in SDN und Unicast miteinander vergleichen werden.

Abstract

Multicasting is a transmission method for delivering data to a group of destination devices simultaneously. It makes bandwidth usage more efficient by sending the data to every interested participant in the multicast group in a single transmission. This is especially beneficial in big networks involving a 1:n or n:m communication pattern like video streaming or videoconferencing where one or many senders can send data to all the participants. While it works great once set up, it still requires a lot of network management to get to that point. It might for example demand setting up multicast routers or require the configuration of multicast applications by the end user. This might add more complexity and work to the network operators and users, which they might not deem worthy of the effort.

The Privacy-Preserving Multicast to Explicit Agnostic Destinations protocol (MEADcast) offers a sender-centered multicast solution. It achieves that by having the sender be the one to have all the necessary information about the network topology, the available MEADcast support of routers in that topology and all intended receivers. The sender decides based on that information if the data will be transmitted through the MEADcast protocol or through unicast.

The objective of this thesis is to implement and deploy MEADcast in combination with Software-Defined Networking (SDN). The MEADcast protocol is implemented with Internet Protocol version 6 (IPv6) and OpenFLow protocol as an enabler of SDN. Additionally raw sockets were chosen instead of datagram sockets or stream sockets to allow access to every section of the data packet to be sent and received. This is required to implement MEADcast. The implementation is then deployed on a network and tested.

Instead of simulating the deployment on a network simulator like ns-2, which doesn't provide a projection of a realistic environment, the implementation is deployed on a more realistic testbed. The implementation is evaluated through various experiments comparing traditional unicast and MEADcast in a SDN.

Contents

1. Introduction

As more and more people get connected to the internet and use services like real-time multimedia streaming, the service provider inevitably has to find a way to meet the demands. In a unicast streaming scenario the service provider would need to serve all the connected users individually, which in turn requires more work, processing, network capabilities and ultimately more expenses. Multicast has the possibility to save a lot of bandwidth and strain on the network by condensing identical IP packets meant for different targets to a single IP packet. That IP packet then gets handled by a multicast capable router, duplicated and distributed among the intended receivers [Dee89].

One of the problems however is that most of todays network infrastructure is built upon hardware and its own highly specialized firmware. Implementing even old technologies like multicast on the network would require the change of that hardware and the software that runs it. It is not surprising that multicast and other efficient technologies haven't been utilized more in every aspect of networking, given how large and diverse the amount of existing hardware is. All this makes our legacy networks very complex, static and hard to manage [ONF12].

The introduction of Software-defined networking (SDN) makes a network more flexible by seperating the network control plane and data planes. By moving the entire network control logic into software it becomes possible to only have switches present that contain nothing more but forwarding tables. The entire logic would be handled by a controller which is decoupled from the hardware. This enables the entire network to be more programmable [FRZ14]. SDN facilitates a more dynamic approach in how network applications and new technologies are introduced to the network. It becomes possible to push new applications and protocols to a large amount of switching devices without having to know each and every technical detail about each specific switch.

SDN makes it easier in theory to implement multicast in networks. A network that has its network traffic handled by a seperate control plane is not limited by vendor-specific devices. It enables the network control to become programmable and easier for multicast to be deployed.

1.1. Challenges of multicast deployment

There are several problems with traditional multicast protocols. The network management being required to authorize the multicast sessions and possibly having to set up multicast routers is one of them. Another one is that users may be required to install additional applications to take part of multcast. There are also privacy concerns about existing multicast protocols sending partial address lists of group members to each receiver[TD18]. Another severe problem is that the network as a whole has to support multicast. If the network doesn't support multicast or the multicast network happens to fail there is no fallback to default unicast. The thesis focuses on Multicast to Explicit Agnostic Destinations (MEADcast) [TD18] in a stub SDN as a proposed solution to privacy issues and the varying support of multicast in individual networks.

Today's internet can be described as a collection of interconnected routing domains consisting of switches, routers and hosts alike. Each routing domain is under a single administration and every routing domain is either a *transit* domain or a *stub* domain. A *stub* domain can be described as a routing domain not capable of transmitting traffic, that did not originate or terminate in the domain [CDZ97]. *Transit* domains on the other hand are capable of doing that. They are responsible for interconnecting stub domains efficiently. The problem this creates for this thesis is the added time complexity. Not only does the MEADcast router acting as the border router need to be aware of other connected MEADcast routers, it also needs to be able to perform all the necessary tasks required for both *stub* and *transit* domains. This is the reason why this thesis only focuses on the deployment of MEADcast in stub SDN.

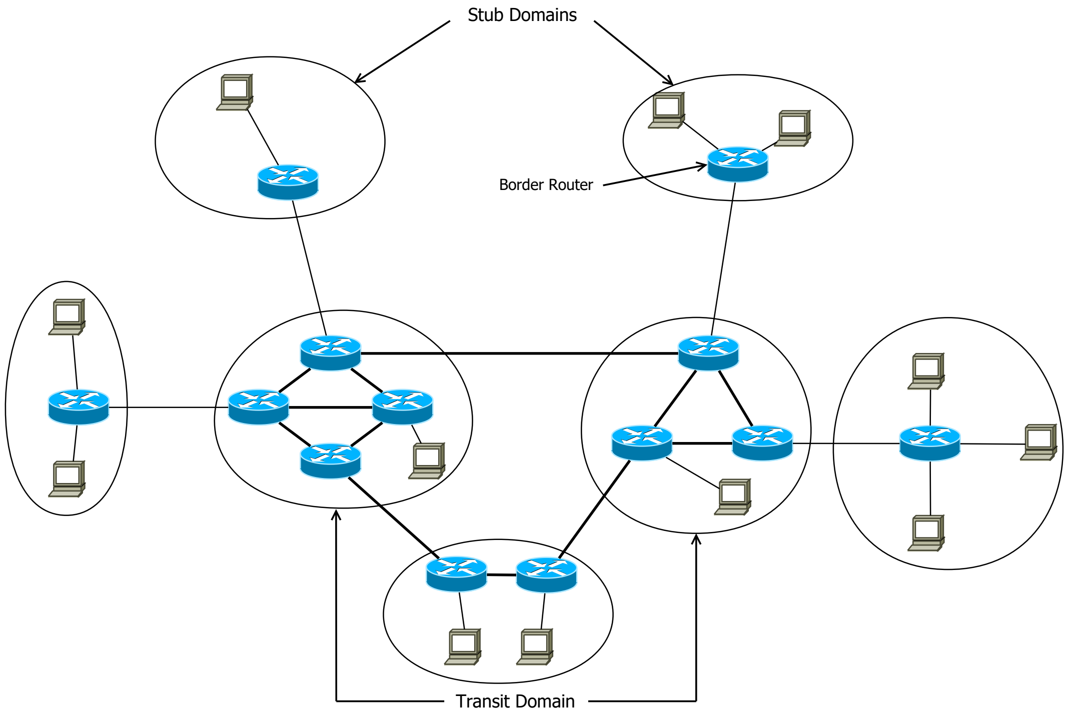


Figure 1.1.: Example of the Internet domain structure [CDZ97]

1.2. Goal and Contributions

The goal of this thesis is to implement and deploy of MEADcast in a stub SDN and evaluate the performance of MEADcast compared to traditional unicast. MEADcast enables sender-based multicast of IPv6 over the Internet. Advantages of this protocol are the protection of its recipients anonymity and the possibility to be used on networks with varying amount of MEADcast routers.

The thesis presents the feasibility of implementing MEADcast on OpenFlow SDN and the problems and limitations coming with it. It also contains several tests about how MEADcast performs over traditional unicast in different scenarios, how their network traffic compares and if the used OpenFlow version can improve multicast performance or not.

In this thesis, the following contributions are made:

- Implementation of MEADcast in an OpenFlow controller.
- Implementation of MEADcast as a traffic generator on the sender side.
- Deployment of MEADcast in SDN and evaluation of the implementation.
- Comparison between the performance of MEADcast in SDN and unicast in regards to transmission speed and total traffic volume.

The findings show that MEADcast in OpenFlow SDN is possible, but very limited in practicality and reliability. It shows that depending on the scenario, using MEADcast in SDN leads to a reduction in total traffic volume and and in increase in transmission speed in comparison to unicast.

1.3. Approach

Different network scenarios are created to test and evaluate the protocol. Every network scenario consists of different virtual machines that act as the controller, the switches or as hosts. We use the RYU SDN Framework [1] as our base for the controller, which handles the control plane. The data plane consists of machines that simulate switches and have Open vSwitch[2] as their switch implementation. MEADcast is both implemented on the controller and on the sender. The Network is generated with a traffic generator that sends both MEADcast packets and unicast packets. That traffic generator simulates the traffic generated by the sender. The traffic and other important performance indicators like transmission speed or individual packet sizes are measured on the network with wireshark[3] and results are compared.

1.4. Structure

The thesis starts with a short introduction. Background and related works are the second Chapter, where the motivation and basics behind Multicast, SDN and MEADcast are explained. It also features a small selection of related work that cover the same topic.

Chapter Three goes more into detail on how the MEADcast protocol was implemented.

Chapter Four describes the experiments and the tested topologies.

Results and evaluation will then be covered in Chapter Five and the thesis ends with a conclusion and future work chapter.

[1]https://osrg.github.io/ryu/
[2]https://openvswitch.org/
[3]https://www.wireshark.org/

2. Background and Related Work

MEADcast is a relatively new multicast protocol. Multicast as one of the fundamentals behind MEADcast however is not. There have been several protocols in the past that have proposed a solution to the problems that arise from using multicast. Explicit Multicast (Xcast) [OFI$^+$07] is one of them and is the protocol that MEADcast is roughly based on. It will be introduced in this chapter. Additionally, Application Layer Multicast as a different approach to multicast and is also briefly introduced.

The thesis has the goal of implementing MEADcast in a Software-Defined Network and hence it is required to include the principles behind the MEADcast protocol itself and SDN.

2.1. Software-Defined Networks

Since MEADcast is implemented in SDN, it is necessary to introduce SDN related information.

2.1.1. SDN Basics

Software-Defined Networking (SDN) is a new approach to combat the disadvantages of the static architecture of legacy networks [KRV$^+$15]. It aims to control, change and manage network behavior dynamically through open interfaces [HPD$^+$15]. Legacy networks rely on very specific network devices that differ depending on the vendor. To implement a network-wide policy for example a network operator may need to change the configuration of multiple network devices, with each network device having their own vendor-specific instructions. This may take a lot of time and resources and only gets worse as current networks grow in size and the requirements of networks change. Automatic reconfiguration depending on dynamic network requirements is virtually non-existent in legacy networks [KRV$^+$15].

The network devices themselves are very inflexible too, since the control plane and the data plane are tightly integrated into every device. This means that updates or innovations to the current architecture may take a very long time or may not even be feasible at all [KRV$^+$15].

Figure 2.1 shows the three planes of functionality found in networks today: The Management plane provides software services for network operators to operate the network device. Example for such tools can be monitoring, configuration or management services. The Control plane decides where the traffic is sent. It decides the entries of the forwarding table used by the data plane. The Data plane, sometimes called the forwarding plane, is responsible for forwarding traffic to the next hop according to the decisions made by the control plane.

Those network devices that are based on this structure all have their own control and data plane and are autonomous from other network devices, which leads to the static and inflexible status quo of networks that is present today.

Figure 2.2 shows the SDN network architecture. SDN separates the control plane from the data plane and enables the abstraction of the network through a controller that is

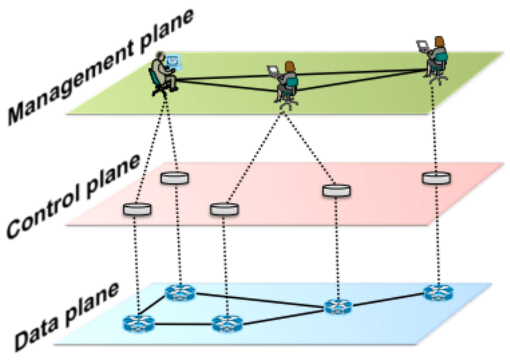


Figure 2.1.: Traditional network architecture [KRV+15].

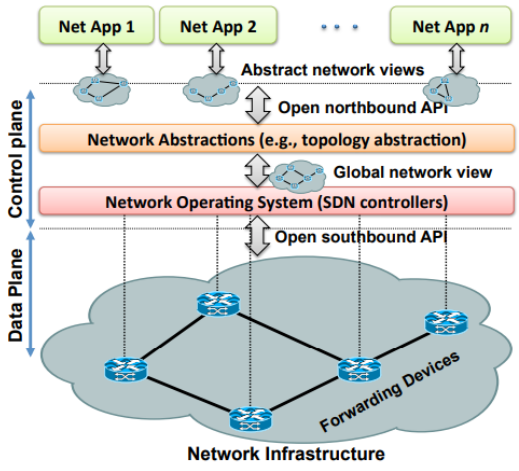


Figure 2.2.: SDN network architecture [KRV+15].

responsible for multiple devices. Those devices get reduced to simple forwarding devices that only forward traffic based on the flow table that has been installed on them by the controller. Each individual flow in that flow table contains packet match fields, flow priority, actions on how to process the packet and other information. Every incoming packet gets matched with the flows and dealt accordingly, unmatched packets may get forwarded to the controller. The Application layer contains traditional network services and applications like load balancers or firewalls. Instead of using low-level and sometimes very vendor-specific commands to configure the network devices the application communicates with the controller to configure the data plane behavior [HPD+15] [Fun12].

2.1.2. SDN Controller - RYU

The control plane that has been traditionally integrated in network devices gets seperated and becomes its own entity. The SDN controller lies between network devices and the application layer and communication between those two occurs through the SDN controller. The communication between the SDN controller and applications like load balancers happens via northbound interfaces, while the communication between SDN controller and the devices happen through southbound protocols (see figure 2.2). One major advantage of using a centralized SDN controller is that the controller has a global network view over the entire network and is aware of all available paths. Another one is that the controller itself can be physically detached from the network and still be able to manage the network functionality

Notable SDN Controllers include NOX [GKP+08], POX and OpenDaylight [MVTG14]. This thesis however focuses solely on RYU.

RYU is a component-based SDN framework and aims to be both agile and flexible. Every application consists of components and RYU provides several components and libraries already [1].

2.1.3. Southbound API - OpenFlow

Southbound APIs enable the SDN Controller in the control plane to efficiently configure the switches in the data plane according to real-time demands and requirements. There are a variety of southbound API, including Network Configuration Protocol (NetConf) [Enn06] and Lisp [RNPCE+15], that enable communication between the controller and the switches.

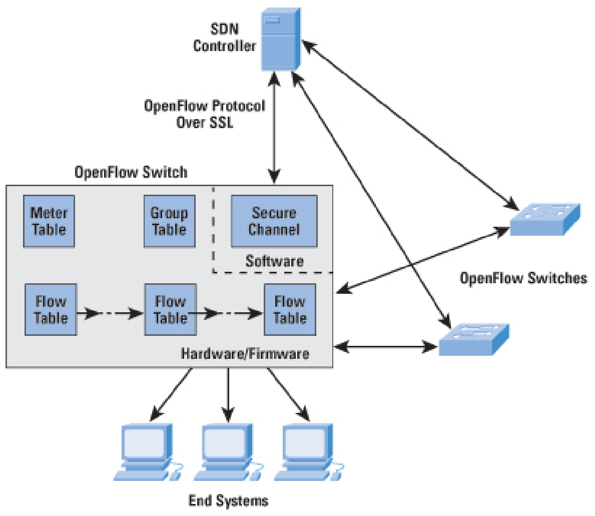


Figure 2.3.: OpenFlow communication between the controller and OpenFlow switch [Cis].

OpenFlow was initially designed to provide a way to run experimental protocols in production networks [HPD+15]. It became synonymous with SDN itself however and has become one, if not the most used southbound application program interface (API) for communication between a logically centralized controller with switches (see Figure 2.3). It also provides a logical structure of the network switch functions[Lim12].

SDN controllers communicate with OpenFlow switches through OpenFlow and ideally via Secure Sockets Layer (SSL). The communication is used for management activities, like installing flows on the flow tables located on the switches. Each OpenFlow switch can maintain one or multiple flow tables. The OpenFlow switch matches those flow tables against incoming packets to determine further procedure[Cis] [Fou12].

Each flow table is made out of the same six components:

[1]https://osrg.github.io/ryu/

- Match Fields: a field against which a packet is matched. The match fields consist of the ingress port, used packet headers and optionally metadata.
- Priority: the field that determines the matching precedence of the flow entry.
- Counters: used to update for matching packets. The OpenFlow specification defines a variety of timers that can be used here.
- Instructions: has the actions that are to be taken if a match occurs.
- Timeouts: shows the maximum amount of time before the flow is expired by the switch.
- Cookies: data values chosen by the SDN controller. It is generally not used for processing packets.

OpenFlow provides a vendor-independent standardized way to manage network devices.

2.2. Multicast Protocols

The motivation and idea behind IP multicast is explained in this section. As this thesis implements the MEADcast protocol, it is included in this chapter. The Xcast protocol and Application Layer Multicast as alternatives to IP multicast are also looked at.

2.2.1. IP Multicast

The three most basic ways of transmitting data are through unicast, multicast and broadcast. Unicast describes the transmission of data between a single point to another point in a 1:1 communication pattern. Broadcast refers to a 1:m communication pattern, where a single point sends data to every single possible endpoint within a network. IP multicasting in comparison is the transmission of IP datagrams from one or many points to many points 1:m or n:m. The difference between broadcast and multicast is that multicast sends to a “host group” instead of the entire subnetwork[Dee89].

Host Groups

The membership of a host group is fully optional. The host group can contain either zero or an unlimited amount of members belonging to the group. The amount of host groups a host can be part of is not limited either and it is not required to be a member of a specific host group to send data into it.

Levels of conformance

Three levels of multicast conformance have been specified in RFC 1112 [Dee89].

Level 0: no support for IP multicasting. - Level 0 means no conformance and thus no support for IP multicasting. The hosts in this group will usually be unaffected by multicast traffic.

Level 1: support for sending but not receiving multicast IP datagrams. - Hosts in this group have the ability to send multicast data to multicast groups but not receive multicast data themselves. It is easy to upgrade from Level 0 to Level 1, because sending to a specific multicast destination group address is similar to sending an unicast packet to the IP address of a single host.

Level 2: full support for IP multicasting. - Members in this group may join or leave host groups at will. Like Level 1 they can send to host groups and can also receive multicast IP datagrams by being in host groups. Hosts in host groups express their membership in a group by using mandatory the Internet Group Management Protocol (IGMP) as described in RFC 3376 [CDK$^+$02] and RFC 4604 [HH06].

Multicast Routing

Many multicast routing algorithm feature a distribution tree. They are used to forward the data from the sender to the hosts while minimizing the amount of packet duplication. Fundamentally, there are two different kind of trees:

Source-specific trees: Every sender maintains their own distribution tree and every sender is the root in that tree. Sender and receiver are connected via the shortest possible path.

Shared trees: Every multicast group uses a shared tree for all sources. The root of the distribution tree is a router and not a host. That router is sometimes called rendezvous point.

2.2.2. Explicit Multicast

The Explicit Multicast (Xcast) protocol offers a solution to one of the problems traditional IP multicast schemes face: the scalability issues with a very large amount of distinct multicast groups as described in the “Small Group Multicast” draft [BF01] and RFC 5058 [OFI$^+$07].

Xcast supports a large number of multicast groups with a small number of recipients in each individual group.

Overview

Instead of sending the data to a multicast group address, the sender encodes each destination IP Address in the Xcast header. The sender then sends the packet to the router, which has to perform a routing table look up for every destination listed in the Xcast header in order to determine the next hop of each packet. The router then partitions the destination list based on the prior step and sends the modified packet with the new Xcast header to the corresponding path. This continues until there is only one destination left. In this case the router converts the Xcast packet into a unicast packet in a process called Xcast to Unicast (X2U) and handles it as a unicast packet.

If the network has routers not capable of Xcast, other mechanisms have to be implemented. Possible solutions to this problem are tunneling between Xcast capable routers or premature X2U if a Xcast router detects that its downstream neighbor is not a Xcast router. Another special solution to this problem is to deploy Xcast by upgrading only the hosts. If a Xcast capable host receives a packet it can process the other destinations in the Xcast header and send the packet to the remaining destinations.

Advantages

A lot of advantages of traditional multicast schemes apply to Xcast as well. The main one being the minimization of bandwidth usage for small groups. An advantage of Xcast is, that it eliminates the per-session signaling and per-session state information of traditional multicast schemes, which allows it to support a greater number of multicast sessions. Xcast itself provides a lot of flexibility too. Depending on the current network situation Xcast can be used as an alternative to unicast or multicast. Other advantages include simplicity and no need for multicast routing protocols.

Disadvantages

The router however has to perform a significant amount of work. Every time it has to handle a Xcast packet it has to look up the routing table for every destination in the Xcast header and has to create a new header after every hop. If there are hosts available that have been upgraded due to lack of Xcast support in the network, more disadvantages arise. Since the host has to perform network functions, which is a disadvantage in itself, and can handle the Xcast header, it knows the identity of other parties in the session. This is an anonymity and privacy issue. Since Xcast is designed for multicast sessions with a small number of hosts in mind it is limited to a smaller range of applications compared to traditional IP multicast.

2.2.3. Application Layer Multicast

Application layer Multicast (ALM) is not based on IP multicasting or data link layer multicast. ALM is implemented in the application layer and only relies on the network for basic unicast forwarding. Group specific communication like group management, multicast tree formation or packet replication are moved to the application layer [KM05]. Compared to IP multicast it is easier to deploy and doesn't add additional overhead to the routers like saving all unique multicast group addresses. in ALM every end-host is responsible for group membership management and the construction of the overlay network with the goal of maintaining an efficient overlay for data transmission. Another difference between ALM and IP multicast is that the data in ALM is duplicated at the end-host which also perform the necessary network functions.

That raises the issues of privacy and overall efficiency of ALM since the ALM protocol must send identical packets over the same link [BBK02] [KM05].

2.2.4. Multicast in SDN

A lot of work on SDN in the past has been based on unicast traffic in SDN. This section features a small selection of related papers and work that deal with multicast in SDN.

Avalanche

Avalanche is a SDN based system that enables multicast in commodity switches, used in data centers [IKM14]. This paper introduces the Avalanche Routing Algorithm (AvRA), a new multicast routing algorithm. Avalanche is implemented as an OpenFlow controller module and AvRA is used with the goal create efficient and near-optimal multicast trees. Avalanche balances the link utilization by taking advantage of the high path diversity found in data centers. It is common for data center topologies to have multiple equal length paths between any given hosts. IP multicast might result in some of those links having a link utilization of greater than 100%, while other equal links might not get utilized at all. Avalanche balances link utilization by using all links available and reduces packet loss caused by over utilization [IKM14].

Recover-aware Steiner Tree

Recover-aware Steiner tree (RST) is a proposed multicast tree for SDN [SHYC15]. It deals with the problem of the shortest-path tree (SPT), used in current Internet, not being bandwidth-efficient. RST also deals with the problem of traditional Steiner trees (RT) not being reliable. This paper introduces an approximation algorithm called Recover Aware Edge Reduction Algorithm (RAERA) and was also implemented on a SDN. The results show that RAERA outperforms normal SPT and ST in both real and large synthetic networks, when considering latency and other factors.

LAMA

In addition to source-specific trees and shared trees found in IP multicast, SDN enables three additional alternatives to shared trees: per-group shared tree, multi-group shared tree and single shared tree. LAMA takes advantage of the multi-group shared trees. Locality-aware multicast approach(LAMA)[LLT+17], constructs multi-group shared trees in SDN, where one tree covers several multicast groups. This is done by clustering the multicast sources into a multicast cluster. Similar to IP multicast, a rendezvouz point is selected, in this case a switch, which has the minimum distance to all multicast sources. A shortest-path tree from the rendezvouz point to all hosts is constructed and based on that, the controller installs flows on the flow table of the on-tree switches. This results in a lower amount of flows installed on each switch.

Multiflow

Lucas Bondan *et al* developed and evaluated “Multiflow”, clean-slate approach to programmable networks [BMK13]. It enables traditional IP multicast functions like multicast group management and multicast transmission. Compared to IP multicast,“Multiflow” is executed in the SDN controller and has a network-wide global view of all network devices and routes. “Multiflow” takes advantage of that by creating a more efficient routing for multicast receivers and the sender. Compared to OpenMcast [BMK13], which offers network behavior similar to what is observed in normal networks, “Multiflow” achieved faster average time, from the host first joining to the host receiving the first packet.

2.2.5. MEADcast

MEADcast is a multicast protocol that allows sender-based multicast of IPV6[TD18]. It takes many core concepts and advantageous traits of Xcast (as discussed in 2.1.2). Compared to Xcast, it takes off a lot of work from the routers by shifting the work to the sender. The problems mentioned in Application Layer Multicast (as discussed in 2.1.3) are also a non issue. The end host is agnostic of the technology being used and is not required to have any MEADcast application deployed to partake in a specific MEADcast session. This results in a lower average time from the client joining the multicast group to the receipt of the first packet.

The only entities that have MEADcast implemented are the sender and the routers. MEADcast also supports the gradual deployment of MEADcast capable routers in networks and falls back to unicast if there is no MEADcast support in the network.
The information necessary to describe the protocol is as follows:

- the sender
- the set of end hosts
- the set of MEADcast capable routers in the network
- the distance of each MEADcast capable router to the sender in hops

  The process from the end host first joining the MEADcast session to the host getting the data from the sender can be described in two distinct phases: The discovery phase and the data delivery phase.

MEADcast Protocol Header

MEADcast takes advantage of extension headers and 2.4 shows the most important fields.

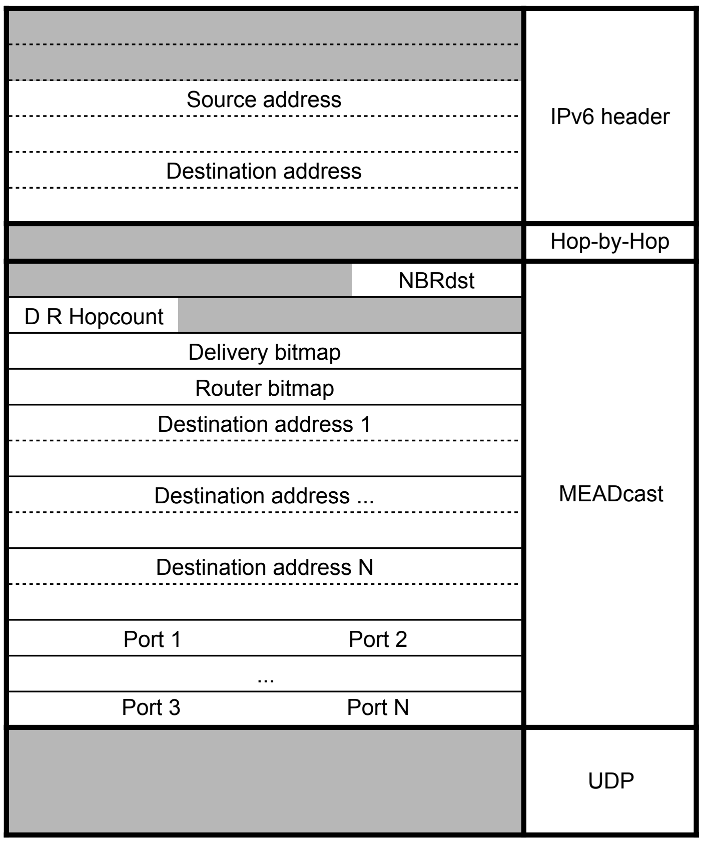


Figure 2.4.: MEADcast packet headers[TD18]

- IPv6 header
  - Source address: IPv6 address of the sender.
  - Destination address: IPv6 destination address of the receiver.
- Hop-by-Hop extension header This is required to make every router examine MEADcast packets.
- MEADcast extension header
  - NBRdst: number of destinations encoded in the destination address list of the MEADcast header. This number can not be higher than 64.
  - D : single bit to classify the packet as a discovery request or data delivery packet. If it is set, it is a discovery request packet. If not it is a data delivery packet.
  - R : single bit to classify if a discovery request packet is a request or response packet. If it is set, it is a discovery response packet. If not it is a discovery request packet.
  - Hopcount : stores the distance from a MEADcast capable router to the sender in hops between each MEADcast routers. It is used in the discovery phase for the sender to build his own view of the network.
  - Router Bitmap: shows the position of MEADcast routers in the destination address list. A value of 1 points to a MEADcast router while a value of 0 points to a receiver.
  - Delivery Bitmap: shows if this packet has been received by that specific MEADcast router yet. This information is important to know for the current MEADcast router that handles this packet, since it will forward or duplicate and forward to those MEADcast routers depending on this bitmap.
  - Destination address 1 . . . n: IPv6 addresses of the receivers and the responsible MEADcast routers. There can't be more than 64 IP addresses in this field.
  - Port 1 . . . n: the transport layer protocol port number for each corresponding receiver. In this case UDP port numbers.
- UDP header Since Transmission Control Protocol (TCP) can not be used for multicasting, User Datagram Protocol (UDP) is the only available choice.

The phases of the MEADcast transmission process

Each MEADcast entity has different functions that it needs for the successful completion of each phase. They are further described below. In the following sub chapters the sender is denoted as S, the set of end hosts as E, the set of MEADcast capable routers as R and the distance of MEADcast router to the sender in hops as d [TD18].

Discovery phase

The goal of the discovery phase is for the sender to gain knowledge about every MEADcast capable router and it's position on the path to the end host. The sender then builds a network topology from the sender viewpoint based on the distance of each router and their associated end hosts to the sender. In order to achieve that both the sender and routers are involved in this process.

1) The multicast session is established

2) The sender starts sending the data to the recipients until step 5) is finished.

3) The sender sends a MEADcast discovery request, which can be denoted as *req(E,0)* to each recipient. The *0* in *req(E,0)* indicates the initial value 0 for the distance d of a MEADcast router to the sender. In the actual MEADcast header the discovery bit D is set to 1, the response bit R is clear, the *hopcount* field, *routerbitmap* and *deliverybitmap* are all empty.

4) Upon receipt of the discovery request packet a normal router forwards the packet normally to the end host *E.* If a MEADcast router receives it then the router will duplicate that packet and increase d of one of the packets by 1 and and forward it to the destination. The forwarded discovery request packet can be denoted as: *req(E, d+1).* The Router does that by incrementing the *hopcount* field by 1. Additionally the discovery response packet *resp(E, d+1, R)* gets sent back to the Sender. The response packet is created by setting the discovery bit D to 0, the response bit R to 1 and *destinationnumber* field to 1. The first destination in the MEADcast header is the IP address of the intended recipient.

This gets repeated for every router on the path to the end host.

5) After the sender receives all the discovery response packets or after a timeout the sender builds the MEADcast data delivery packet based on the current viewpoint of the topology the sender has. The viewpoint of the topology gets built at the sender as the discovery response packets from each MEADcast router arrive.

6) The sender has knowledge about the network topology from its viewpoint now and can start sending MEADcast data delivery packets or unicast packets to the recipients based on the topology.

Data delivery phase

7) The sender starts sending the MEADcast packets or unicast packets to the recipients after finishing the discovery phase.

If a end host is included in the MEADcast packet or receives a unicast packet directly depends on the implementation. A MEADcast router R_1 that has the same amount of end hosts it's responsible for should get a higher priority than R_2 if R_1 has a higher hop count than R_2 for efficiency sakes. It would be a waste of header space for the same reason to include MEADcast routers with only one responsible end host.

If there are no MEADcast capable routers in the network then no response packages will be received by the sender. If that is the case then the sender will go back to step 2) and send the data out in unicast.

8) Upon receiving a MEADcast packet meant for data delivery the MEADcast router starts a process called *Decomposition*[TD18], which is the process of decomposing a MEADcast packet. It determines the end hosts it is responsible for, by looking at these fields: *routerbitmap* and *deliverybitmap* and creates unicast packets for each of them with the specified payload. It will also generate and send modified MEADcast packets based on the remaining recipients to the remaining routers. When the opposite is true however and more receivers are present than the maximum amount of encodable addresses in the MEADcast header, the sender will divide the MEADcast packet into suitable subgroups, each with their own MEADcast packet.

Example of data transmission in a network with full MEADcast support

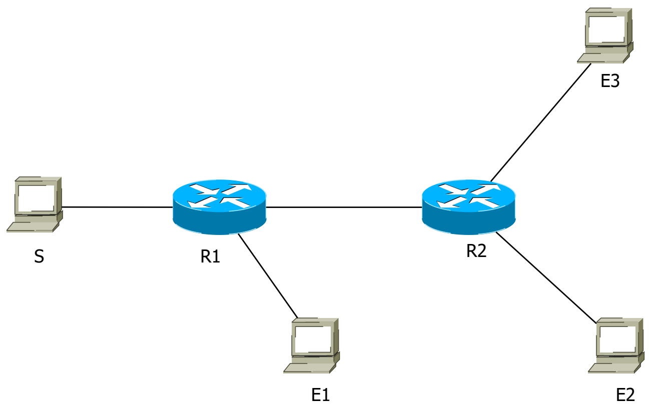


Figure 2.5.: Data transmission in a topology with full MEADcast support.

Figure 2.5 depicts a simple network with one sender S, two MEADcast capable routers R_1 and R_2 and three receivers E_1, E_2 and E_3.

The communication between the sender S, routers R_1 and R_2 and receivers E_1, E_2 and E_3 are as follows:

1. S and E_1, E_2, E_3 establish a MEADcast session through unicast.
2. S starts sending the data via unicast to E_1, E_2, E_3.

3. The unicast packets get forwarded normally by R_1 and R_2.
4. S creates a MEADcast discovery request packet for each receiver, three in total: *req(E_1,0) req(E_2,0) req(E_3,0)* and sends them to R_1. The discovery bit is set and response bit is clear.
5. R_1 receives *req(E_1,0)*. The discovery bit is set and response bit is clear. R_1 is the router responsible for E_1 and sends back a discovery response packet *resp(E_1, 1, R_1)*. The response bit of that packet is set.
6. R_1 receives *req(E_2,0)*. The discovery bit is set and response bit is clear. E_2 is not connected to R_1. R_1 duplicates the packet, sends back a discovery response packet *resp(E_2, 1, R_1)* to S and furthermore forwards *req(E_2,1)* to R_2.
7. R_1 receives *req(E_3,0)*. The discovery bit is set and response bit is clear. E_3 is not connected to R_1. R_1 duplicates the packet, sends back a discovery response packet *resp(E_3, 1, R_1)* to S and furthermore forwards *req(E_3,1)* to R_2.
8. R_2 receives *req(E_2,1)*. The discovery bit is set and response bit is clear. R_2 is the router responsible for E_2 and sends back a discovery response packet *resp(E_2, 2, R_2)* to S. The response bit of that packet is set.
9. R_2 receives *req(E_3,1)*. The discovery bit is set and response bit is clear. R_2 is the router responsible for E_3 and sends back a discovery response packet *resp(E_3, 2, R_2)* to S. The response bit of that packet is set.
10. R_1 receives *resp(E_2, 2, R_2)*. The discovery bit is set and response bit is set. Since the response bit is set, R_1 forwards the packet to S.
11. R_1 receives *resp(E_3, 2, R_2)*. The discovery bit is set and response bit is set. Since the response bit is set, R_1 forwards the packet to S.
12. E_1, E_2 and E_3 receive the unicast packets sent at step 2 normally. Every MEADcast packet gets dropped however since they don't understand it.
13. S receives *resp(E_1, 1, R_1)*. It starts building the network topology from the sender viewpoint, based on the response packets so far: (R_1, 1, E_1).
14. S receives *resp(E_2, 1, R_1)*. It starts building the network topology from the sender viewpoint, based on the response packets so far: (R_1, 1, E_1,E_2).
15. S receives *resp(E_3, 1, R_1)*. It starts building the network topology from the sender viewpoint, based on the response packets so far: (R_1, 1, E_1, E_2, E_3). The current topology view can be seen in 2.6.
16. S receives *resp(E_2, 2, R_2)*. It starts building the network topology from the sender viewpoint, based on the response packets so far: (R_1, 1, E_1, E_2, E_3), (R_2, 1, E_2). Since R_2 has a higher hopcount than R_1 for E_2, the network topology looks like depicted in Figure 2.7.
17. S receives *resp(E_3, 2, R_2)*. It starts building the network topology from the sender viewpoint, based on the response packets so far: (R_1, 1, E_1, E_2, E_3) , (R_2, 1, E_2, E_3). Since R_2 has a higher hopcount than R_1 for E_3 the network topology looks like depicted in Figure 2.8.
18. S stops sending unicast packets and begins the MEADcast data delivery phase. The IP addresses included in the MEADcast header are: (R_2, E_2, E_3). The corresponding

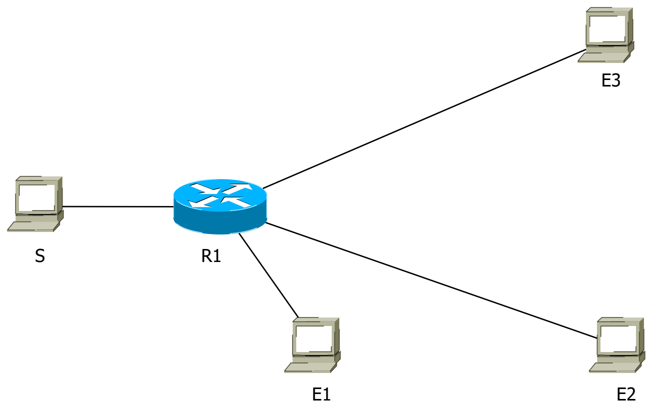


Figure 2.6.: Network topology from sender viewpoint after receiving the first three response packets.

Router Bitmap is 100, where 1 marks the router in the destination address list and 0 the end points. Because R_1 was only responsible for one receiver E_1, they were not included in the MEADcast header. Both the discovery bit and the response bit are cleared

19. S starts sending MEADcast packets to R_1 and the unicast packet towards E_1.
20. R_1 receives the unicast packet and forwards it normally to E_1.
21. R_1 receives the MEADcast packet. The discovery bit is clear and response bit is clear, which indicates a data delivery packet. It sees that it is not responsible for any hosts on the destination list. It sees that R_2 still has yet to receive the packet. The packet doesn't change and R_1 forwards the packet to R_2.
22. R_2 receives the MEADcast packet from R_1. It sees that it is responsible for the two hosts E_2 and E_3. R_2 creates two unicast packets with the payload of the MEADcast packet and transmits them to E_2 and E_3. The Bitmap value of 100 shows that R_2 is the last router to receive this MEADcast packet; hence, R_2 is not required to forward the packet.
23. E_1, E_2 and E_3 all receive the unicast packets normally.

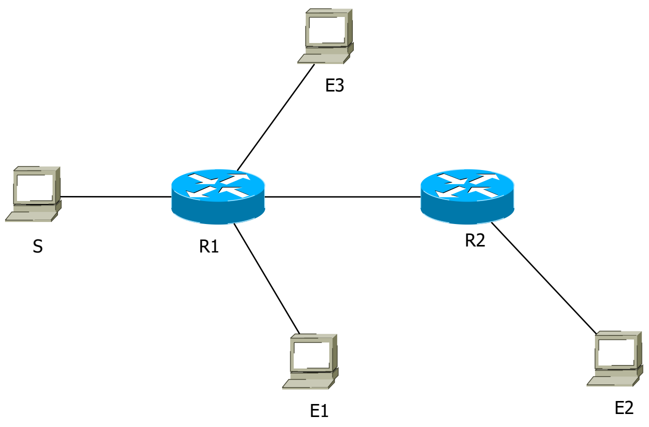


Figure 2.7.: Network topology from sender viewpoint after receiving four response packets.

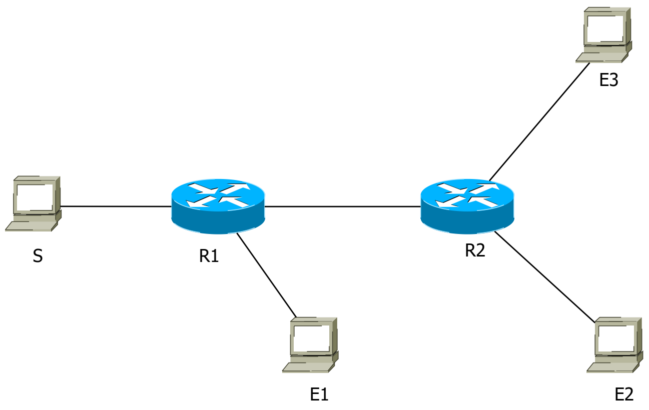


Figure 2.8.: Final topology view from sender viewpoint after receiving all response packets.

3. Analysis and prototype design

The idea of deploying multicast, or in this case MEADcast, in SDN is an interesting one. MEADcast can benefit a lot from SDN. One example for this is the discovery phase of MEADcast. The overall goal of the discovery phase is to create a topology view from the sender, that shows the most efficient way of forwarding the packet. It is similar to the IP multicast forwarding tree in that aspect. They both strive to minimize the amount of packet duplicating and network load.

SDN opens up more possibilities. A SDN controller has a global view over the network and links between network devices. This can be used to create the most efficient network topology viewpoint for MEADcast, well in advance and faster than any sender could. It is possible to take it one step further, by having the controller be the one to act as the root of the topology. The feasibility and practicability of this approach gets explored in this chapter.

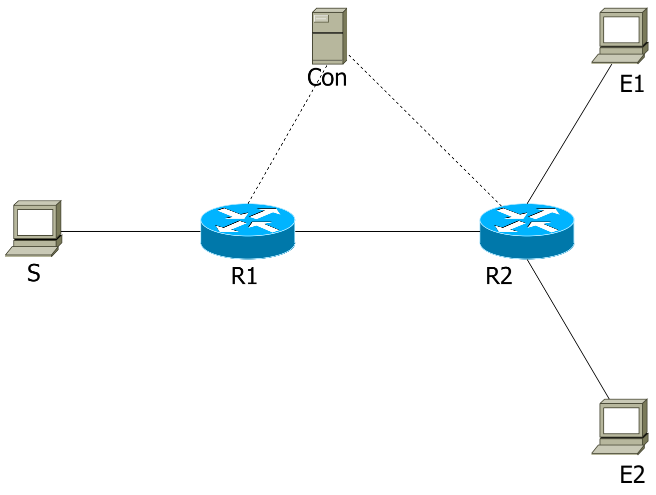


Figure 3.1.: Simple SDN scenario.

A simple scenario is depicted in Figure 3.1. It assumes one sender, two OpenFlow switches, one controller and two end hosts and will be used to demonstrate the difference between each approach.

3.1. Different transmission approaches and prototyping

This section focuses on different approaches on how the data can get transmitted in the given scenario. It compares unicast to a number of MEADcast implementation prototypes.

3.1.1. Unicast transmission

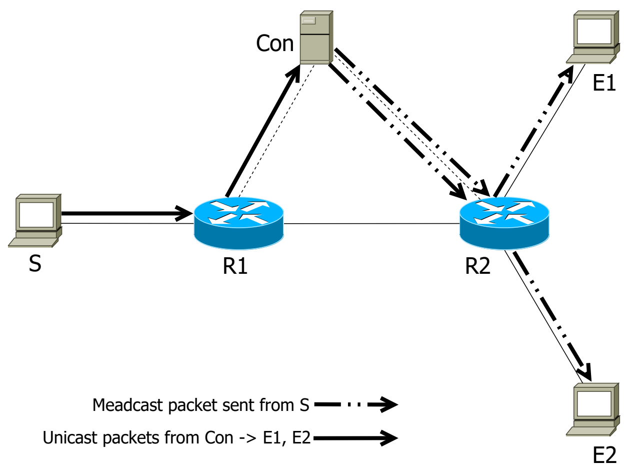

Con
E1
S
R1
R2
E2
Meadcast packet sent from S
Unicast packets from Con -> E1, E2

Figure 3.2.: Unicast transmission with flows already installed on switches

Figure 3.2 shows how unicast packets would get forwarded in SDN with flows already installed on the switch. The amount of packets sent, depend on the number of receivers the sender wishes to address. In this case E_1 and E_2 are the receivers, thus two unicast packets are sent out. Unicast results in a total of six transmissions in this scenario. This serves as the baseline for comparison.

3.1.2. Ideal MEADcast transmission

Figure 3.3 shows how MEADcast would ideally be used to transmit data in this topology. It only shows the data delivery and assumes that all the necessary flows are already installed on the switches. The sender sends one MEADcast packet to R_1. R_1 then forwards it to R_2 since it is not responsible for any end hosts. R_2 receives the packet and sends the payload to E_1 and E_2 in unicast. Transmitting the packets via the ideal MEADcast implementation results in a total of four packets sent in this scenario.

Since the limitations of the switches prohibit the implementation in that way, an alternative has to be used. The limitations stem from OpenFlow 1.3 and Open vSwitch. OpenFlow by design limits the switches ability to process MEADcast packets, even though IPv6 extension header handling support was introduced with OpenFlow 1.3 [1]. The switches may be able to detect packets with extension headers such as the MEADcast header and forward them based on installed flows, but they can not process the MEADcast packet itself. As a result, every MEADcast packet has to get sent to the controller for further processing. Although OpenFlow 1.3 offers extension header support, it is not supported by any version

[1]https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

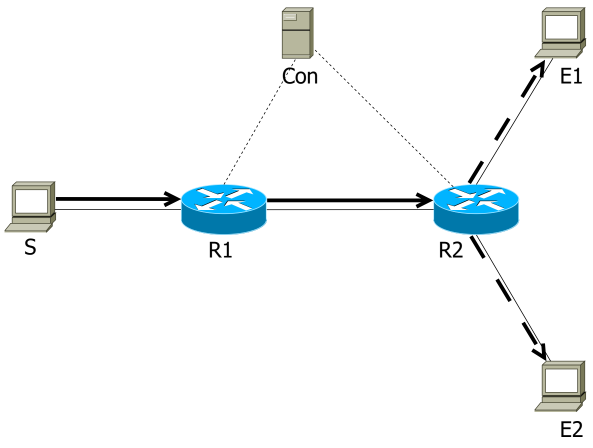


Figure 3.3.: Ideal MEADcast data delivery in a SDN.

of Open vSwitch [2]. This further limits the switches ability to detect MEADcast packets and facilitates the need of using alternative ways of processing MEADcast packets. Two prototypes on how to deal with this problem are suggested in the following subsections.

3.1.3. MEADcast prototype design I

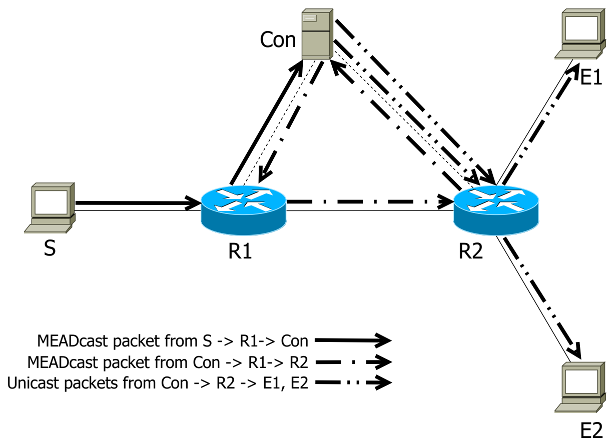

Con
E1
S
R1
R2
E2
MEADcast packet from S -> R1-> Con
MEADcast packet from Con -> R1-> R2
Unicast packets from Con -> R2 -> E1, E2

Figure 3.4.: Naive MEADcast implementation in a SDN.

A naive implementation of MEADcast on that scenario is to have the switches act as MEADcast routers. This can be seen in Figure 3.4, where only the data delivery phase of the MEADcast transmission is shown. The OpenFlow switches have no control plane capabilities to process MEADcast packets however and flow tables with their matching fields are very limited in their use for this case. The controller is forced to handle all MEADcast packets in this case. That means that every incoming MEADcast packet at every switch is

[2]https://docs.openvswitch.org/en/latest/topics/openflow/

sent to the controller for further processing. This implementation would result in a higher total traffic volume on the network compared to unicast and thus will be ignored.

3.1.4. MEADcast prototype design II

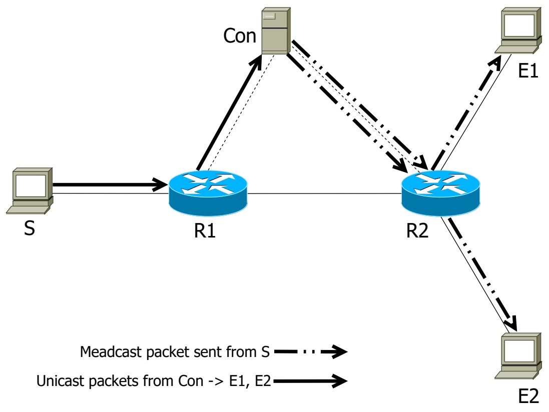

Con
E1
S
R1
R2
E2
Meadcast packet sent from S
Unicast packets from Con -> E1, E2

Figure 3.5.: Chosen approach for deploying MEADcast in a SDN.

Another possible implementation of MEADcast in SDN is to let the controller handle all MEADcast packets for the switches right away, instead of simulating normal MEADcast router behavior. This is shown in figure 3.5. Since the controller has a global view of all connected switches in the network it can generate the necessary unicast packets itself and send them to the hosts through the switches. The switches only perform simple forwarding operations in this case. The internal topology of the SDN does not matter. This is because the controller has a connection to all switches it is responsible for and can use that connection to send packets to the receivers. This put more load on the connection between the controller and the switch, since the controller has to create the unicast packet for the hosts now. This was the chosen prototype and is further explored.

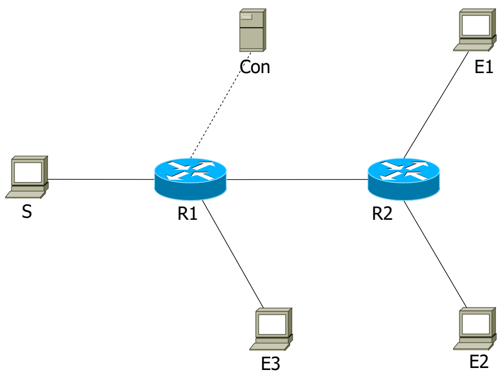


Figure 3.6.: Example SDN with three receivers

Figure 3.6 will be used to demonstrate the MEADcast prototype II.

1. Receiver E_1, E_2, E_3 and the sender S establish a MEADcast session.
2. S starts sending the data via unicast to E_1, E_2, E_3.
3. R_1 receives the unicast packets sent by S and forwards them normally to the intended destinations. The same applies to R_2.
4. S creates a MEADcast discovery request packet for each receiver, three in total: *req(E_1,0) req(E_2,0) req(E_3,0)* and sends them to R_1. The discovery bit is set and response bit is cleared.
5. R_1 receives *req(E_1,0)*, *req(E_2,0)* and *req(E_3,0)* . The Switch forwards them all to the controller.
6. The controller Con receives receives *req(E_1,0)*, *req(E_2,0)* and *req(E_3,0)* . The discovery bit is set and response bit is clear on all three packets. Since the controller is responsible for every end host in its SDN it generates three response packets *resp(E_1, 1, R_1)*, *resp(E_2, 1, R_1)* and *resp(E_3, 1, R_1)* and sends them back to R_1 with further instructions to forward those MEADcast packets back to S only. The discovery bit and the response bit were also set to 1.
7. R_1 receives *resp(E_1, 1, R_1)*, *resp(E_2, 1, R_1)* and *resp(E_3, 1, R_1)* from Con and forwards them back to S.
8. S receives *resp(E_1, 1, R_1)*, *resp(E_2, 1, R_1)* and *resp(E_3, 1, R_1)*. It starts building the network topology from the sender viewpoint based on the response packets so far: (R_1, 1, E_1, E_2, E_3).

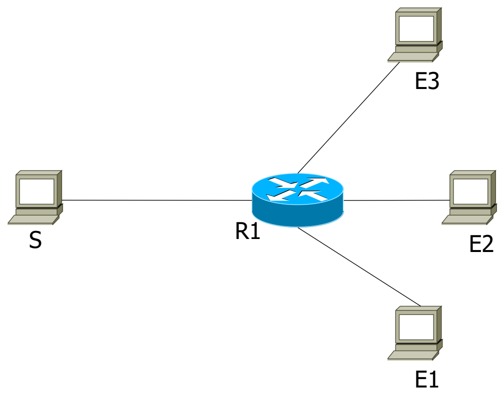


Figure 3.7.: Network topology from sender viewpoint after receiving the three response packets.

9. S stops sending unicast packets and begins the MEADcast data delivery phase. The IP addresses included in the MEADcast header are: (R_1, E_1, E_2, E_3). The corresponding router bitmap is 1000, where 1 marks the router in the destination address list and 0 the end points. Both the discovery bit and the response bit are clear.
10. S starts sending MEADcast packets to R_1.
11. R_1 receives the MEADcast packet. R_1 doesn't know how to process MEADcast packets, and sends them to the controller.

12. *Con* receives the MEADcast packet. Both the discovery bit and the response bit are clear. The controller processes it further because it is a data delivery packet. It sees the destination list, delivery bitmap and router bitmap in the MEADcast header and sees that it is responsible for all receivers listed in the destination list. The controller creates three unicast packets out of the MEADcast packet for E_1, E_2 and E_3 and sends them to the corresponding switches. The one unicast packet destined for E_1 gets sent to R_1 while the two remaining unicast packets for E_1 and E_2 are sent to R_2.
13. R_1 receives the unicast packets sent by *con* and handles them normally according to the flow tables and forwards to the intended destinations. The same applies to R_2.
14. E_1, E_2 and E_3 all receive the unicast packets normally.

3.2. Implementation of MEADcast on the topology

This section deals with the implementation of the MEADcast prototype II. There are three applications that had to be implemented: a traffic generator capable of generating MEADcast traffic, a RYU network application that is capable of handling and processing those MEADcast packets and a UDP server deployed at each receiver.

MEADcast traffic generator

The traffic generator implements the MEADcast protocol on the sender side. It consists of two parts. The first part is a function responsible for crafting the MEADcast packets. The second part is the traffic generator that handles the discovery and data delivery phase.

Each MEADcast packet is created by crafting every necessary packet header with raw socket programming. This is done with a MEADcast packet crafter function *mcf()*. The MEADcast packet is a combination of IPv6 header, Hop-by-Hop extension header, MEADcast extension header and the UDP header. The Ipv6 header, Hop-by-Hop extension header and UDP header follow the header format specified in RFC 2460 [HD98]. IPv6 headers and UDP headers require additional information about the entire packet. The Ipv6 header needs to include the payload length. The payload length is a 16-bit unsigned integer and calculated by summing up the length of the rest of the packet following the IPv6 header, in octets. Additionally, UDP requires the calculation of a checksum. UDP checksum calculation is performed over a “pseudo-header”. The “pseudo-header” is a header consisting of the IP source address, destination address, upper-layer packet length and next header. The final checksum for UDP is calculated based on that “pseudo-header”.

The MEADcast extension header is made based on the topology viewpoint the sender currently has. If the topology viewpoint is empty a discovery request packet will be generated. If the opposite is true, a data delivery packet will be generated. The discovery request packet can be written as *req()* and the data delivery packet as *deli()*. The MEADcast header for the MEADcast data delivery packet varies in size, since the amount of receivers dictate the amount of destination addresses and ports encoded in the header. The delivery packet does not include every IP address and router of the topology network view. It decides based on the hop count of each MEADcast capable router and the amount of destinations they are responsible for if they get included or not. This is done by sorting the network topology view by hop count in descending order and by excluding every router that is only responsible for

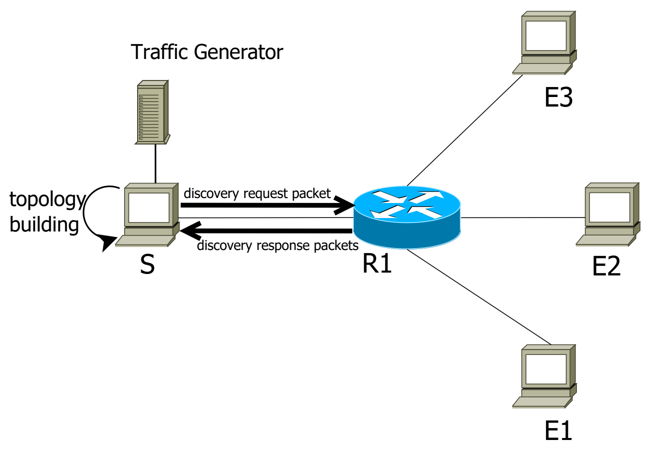


Figure 3.8.: MEADcast traffic generator - topology discovery phase.

one destination. The activity diagram of the MEADcast packet crafter function can be seen in Figure 3.10.

In order to start the MEADcast data delivery process, the sender has to know the network topology of all destinations. This is done by sending a *req()* to every receiver. The traffic generator then waits until all discovery response packets *resp()* arrive back at the sender, or until a certain time threshold has passed. The sender then creates the topology view based on all the *resp()* it has received. Because the discovery phase is finished almost instantly for small topologies like the ones tested, the step where the sender sends unicast packets while waiting for the response packets gets ignored.

It then uses the MEADcast packet crafter function with the current topology viewpoint and the payload to create the MEADcast delivery packet *deli()*. The payload *p* of the packet is made by chunking the target file into suitable chunk sizes. The *deli(p)* is then sent out in addition to all the unicast packets for receivers not listed in the MEADcast header. The traffic generator creates as delivery packets as there are chunks and sends them all out in order. Figure 3.11 shows the MEADcast data transmission activity diagram for the traffic generator.

MEADcast controller

This application was also written in Python and implements the MEADcast router functions on the controller. It uses the RYU SDN Framework [3] as a baseline for the controller application. The main function of this controller are controlling the flow-table of the switches and handling packets the switches can't handle themselves. Communication between the controller and the switch happen via OpenFlow 1.3. The OpenFlow switches run open vSwitch. The MEADcast controller is also capable of handling and creating MEADcast packets.

The controller gets started with the *–observe-links* option. This will start the controller with the topology application provided by RYU and provide the global network view.

Each switch connects to the controller via TCP. The TCP destination port is 6633. Upon

[3] https://osrg.github.io/ryu/

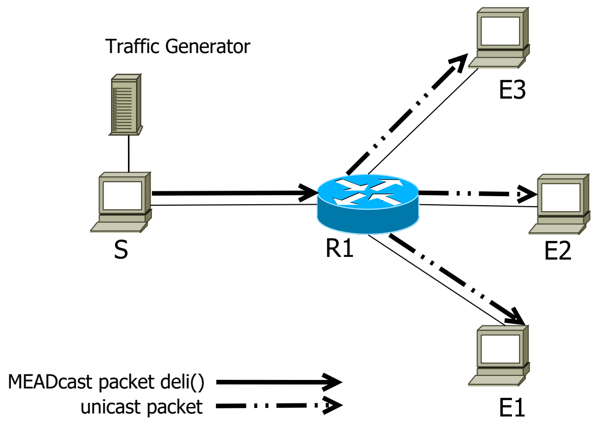


Figure 3.9.: MEADcast traffic generator - delivery phase.

establishing a connection between controller and switch, two flows get added to the switch. The first flow is the default flow and instructs the switch to forward every packet to the controller. The priority of that flow is the lowest value 0, meaning that the switch will only forward it to the controller if there is no higher priority flow on the flow-table. The second flow is used to identify MEADcast packets. OpenFlow 1.3 officially supports IPv6 Extension Header handling[4]. A new flow match field got implemented, which enables OpenFlow switches to match for IPv6 extension headers. There is no version of open vSwitch however, that supports this feature[5]. As a result, the switch can't distinguish MEADcast packets from normal packets. The second flow is a temporary solution to circumvent that problem. The flow has a priority of 500 and contains the following match rules: IPv6 header, UDP header and UDP destination port 5005. This means that every incoming IPv6 MEADcast packet with the UDP destination port 5005 gets sent to the controller. This is used as a pseudo identifier, instead of the missing extension header handling feature.

For the controller to both handle the MEADcast packet handling and normal controller activity it is necessary to implement more functions. The two of them being Neighbor Discovery (ND) and MEADcast packet handling. ND is needed because the global network view of the controller only includes switches in the network but not the hosts. Figure 3.14 shows the activity diagram for the controller for handling MEADcast packets. It assumes that the default flow entry and the MEADcast flow entry are already installed on the flow table of each switch.

ND is performed by using the Neighbor Discovery Protocol (NDP) as specified in RFC 4861 [SNNS07]. NDP is a protocol used with IPv6 and is responsible for gathering information required for Internet communication. It defines the Internet Control Message Protocol

[4]https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf
[5]https://docs.openvswitch.org/en/latest/topics/openflow/

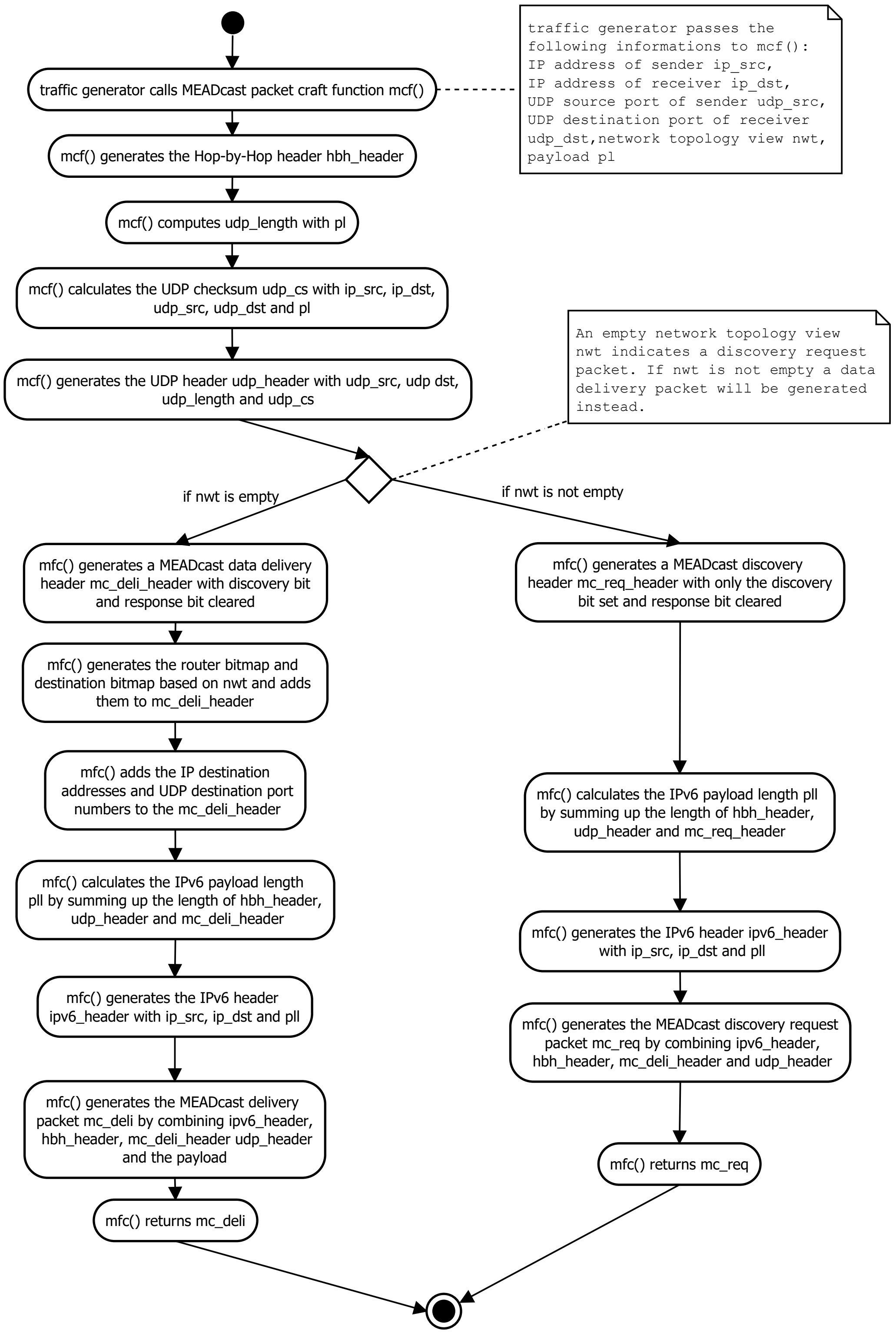

traffic generator passes the following informations to mcf(): IP address of sender ip_src, IP address of receiver ip_dst, UDP source port of sender udp_src, UDP destination port of receiver udp_dst,network topology view nwt, payload pl
traffic generator calls MEADcast packet craft function mcf()
mcf() generates the Hop-by-Hop header hbh_header
mcf() computes udp_length with pl
mcf() calculates the UDP checksum udp_cs with ip_src, ip_dst, udp_src, udp_dst and pl
mcf() generates the UDP header udp_header with udp_src, udp dst, udp_length and udp_cs
An empty network topology view nwt indicates a discovery request packet. If nwt is not empty a data delivery packet will be generated instead.
if nwt is empty
if nwt is not empty
mfc() generates a MEADcast data delivery header mc_deli_header with discovery bit and response bit cleared
mfc() generates the router bitmap and destination bitmap based on nwt and adds them to mc_deli_header
mfc() adds the IP destination addresses and UDP destination port numbers to the mc_deli_header
mfc() calculates the IPv6 payload length pll by summing up the length of hbh_header, udp_header and mc_deli_header
mfc() generates the IPv6 header ipv6_header with ip_src, ip_dst and pll
mfc() generates the MEADcast delivery packet mc_deli by combining ipv6_header, hbh_header, mc_deli_header udp_header and the payload
mfc() returns mc_deli
mfc() generates a MEADcast discovery header mc_req_header with only the discovery bit set and response bit cleared
mfc() calculates the IPv6 payload length pll by summing up the length of hbh_header, udp_header and mc_req_header
mfc() generates the IPv6 header ipv6_header with ip_src, ip_dst and pll
mfc() generates the MEADcast discovery request packet mc_req by combining ipv6_header, hbh_header, mc_deli_header and udp_header
mfc() returns mc_req

Figure 3.10.: MEADcast packet crafter activity diagram.

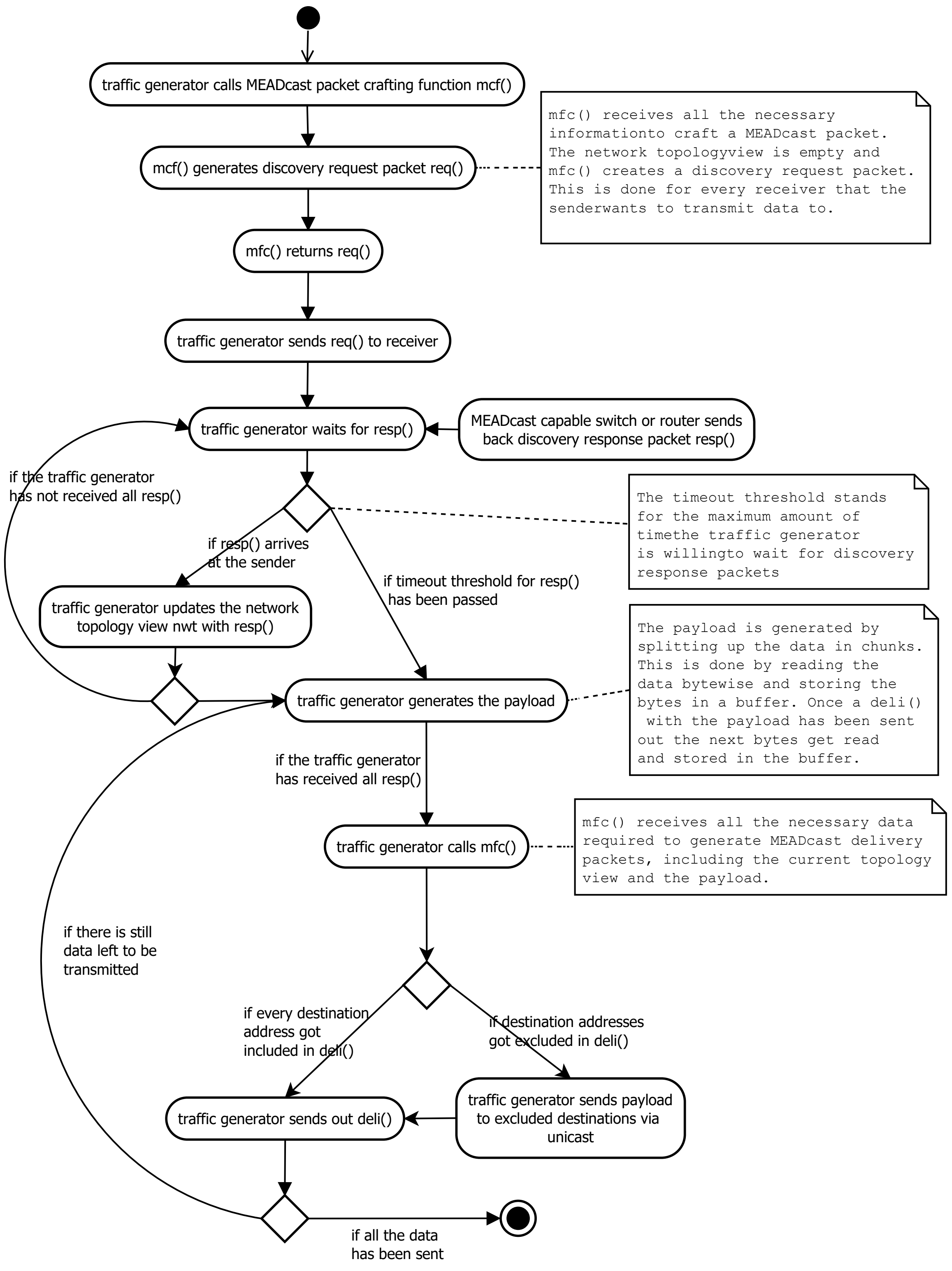

traffic generator calls MEADcast packet crafting function mcf()
mfc() receives all the necessary informationto craft a MEADcast packet. The network topologyview is empty and mfc() creates a discovery request packet. This is done for every receiver that the senderwants to transmit data to.
mcf() generates discovery request packet req()
mfc() returns req()
traffic generator sends req() to receiver
traffic generator waits for resp()
MEADcast capable switch or router sends back discovery response packet resp()
if the traffic generator has not received all resp()
The timeout threshold stands for the maximum amount of timethe traffic generator is willingto wait for discovery response packets
if resp() arrives at the sender
if timeout threshold for resp() has been passed
traffic generator updates the network topology view nwt with resp()
The payload is generated by splitting up the data in chunks. This is done by reading the data bytewise and storing the bytes in a buffer. Once a deli() with the payload has been sent out the next bytes get read and stored in the buffer.
traffic generator generates the payload
if the traffic generator has received all resp()
traffic generator calls mfc()
mfc() receives all the necessary data required to generate MEADcast delivery packets, including the current topology view and the payload.
if there is still data left to be transmitted
if every destination address got included in deli()
if destination addresses got excluded in deli()
traffic generator sends payload to excluded destinations via unicast
traffic generator sends out deli()
if all the data has been sent

Figure 3.11.: MEADcast traffic generator activity diagram - discovery phase and delivery phase

(ICMP) packet types, of which two of which are used here. They are ICMP type 135 *Neighbor Solicitation*(NS) for determining the link layer address of a neighbor and ICMP type 136 *Neighbor advertisement*(NA) for responding to *Neighbor Solicitation* messages. Figure 3.12 illustrates the path the MEADcast discovery packets take and the interaction of NA and NS packets between controller, switch and end hosts PC2 and PC3.

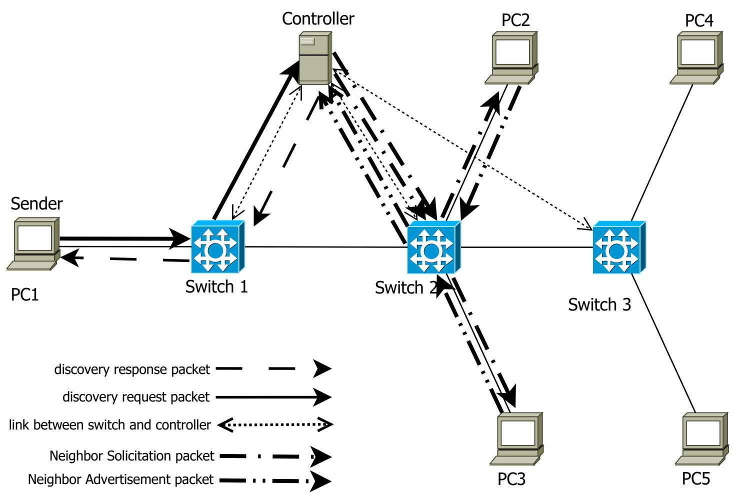

Controller
PC2
PC4
Sender
PC1
Switch 1
Switch 2
Switch 3
discovery response packet
discovery request packet
link between switch and controller
Neighbor Solicitation packet
Neighbor Advertisement packet
PC3
PC5

Figure 3.12.: MEADcast discovery phase and Neighbor Discovery shown for PC2 and PC3.

The MEADcast packet handler is responsible for handling MEADcast packets. It had to be implemented because the native RYU packet handler can't process MEADcast packets. The MEADcast packet handler detects if it deals with a MEADcast packet by dismantling the packet into all the headers it is made up of. This means IPv6 header, Hop-by-Hop extension header, MEADcast extension header and the UDP header. By looking at the *next header* field of the Hop-by-Hop header it can determine if it is a MEADcast packet or not. This happens to be the case if the *next header* field is 253 or 254. Those two values are used as experimental code points for the *next header* field as specified in RFC 4727 [Fen06].

The packet handler will then look after the discovery and response bit in the MEADcast extension header to determine if it is a discovery request *req()* packet or a data delivery packet. If the former is the case, the packet handler floods out a NS packet for the destination addresses encoded in *req()* to all switches. If a NA packet confirms the presence of the destination, a discovery response *res()* packet will get sent back to the source of the *req()*. The controller will save the destination IP address and the MAC address parsed from the NA packet in a dictionary *ndp_cache_db*. Additionally the responsible switch and the port where that NA packet originates from are added to *ndp_cache_db*.

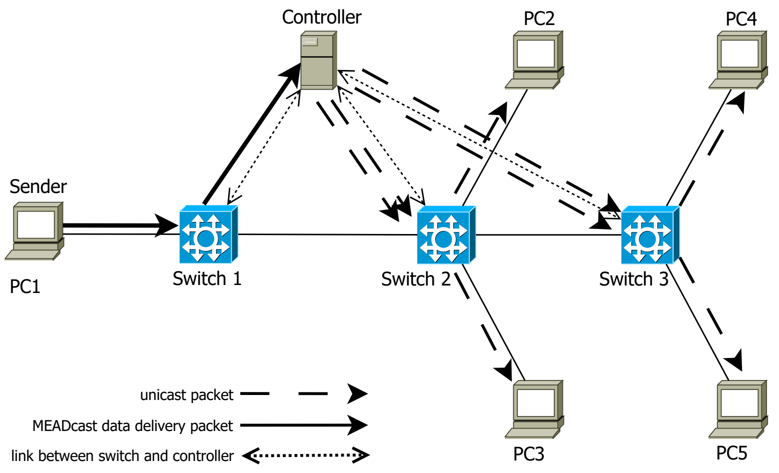


Figure 3.13.: MEADcast data delivery phase in SDN

If the packet is a MEADcast data delivery *deli()* packet, the packet handler will parse out all the necessary information from the MEADcast extension header. They consist of the *destination bitmap*, *router bitmap* and all *destinations* and *ports* encoded in the MEADcast header. By comparing the *destination bitmap* with the *router bitmap* it can determine the destination addresses the controller is responsible for. The packet handler then parses out the payload from the *deli()* packet and constructs a IPv6 packet with an Ethernet frame for the intended receivers. The MAC address needed for the Ethernet frame is saved in *ndp_cache_db*. The packet handler then sends out an OpenFlow packet to the switch associated with that IP address by looking it up in *ndp_cache_db*. The packet contains the Ethernet header and the IPv6 as data and instructions about which port it needs to forward the data to.

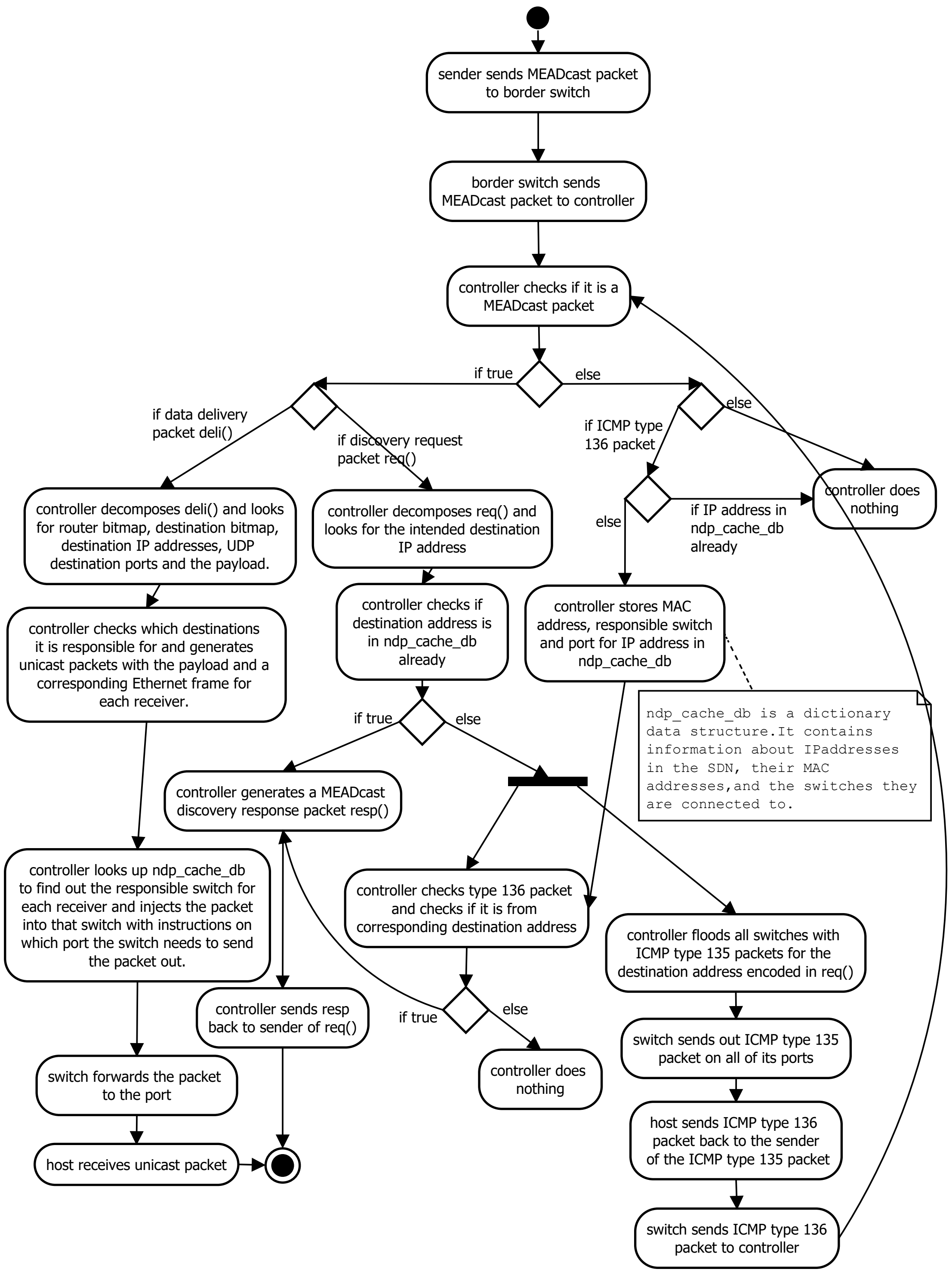

sender sends MEADcast packet to border switch
border switch sends MEADcast packet to controller
controller checks if it is a MEADcast packet
if true
else
if data delivery packet deli()
if discovery request packet req()
else
if ICMP type 136 packet
controller does nothing
else
if IP address in ndp_cache_db already
controller decomposes deli() and looks for router bitmap, destination bitmap, destination IP addresses, UDP destination ports and the payload.
controller decomposes req() and looks for the intended destination IP address
controller checks which destinations it is responsible for and generates unicast packets with the payload and a corresponding Ethernet frame for each receiver.
controller checks if destination address is in ndp_cache_db already
controller stores MAC address, responsible switch and port for IP address in ndp_cache_db
ndp_cache_db is a dictionary data structure.It contains information about IPaddresses in the SDN, their MAC addresses,and the switches they are connected to.
if true
else
controller generates a MEADcast discovery response packet resp()
controller looks up ndp_cache_db to find out the responsible switch for each receiver and injects the packet into that switch with instructions on which port the switch needs to send the packet out.
controller checks type 136 packet and checks if it is from corresponding destination address
controller floods all switches with ICMP type 135 packets for the destination address encoded in req()
controller sends resp back to sender of req()
if true
else
switch sends out ICMP type 135 packet on all of its ports
controller does nothing
switch forwards the packet to the port
host sends ICMP type 136 packet back to the sender of the ICMP type 135 packet
host receives unicast packet
switch sends ICMP type 136 packet to controller

Figure 3.14.: Controller activity diagram for MEADcast packets handling

4. Design of the Experiments and Deployment of MEADcast in SDN

To test the implementation it is required to conduct meaningful experiments.The main goal is to gain knowledge about the practicality and efficiency of the deployed MEADcast implementation in SDN. The tested topologies are described and the experiments explained in this chapter.

4.1. Tested Topologies

The experiments are tested on two different SDN topologies:

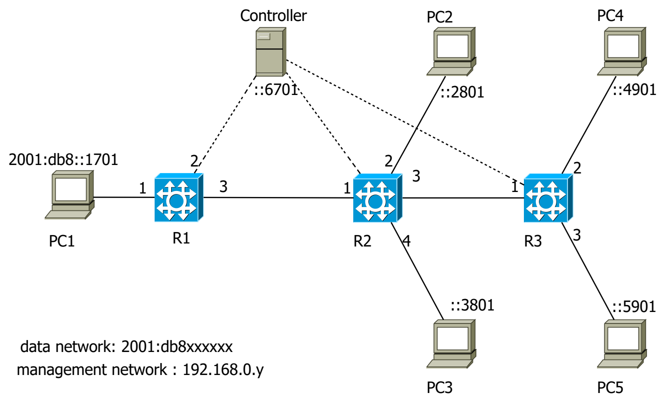


Figure 4.1.: Tested topology 1.

The first topology (Figure 4.1) is relatively small and consists of one sender PC1, three switches R1, R2 and R3, one controller and four receivers PC2, PC3, PC4 and PC5. The dotted lines, between R1, R2, R3 and the controller indicate the virtual bridge used for management between the controller and the three network switches.

The second topology (Figure 4.2) is larger. It has seven switches R1 . . . R7, three routers R8, R9 and R10, one sender PC1, one controller and nine receivers. The dotted lines between R7, R8 and R9 indicate the virtual bridge used for management, linking the controller and all the network devices. It should be noted however that every switch is connected to the controller in that manner.

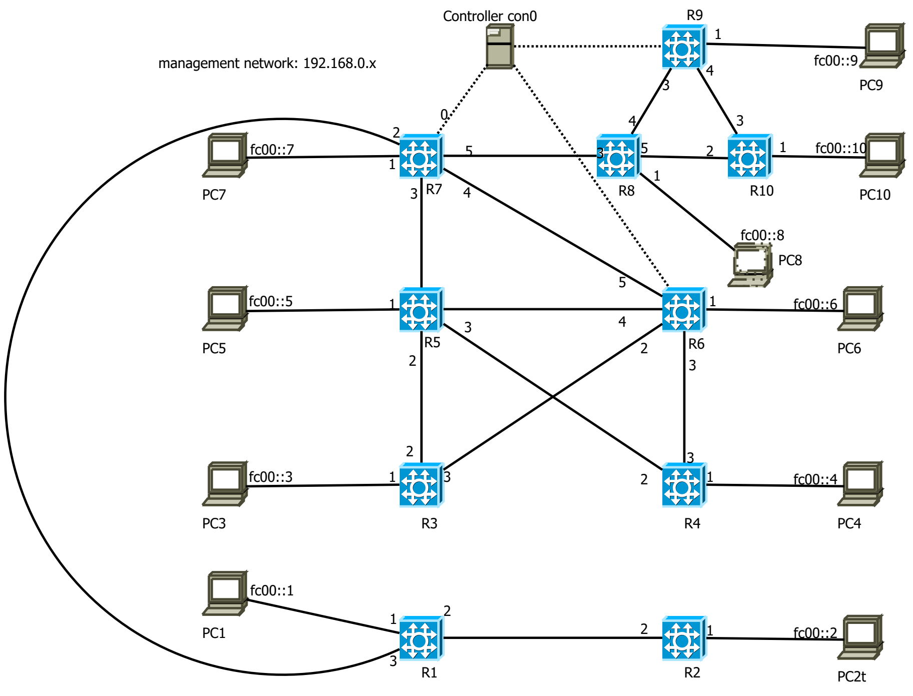


Figure 4.2.: Tested topology 2.

4.2. Experiment parameters

In order to find out significant differences between MEADcast in SDN and unicast in SDN, it is important to first look at a variety of available parameters and discuss whether they are relevant for the main goal or not.

Number of distinct MEADcast sessions

The number of distinct MEADcast sessions can have an influence on the overall performance of MEADcast. Depending on the implementation the controller might get overwhelmed if it has to process a high quantity of MEADcast packages. Possible observations of this could include latency or speed issues when it comes to the data delivery via MEADcast. This parameter however is not included in the experiments, because it might take a large number of simultaneous sessions to show a significant change in performance.

Amount of differing topologies tested

The keyword here is differing. Topologies itself can have a lot of parameters themselves, that when changed have a massive impact on the performance. Size, amount of controllers, amount of connections between switches or even the physical distance of the controller to the SDN are parameters that change how MEADcast or unicast perform on the topology. Therefor, this is a important parameters that can be included.

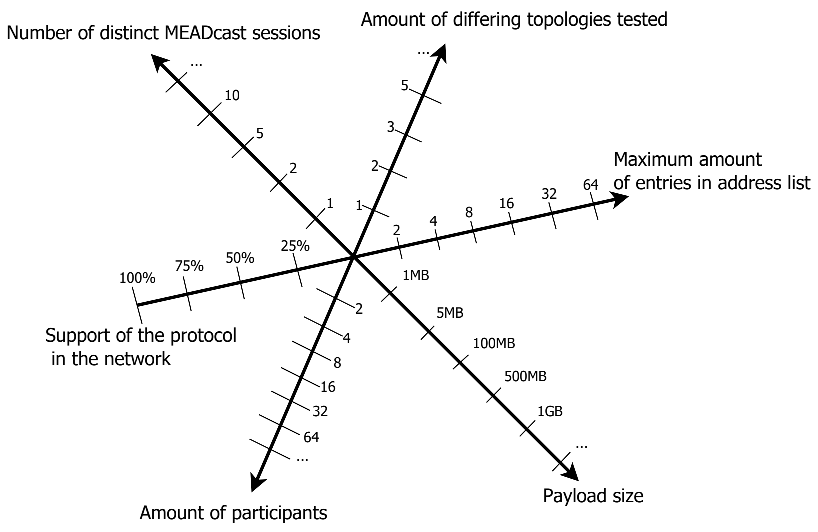

Number of distinct MEADcast sessions
...
10
5
2
1
Amount of differing topologies tested
...
5
3
2
1
Maximum amount of entries in address list
2
4
8
16
32
64
Support of the protocol in the network
100%
75%
50%
25%
Payload size
1MB
5MB
100MB
500MB
1GB
...
Amount of participants
2
4
8
16
32
64
...

Figure 4.3.: A selection of possible testable experiment parameters.

Maximum amount of entries in address list

The number of destinations can also affect the overall performance of the protocol since changing the amount of entries also changes the remaining available space for the payload. The path maximum transmission unit for example could limit the size of the MEADcast packet.

If the packet size itself is limited to a smaller size, it might not make sense to take address the full 64 receivers. As more entries get added to the MEADcast header, the IP destination address fields reserve a higher percentage of the maximum packet size. This leaves less space for the payload and might lead to high header overhead. It is a experiment parameter that can be used to directly influence the performance of the protocol and show how it might perform under non ideal situations.

Payload size

Payload size can be used in different ways. If experiments are conducted with varying payload sizes it is possible to directly measure the impact the of itself on MEADcast and unicast. If used as a constant it is helpful in determining changes of other variables. An example for that would be to send a 500 MB file from one sender to 10 receivers via MEADcast and unicast and then measure the amount of time it takes for both to finish sending the file.

Amount of participants

There can be a small or large amount of entities that take part of the experiment. This affects the performance of both MEADcast and unicast and can show if one of them gets affected more by an increased amount of participants than the other. Since this is one of the easiest experiment parameters to change, it is included in the experiment.

Support of the protocol in the network

This is a good parameter to see if the percentual increase in network support for MEADcast results in the same percentual increase or decrease of performance.

Bandwidth load of the link to the controller

Albeit very specific this is one of the most important ones. Since the controller handles every MEADcast packet in the network it needs to be able to handle all incoming and outgoing MEADcast traffic. The effects of this parameter can be observed by either reducing or increasing the bandwidth of the controller itself or by increasing the MEADcast traffic and as a result the load on the link.

4.3. Design of the Experiments

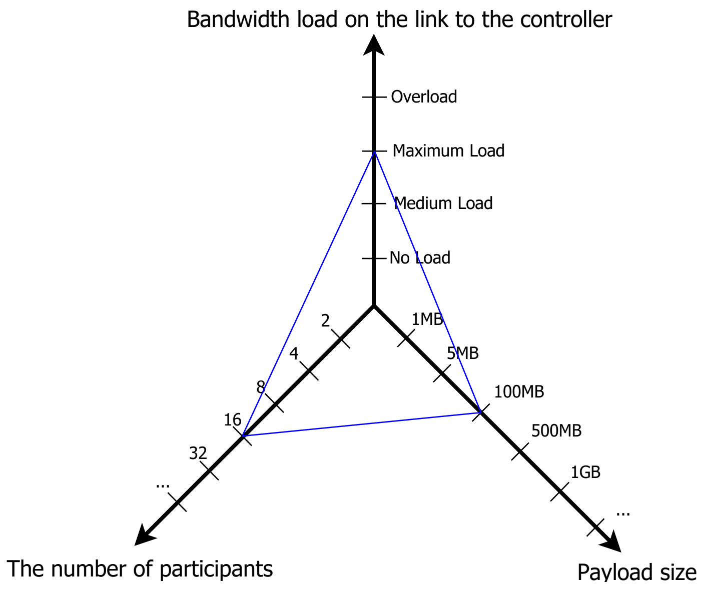


Figure 4.4.: Selected parameters for the experiments.

The experiments have been performed within the space of the selected parameters. Both unicast and MEADcast have been tested with different payload sizes and on topologies with different amount of participants. The standard speed at which the sender sends out his data is 5 KB/s. Different values however are tested to evaluate the reliability of MEADcast under heavy load compared to unicast. In each experiment PC1 will act as the sender and all remaining end hosts as receivers.

The following experiments are performed on each topology:

1. A 10 MB file is sent from the sender to every receiver in the topology via MEADcast.
2. A 100 MB file is sent from the sender to every receiver in the topology via MEADcast.
3. A 10 MB file is sent from the sender to every receiver in the topology via unicast.

4. A 100 MB file is sent from the sender to every receiver in the topology via unicast

In addition, each one of those experiments is tested with different transfer speeds. 5 KB/s was chosen as the base speed and the speed was adjusted up or down until the MEADcast implementation did not drop any more packets. This was done in order to have comparable results. Topology 1 with four end hosts got tested with transfer speeds of 4.1 KB/s, 5 KB/s, 6.8 KB/s and 10.2 KB/s. The experiments on topology 2 with nine receivers were conducted with two different speed values: 5 KB/s and 4.1 KB/s, resulting in a total of eight experiments for topology 2. The values for each topology were chosen based on the base value of 5 KB/s. If the controller did not drop packets at 5 KB/s the speed would get increased until packet loss started occurring. If the other case was the true then the speed was decreased until no more packet loss could be observed.

The experiments are also conducted with varying number of receivers. The experiments with two and four receivers were done on the first topology, while the experiments with six and nine receivers were carried out on topology 2. The two receivers in topology 1 are: PC1 and PC2. The six receivers in topology 2 are: PC2, PC3, PC4, PC5, PC6 and PC7.

There are three variables that are worth observing:

- The amount of time t it takes for every receiver in the network to finish receiving the data
- The total traffic volume v on the network
- The amount of packet loss occurred pl for each experiment

How each of them is measured depends on if the data was sent via MEADcast or unicast.

The time t is measured by comparing the time from when the sender first starts sending out the data to the time when every receiver has finished receiving the file.

The volume v is measured with the help of wireshark. Wireshark provides us the information about the path the packet takes to reach the destination and the packet sizes. Calculating v for unicast is fairly simple. We observe the path the packet takes from the sender to the receiver and count the amount of hops h it takes to reach the destination. This then gets multiplied with the sum of all packet sizes sent out ps and finally summed up for each receiver r.

$$v = \sum_{r=1}^{r} (h * \sum ps) \tag{4.1}$$

The traffic volume of MEADcast is calculated slightly different, the additional overhead of the MEADcast protocol header and OpenFlow header have to be taken into consideration. Since the discovery phase only takes up a small amount of total volume it will be ignored. The steps and headers for the transmission of a single packet are as follows:

1. Sender S sends out a MEADcast data delivery packet M with the data d to the router R. The packet can be depicted as: *M(d)*
2. R can not process the MEADcast packet and encapsulates the packet with an OpenFlow header OF. R proceeds to send it to the controller C for further processing. The current packet: *OF(M(d))*.

3. C receives the OpenFlow packet and handles the MEADcast packet inside it. C then sends out an OpenFlow packet containing a unicast packet U with the data to the switch that is responsible for the end host it is responsible for.the unicast packet *(U(d)* is a simple IPv6 packet containing the data from the MEADcast packet. The OpenFlow packet contains instructions for the switch on which port it needs to forward the packet to. The packet from C to the responsible switch looks like this: *OF(U(d))*.
4. The responsible switch receives *OF(U(d))* and forwards the data in that OpenFlow packet, in this the unicast packet *U(d)* according to the instructions in the OpenFlow header.
5. The receiver receives *U(d)* and has no problems processing it, since it is an IPv6 packet.

This is specific to the topologies used in this thesis, as the sender is only one hop away from the border switch.

The total traffic for the MEADcast experiments is calculated as follows:

$$v = p1 + \sum_{x=1}^{x} r * p2 \tag{4.2}$$

where :

- x stands for the amount of packets sent from the sender to the router
- p1 stands for the sum of the packet sizes from step 1 and 2
- r stands for the amount of receivers specified in the MEADcast packet
- p2 stands for the sum of the packet sizes from step 3 and 4

Packet loss *pl* is measured by summing up the amount of packets received by the receiver *pr* and dividing it by the sum of packets sent by the sender *ps*

$$pl = \frac{pr}{ps} \tag{4.3}$$

5. Results and Evaluation

This chapter shows the results of the performed experiments and the evaluation of those results.

Table 5.1.: Time to finish transmitting file in seconds

Receivers	**File size**	**Speed**	**Unicast**	**MEADcast**	**Difference in %**
2	100 MB	4.1	480	162	296
4	10 MB	5	846	214	394
	100 MB	5	8658	2094	403
4	10 MB	5	846	214	394
	100 MB	5	8658	2094	403
4	10 MB	6.8	636	160	398
	100 MB	6.8	6293	1592	395
4	10 MB	10.2	21% packet loss	420	-
	100 MB	10.2	20% packet loss	4219	-
6	100 MB	4.1	1560	260	600
9	10 MB	4.1	260	2458	920
	100 MB	4.1	2701	25902	958
9	10 MB	5	23% packet loss	1893	-
	100 MB	5	22% packet loss	19201	-

Table 5.1 shows the amount of time it took for both unicast and MEADcast to finish transmitting a file. A file is considered transmitted once every receiver in the topology has received the file. The lower it takes to finish the better. *Receivers* is the amount of hosts that are addressed in the MEADcast header and *Speed* indicates the speed at which the sender sends out MEADcast packets in KB/s. *Difference in %* is calculated by dividing the speed of unicast by the speed of MEADcast. The packet loss is calculated with formula 4.3 and by counting the amount of packets received by each receiver and the amount of packets sent by the sender.

While MEADcast might look superior to unicast, that might not be the case for MEADcast in OpenFlow SDN. That very fact can be seen in Table 5.1. If the speed at which the sender is chosen high enough the controller won't be able to process all the packets which will in turn result in packet loss. As a result, there will be cases in this specific implementation where using unicast instead of MEADcast will improve reliability when it comes to packet transmission. Unicast is more reliable in every case where packet loss for MEADcast starts occurring. This means at around 9-10 KB/s for topology 1 and 4-5 KB/s for the topology 2.

There are several possible reasons why the packet loss is present under those conditions:

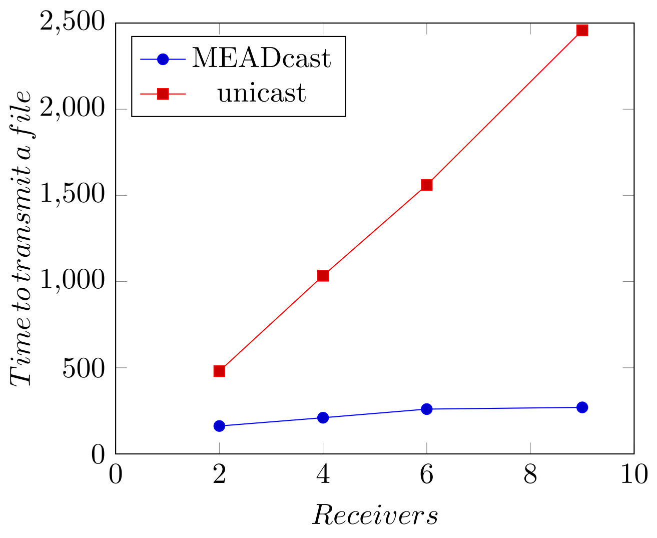


Figure 5.1.: Time it takes for MEADcast and unicast to finish transmitting a file.

- One that can never be ignored is the implementation of the MEADcast controller itself. Inefficient code may result in those packets being lost. The way to determine it, is to monitor where the packet loss occurs. If there is no packet loss between the sender and the controller, but packets get lost between the controller and the receiver, the fault most likely lies in the controller itself. In the case of this implementation, the packet loss occurs before the controller and thus the controller as the source of packet loss can be ruled out.

- UDP by design can result in packet loss. It is however impossible to use TCP as an alternative. MEADcast is a one-to-many communication and TCP requires a one-to-one handshake to synchronize sequence numbers to avoid packet loss. This can not be done with multiple receivers. That means that by using MEADcast packet loss is inevitable.

- The last and most likely reason could be the connection between the controller and the switches themselves. A feature of SDN is that the control plane is decoupled from the data plane, meaning that the controller itself can be physically disconnected from the switches. The only requirement is that they are able to communicate via IP. That means that the connection that was originally meant to be used for management only can be the cause of the lost packets, because it is being used to handle the MEADcast traffic of the entire network. This brings further implications in regards to other issues like practicality and reliability if this is the case. A heavy load could compromise the functionality of other tasks and applications, that rely on the connection between the controller and switch. There is no other way to implement MEADcast in SDN with OpenFlow, because OpenFlow and open vSwitch in their current form limit the capabilities of the switches.

On the contrary if MEADcast works it has performance advantages when compared to unicast. When comparing the observed performance indicators *time to finish transmitting the file* and *total traffic volume*, MEADcast performs better the more receivers are present.

The performance difference can be explained by the implementation and the design of the experiment itself. Since the topologies are deployed on virtual machines the latency between every switch and controller is almost non existent. This means, that a packet that would traditionally pass through several dozen switches in the topology via unicast can bypass that restriction with the use of MEADcast.

The transmission time for a MEADcast packet in this implementation can be described as:

$$Transmission\, time = \sum T_{psb} + T_{pbc} + T_{pcs} + T_{psr}$$

One MEADcast packet with n encoded receivers, the maximum value for n being 64 in this case, can send data to the intended destination in a maximum of a+n*2 hops, regardless of the internal SDN topology. The variable a is the amount of hops needed from the sender to the border switch. Unicast however has to traverse every hop in the topology to reach the receiver. This results in a scenario where MEADcast in SDN becomes more efficient the larger and more complex the topology becomes. There is a trade off however, this puts the majority of the load on the link between the controller and switches. Considering that all switches and the controller are deployed on the same physical machine as virtual machines, the latency between them is extremely low. As this might not be the case in practice, a more realistic experiment would have the controller be located in a physically different location, resulting in more realistic numbers.

Table 5.2.: MEADcast packet sizes for each hop in bytes

r	psb	pbc	pcs	psr	p1	p2	avg	uni
2	1174	1174	1192	1086	2456	2278	1204	1086
4	1214	1322	1192	1086	2536	2278	1476	1086
6	1246	1353	1192	1086	2600	2278	1184	1086
9	1302	1410	1192	1086	2712	2278	1220	1086

Another fact that can not be ignored is the speed at which the sender sends the packets out itself. Since the Figure 5.1 only shows the transmission times for the speed, at which the the MEADcast implementation does not drop packets, it is very biased towards MEADcast. Unicast is capable of sending data way above the speed at which no packet loss for MEADcast occurs. If the OpenFlow switch were able to handle MEADcast packets themselves, the aforementioned problems would not occur, because the controller would only be responsible for management activities. This would reduce the load on the link between the switches and the controller and would most likely eliminate the need to throttle the speed at the sender side.

Explanation of the column headers in Table 5.2:

- *r* stands for the amount of receivers in the topology.
- *psb* stands for the MEADcast packet sent from the sender to the border switch.
- *pbc* stands for the OpenFlow packet sent from the border switch to the controller.
- *pcs* stands for the OpenFlow packet sent from the controller to the switch responsible for the receiver.

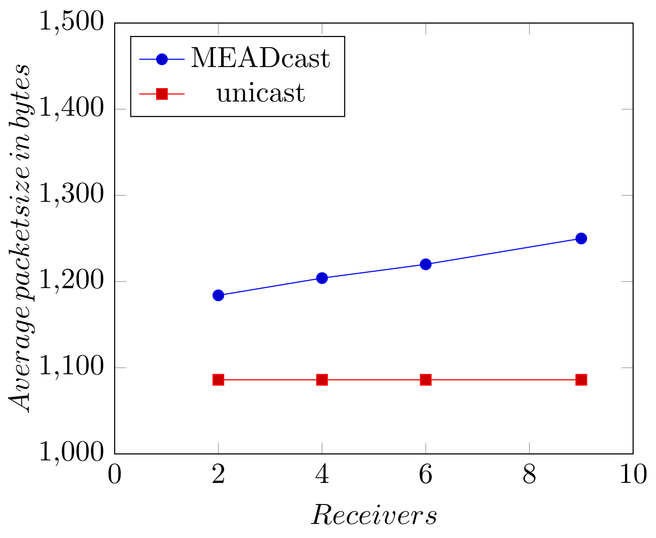


Figure 5.2.: Average packet sizes of MEADcast and unicast depending on the amount of receivers

- *psr* stands for the IPv6 packet sent from the switch to the receiver.
- *p1* the sum of psb and pbc.
- *p2* is the sum of pcs and psr.
- *avg* is the average of psb, pbc,pcs and psr. It is the average MEADcast packet size.
- *uni* is the unicast payload size in bytes. It is added for reference.

The variables r, p1 and p2 are used in formula 4.2 to calculate the traffic volume of the MEADcast experiments.

Table 5.3.: Total traffic volume on the network in MB

# of receivers	**File size**	**Unicast**	**MEADcast**	**Difference in %**
2	100	636	684	92
4	100	1484	1137	130
6	100	2545	1588	160
9	100	4030	2266	177

Difference in % is calculated by dividing the unicast traffic volume by the MEADcast traffic volume.

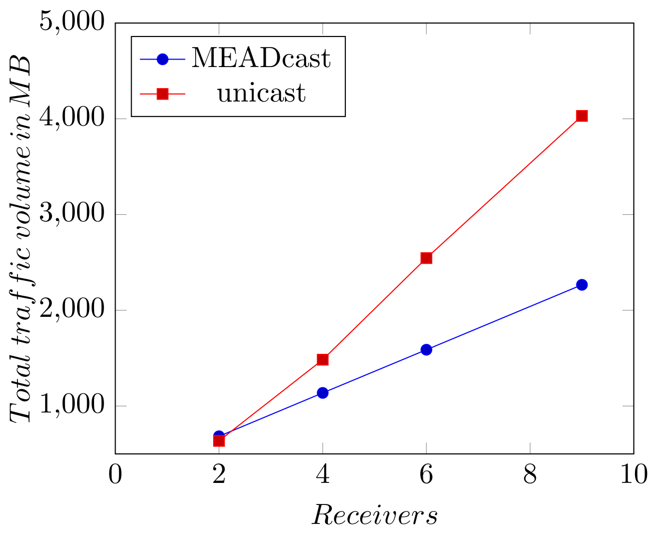


Figure 5.3.: Total traffic volume depending on the number of receivers.

Another performance indicator to look at is the total traffic volume as depicted in Table 5.3 or illustrated in Figure 5.3. This performance indicator is much more significant than the transmission time, because it does not get affected by the slower packet transmission speed. The only variables that affect this are the amount of hops it takes for the sender to reach each receiver and the amount of receivers encoded in the MEADcast header. This is important because the higher the amount of receivers, the more addresses need to be encoded in the header, resulting in a larger header size. This can be seen in Table 5.2 and Figure /refpacketsize, where the MEADcast has a higher overall packet size than unicast. Although MEADcast packets grow in size with larger number of receivers, it has barely any effect on the total traffic volume. The bigger and more complex the topology, the better MEADcast performs. The only exception to this are cases where the number of receivers is so small and the topology so simple, that using MEADcast would result in a higher total traffic volume. This can be seen in Table 5.3, where MEADcast performs worse than unicast. Since both MEADcast and unicast require a total of six transmission to transfer the data in this case, the larger packet size of MEADcast results in a higher total traffic volume.

Link stress is another factor. It can be defined as the number of packets with identical payload sent by a protocol over each link in the network from sender to the receiver. The average amount of link stress is naturally lower for MEADcast close to the sender. This is because MEADcast by design only duplicates packets when needed, instead of duplicating the packet for each receiver at the sender like unicast.

6. Conclusion and Future Work

6.1. Conclusion

Both Software-Defined Networks and MEADcast offer advantages that can't be overlooked. SDN decouping the data plane from the control plane enables easier deployment of applications on the control plane. This eases the implementation and evaluation of new protocols like MEADcast in SDN.

By implementing MEADcast in SDN and comparing it to unicast the advantages and disadvantages of MEADcast in OpenFlow SDN are shown. This was done by implementing both a MEADcast traffic generator and a controller capable of handling MEADcast traffic and deploying them both on a SDN.

While the transmission speed is a lot faster than unicast, this is only true when the speed at which the sender sends the data is throttled to a point, where the MEADcast implementation does not drop any packets. As the speed does not even break 10 KB/s in a topology with as little as four receivers, the practicality of MEADcast in current OpenFlow SDN can be described as very limited.

The total traffic volume of MEADcast and unicast display results independent of transmission speed, latency and payload size. The results show, that MEADcast is much more efficient the more receivers each packet addresses and the more complex the network topology is. The only exception can be observed in the experiment with only two receivers, where the larger MEADcast packet size results in a higher total traffic volume.

6.2. Future Work

There is certainly still a lot of work to be had when it comes to MEADcast in SDN or multicast in SDN in general. The thesis only explored a small subset of possible parameters and left many options open. A direction this topic can take in the future is to explore different implementations of MEADcast in SDN. One could argue that using OpenFlow itself is too restrictive since manipulating the flow table and flow entries can only achieve so much in OpenFlow switches. There are more options left to explore however. P4 Programming [BDG$^+$14] for example takes a more direct approach on switches that could be beneficial for future MEADcast in SDN development. P4 is designed to program the switch behavior itself. Giving the switches more ways to interact with incoming packets can ease the work, the controller has to do in this implementation. This could also solve the other problems encountered during the deployment of MEADcast in OpenFLow SDN.

While testing the implementation on virtual machines is a step up from using network simulators like ns-2, it is still far from actual real machines. A possible continuation of this thesis could include the testing of the MEADcast protocol on real hardware. This could include having the controller be physically detached from the rest of the network or have a varying amount of MEADcast capable switches and routers in the network.

A. Appendix

A.1. Creating the test environment and topologies

This section describes the process how the topologies got set up for the thesis.

A.1.1. Topology

Xen Project[1] is used to create the topologies in a virtual environment. Each topology can be created by following the instructions in:
https://wiki.xenproject.org/wiki/Xen_Project_Beginners_Guide
To create a virtual machine:

```
xl create <config file>
```

An example controller config file could look like this:

```
# sample configuration
# Xen config for controller.cfg
 kernel = '/boot/vmlinuz-3.16.0-4-amd64'
 extra  = 'elevator=noop'
 ramdisk = '/boot/initrd.img-3.16.0-4-amd64'
 vcpus  = '1'
 memory = '256'
 maxmem = '384'
 root   = '/dev/xvda1 ro'

 disk   = [ 'file:/xen/domains/controller/disk.img,xvda1,w' ]

 name   = 'controller' #change this

# This pc has 2 interfaces, the first one is for the link to the host, the second
# Change accordingly
 vif = [
 'mac=00:16:3E:00:00:06, vifname=c_vif0, bridge=br_man',
 'mac=00:16:3E:00:67:01:, vifname=c_vif1, bridge=br_r1c',
]

 on_poweroff = 'destroy'
 on_reboot = 'restart'
 on_crash = 'restart'
```

It is now possible to access each via SSH from the host machine.

[1] https://xenproject.org/

A.1.2. Controller

To install Ryu on the controller machine, pip can be used:

```
pip install ryu
```

The default directory for Ryu is:

```
/usr/local/lib/python2.7/dist-packages/ryu/
```

and contains several example and learning ryu applications in the /app folder.

A.1.3. Switches

A good tutorial for installing openVswitch can be found here:
https://docs.openvswitch.org/en/latest/tutorials/faucet/
To install Open vSwitch on the machines meant to be used as switches, use the following command:

```
apt-get install openvswitch-switch
```

To create a new switch (bridge) named br0, the following command should be used:

```
ovs-vsctl add-br br0
```

To ease the readability of the switch datapath ID the following command can be used to change it:

```
ovs-vsctl set bridge br0 other-config:datapath-id=0000000000000001
```

To connect it to the controller:

```
ovs-vsctl set-controller br0 tcp:192.168.0.6:6633
```

192.168.0.6 is the IP adress of the controller in this case, 6633 the default TCP port for ryu controllers.
Depending on the amount of virtual interfaces on the switch, it might be needed to repeat the following command for each interface:

```
ovs-vsctl add-port br0 eth0
```

This adds the port eth0 to br0.
Sometimes it might be necessary to bring up the newly created interface:

```
ip link set br0 up
```

To set the used OpenFlow version to 1.3:

```
ovs-vsctl set Bridge br0 protocols=OpenFlow13
```

The following command can be used to see an overview of the bridges on the switch:

```
ovs-vsctl show
```

It should return something like this:

```
root@router3:~# ovs-vsctl show
be7b83da-88f5-46ce-aadf-e530a03c219d
    Bridge "br3"
        Controller "tcp:192.168.0.6:6633"
        fail_mode: standalone
        Port "br3"
            Interface "br3"
                type: internal
        Port "eth1"
            Interface "eth1"
        Port "eth2"
            Interface "eth2"
        Port "eth3"
            Interface "eth3"
    ovs_version: "2.6.2"
root@router3:~#
```

A.2. User Manual for Deploying and Testing MEADcast in SDN

This section shows the components used for the experiments in the thesis and how they are deployed on the virtual machines.

A.2.1. Components

These applications and files are the components deployed on the virtual machines and are to be used with python 2.7:

- mc_functions.py
- mc_file_sender.py
- mc_controller.py
- udp_file_sender.py
- udp_file_receiver5005.py and udp_file_receiver5006.py
- ndp.py

mc_functions.py is a python file that contains functions related to crafting MEADcast packets. It gets imported and used by both **mc_file_sender.py** and **mc_controller.py**

mc_file_sender.py is the application that generates traffic on the sender machine. It has two functions. The first one sends out MEADcast discovery request packets and the other one waits for MEADcast discovery response packets, generates the topology viewpoint and sends data to all the receivers.

mc_controller.py is an application that acts as a controller in a SDN. It only handles MEADcast packets and packets related to the Neighbor Discovery Protocol (NDP).

udp_file_sender.py is a simple application that sends a file to a list of receivers via unicast.

udp_file_receiver5005.py and **udp_file_receiver5006.py** are both applications that are run on the receiver machines. **udp_file_receiver5005.py** listens on UDP destination port 5005 and is used for MEADcast testing while **udp_file_receiver5006.py** uses UDP destination port 5006 and is used for unicast testing. **udp_file_receiver5006.py** is being used to avoid the flow installed on every switch, that sends every packet with UDP destination port 5005 to the controller. An alternative to avoid that flow, would be to delete every installed flow on all switches, before testing with unicast packets.

ndp.py is the NDP application written by Cuong Ngoc Tran and is used to transmit unicast packets. It acts as the controller and installs flows on the switches. It is used instead of **simple_switch_13.py** provided by RYU, because it can handle the loops in topology 2.

A.2.2. Sending and receiving MEADcast packets

To transmit data it is necessary to first run **mc_controller.py** on the controller machine:

```
ryu-manager mc_controller.py --observe-links
```

–observe-links is necessary for the controller, to view the links between the switches.

For every host that wants to receive the data sent via MEADcast, it is necessary to run this command on every receiving host machine:

```
python udp_file_receiver5005.py
```

This application will receive unicast packets and listens on UDP destination port 5005. It tracks the time it takes to finish the file transmission and the total size of all received packets at the end, and prints them out in the console.

It will look like this while receiving packets:

```
packet number: 1        with packet length: 1024
  total file size: 1024    current time: 0.0023552
packet number: 2        with packet length: 1024
    total file size: 2048    current time: 0.0042721
...
```

Once no more packets arrive it will save the file with a randomly generated name in a subfolder named /filedump

```
packet number: 1023        with packet length: 1024
   total file size: 1048576    current time: 15.3281741142
finished with: sxxbi.jpg
file can be found in filedump/sxxbi.jpg
```

For MEADcast file transmission, the latest current time printed is the time it took for the file transmission to finish.

The sender has to start **udp_file_sender.py** twice with different parameters. The first instance listens to discovery response packets and sends the data:

```
python mc_file_sender.py senddata < argument1> < argument2>
```

where $< argument1 >$ is the name of the file that is intended to get sent. $< argument1 >$ can be any file as long as it is in the same directory as the application. This instance will wait for discover response packets. It will build the topology viewpoint after every incoming discovery response packet and will start sending out the data, once 5 seconds have passed after the last discovery response packet.

The second argument $< argument2 >$ is the delay in seconds between between each packet that is being sent. It is being used to regulate the speed at which the sender sends out the packet. A value of 5 translates to 1 packet being sent every 5 seconds, while a value of 0.02 results in 50 packets being sent out per second.

Once the application has been started it will keep printing out:

```
 still empty, continue
```

as long as no MEADcast discovery response packets have been received.

To send the discovery request packets to each receiver:

```
python mc_file_sender.py senddisco < argument1>
```

where $< argument1 >$ is the amount of receivers that the sender wants to send discovery request packets to. Currently 2, 4, 6 and 9 can be chosen as arguments. 2 and 4 only work for topology 1, while 6 and 9 only work on topology 2.

To create a file with a size of 10MB:

```
 dd if=/dev/zero of=10MB.txt count=10 bs=1048576
```

To start sending the file *10MB.txt* from pc1 to 4 receivers, with a delay of 0.02 seconds, the commands on pc1 would look like this:

```
python mc_file_sender.py senddata 10MB.txt 0.02
python mc_file_sender.py senddisco 4
```

The first instance of mc_file_sender.py should print out something similar to this, whenever a new response packet is received.

```
 discovery response packet in
 ['2001:db8::7807', '2001:db8::2801', 2001:db8::3801']
 [1, 0, 0]
 [1, 0, 0]
 [0, 5005, 5005]
```

The internal topology view point will be updated with every new response packet and after a few seconds of not receiving any more packets the application will start sending out MEADcast packets.

A.2.3. Sending and receiving unicast packets

In order to send unicast packets, it is required to start **ndp.py** instead of **mc_controller.py** on the controller machine.

```
ryu-manager ndp.py --observe-links
```

To start the application that receives can receive data on the receiver host machines:

```
python udp_file_receiver5006.py
```

This application listens on UDP port 5006 for incoming packets and behaves the same way as the application that listens for MEADcast packets on port 5005. To measure the time it takes for finish the file transmission via unicast, it is necessary to wait until all receivers finished the file transmission. Once this has been done then, the time on all receivers is added together to get the end result.

An unicast file transmission to two receivers for example could yield:

```
packet number: 1023          with packet length: 1024
    total file size: 1048576     current time: 14.9236232882
finished with: snoos.jpg
file can be found in filedump/snoos.jpg
```

```
packet number: 1023          with packet length: 1024
    total file size: 1048576     current time: 15.0228192921
finished with: aliss.jpg
file can be found in filedump/aliss.jpg
```

The time it took to finish the transmission would be :
14.9236232882 s + 15.0228192921 s = 29.9464425803 s

To start sending unicast packets from the sender:

```
python udp_file_sender.py  < argument1> < argument2> < argument3>
```

The first argument $< argument1 >$ is the amount of receivers the sender wants to address. Available options are 2, 4, 6 and 9. 2 and 4 only work for topology 1, while 6 and 9 only work for topology 2.

The second argument $< argument2 >$ is the name of the file.

The third argument $< argument3 >$ is the delay in seconds between each packet that is being sent. It is being used to regulate the speed at which the sender sends out the packet. A value of 5 translates to 1 packet being sent every 5 seconds, while a value of 0.02 results in 50 packets being sent out per second.

If the sender wants to send the file *10MB.txt* to 4 receivers in topology 1, with a speed of 100 packets per second (delay of 0.01), the command would look like this:

```
python udp_file_sender.py 4 10MB.txt 0.01
```

A.2.4. Measuring the packet sizes and traffic volume

In order to find out the packet sizes between each link, any packet sniffer or monitoring application can be used. Wireshark was used in this thesis.

To install wireshark on your Ubuntu host machine:

```
sudo apt install wireshark
```

It might be required to update the APT package repository before with:

```
sudo apt update
```

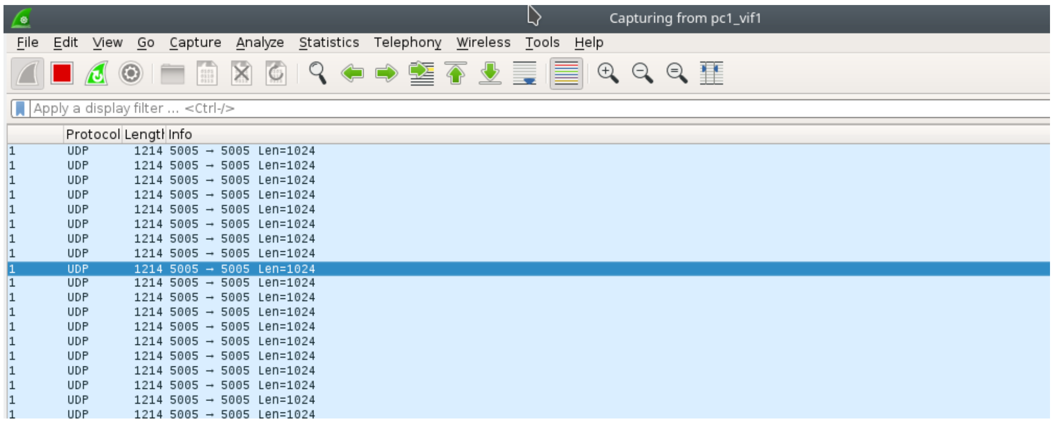
Wireshark must be started with root privileges:

```
sudo wireshark
```

After wireshark has been started, a list with all interfaces and links will show up. Double click the link that is to be monitored, for example pc1_vif1 for topology 1. This interface connects the sender pc (pc1) to the first switch. Wireshark will capture all the packets on pc1_vif1.

If for example

```
python udp_file_sender.py 4 10MB.txt 0.01
```

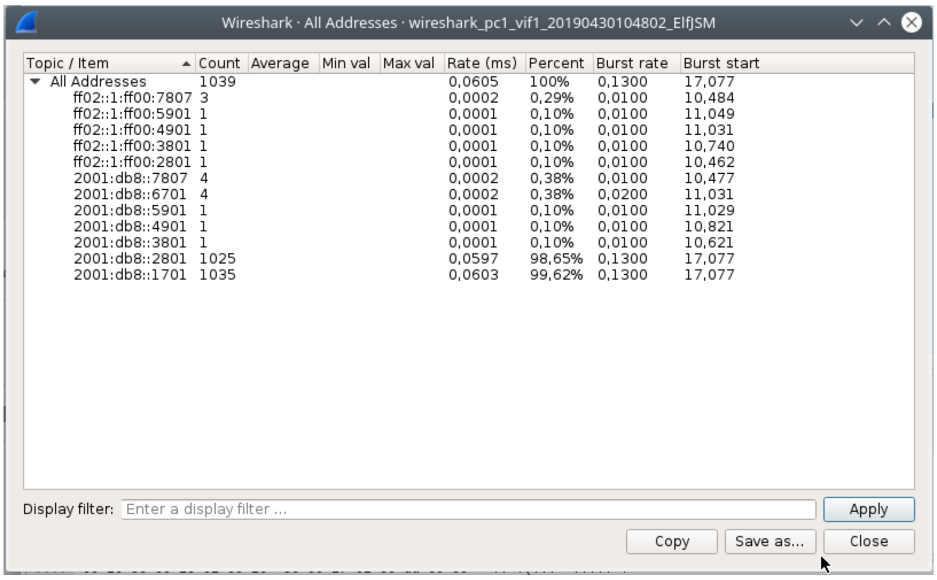

Capturing from pc1_vif1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
  Protocol Lengtl Info
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
1  UDP     1214 5005 → 5005 Len=1024
```

Figure A.1.: Wireshark packet capturing.

has been used to send MEADcast packets, the captured packets in wireshark will show up as depicted in (Figure A.1)

The length of the packet is 1214 in this case and the payload length is 1024. Additionally it is possible to measure the total amount of packets sent over that link by going to Statistics - Ipv6 Statistics - All Addresses in Wireshark. This can be used to calculate the total amount of traffic on the link. By observing all links that way it is possible to measure the total traffic volume.



Wireshark · All Addresses · wireshark_pc1_vif1_20190430104802_ElfJSM

Topic / Item	Count	Average	Min val	Max val	Rate (ms)	Percent	Burst rate	Burst start
All Addresses	1039				0,0605	100%	0,1300	17,077
ff02::1:ff00:7807	3				0,0002	0,29%	0,0100	10,484
ff02::1:ff00:5901	1				0,0001	0,10%	0,0100	11,049
ff02::1:ff00:4901	1				0,0001	0,10%	0,0100	11,031
ff02::1:ff00:3801	1				0,0001	0,10%	0,0100	10,740
ff02::1:ff00:2801	1				0,0001	0,10%	0,0100	10,462
2001:db8::7807	4				0,0002	0,38%	0,0100	10,477
2001:db8::6701	4				0,0002	0,38%	0,0200	11,031
2001:db8::5901	1				0,0001	0,10%	0,0100	11,029
2001:db8::4901	1				0,0001	0,10%	0,0100	10,821
2001:db8::3801	1				0,0001	0,10%	0,0100	10,621
2001:db8::2801	1025				0,0597	98,65%	0,1300	17,077
2001:db8::1701	1035				0,0603	99,62%	0,1300	17,077

Display filter: Enter a display filter ...

Figure A.2.: Wireshark traffic volume on a single link

List of Figures

Bibliography

[BBK02] Banerjee, Suman ; Bhattacharjee, Bobby ; Kommareddy, Christopher: *Scalable application layer multicast.* ACM, 2002

[BDG$^+$14] Bosshart, Pat ; Daly, Dan ; Gibb, Glen ; Izzard, Martin ; McKeown, Nick ; Rexford, Jennifer ; Schlesinger, Cole ; Talayco, Dan ; Vahdat, Amin ; Varghese, George ; Walker, David: P4: Programming Protocol-independent Packet Processors. In: *SIGCOMM Comput. Commun. Rev.* 44 (2014), Juli, Nr. 3, 87–95. `http://dx.doi.org/10.1145/2656877.2656890`. – DOI 10.1145/2656877.2656890. – ISSN 0146–4833

[BF01] Boivie, Dr. Richard H. ; Feldman, Nancy: Small Group Multicast / Internet Engineering Task Force. Version: Februar 2001. `https://datatracker.ietf.org/doc/html/draft-boivie-sgm-02`. Internet Engineering Task Force, Februar 2001 (draft-boivie-sgm-02). – Internet-Draft. – Work in Progress

[BMK13] Bondan, Lucas ; Müller, Lucas F. ; Kist, Maicon: Multiflow: Multicast clean-slate with anticipated route calculation on OpenFlow programmable networks. In: *Journal of Applied Computing Research* 2 (2013), Nr. 2, S. 68–74

[CDK$^+$02] Cain, Brad ; Deering, Steve ; Kouvelas, Isidor ; Fenner, Bill ; Thyagarajan, Ajit: Internet group management protocol, version 3. 2002. – Forschungsbericht

[CDZ97] Calvert, Kenneth L. ; Doar, Matthew B. ; Zegura, Ellen W.: Modeling internet topology. In: *IEEE Communications magazine* 35 (1997), Nr. 6, S. 160–163

[Cis] *Software-Defined Networks and OpenFlow - The Internet Protocol Journal, Volume 16, No. 1.* `https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-59/161-sdn.html`,

[Dee89] Deering, Steve: *RFC 1112: Host extensions for IP multicasting.* 1989

[Enn06] Enns, Rob: NETCONF configuration protocol. 2006. – Forschungsbericht

[Fen06] Fenner, Bill: *Experimental Values In IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers.* RFC 4727. `http://dx.doi.org/10.17487/RFC4727`. Version: November 2006 (Request for Comments)

[Fou12] Foundation, Open N.: OpenFlow Switch Specificationl. 2012. – Forschungsbericht

[FRZ14] Feamster, Nick ; Rexford, Jennifer ; Zegura, Ellen: The Road to SDN: An Intellectual History of Programmable Networks. In: *SIGCOMM Comput. Commun. Rev.* 44 (2014), April, Nr. 2, 87–98. `http://dx.doi.org/10.1145/2602204.2602219`. – DOI 10.1145/2602204.2602219. – ISSN 0146–4833

[Fun12] Fundation, Open N.: Software-defined networking: The new norm for networks. In: *ONF White Paper* 2 (2012), S. 2–6

[GKP+08] Gude, Natasha ; Koponen, Teemu ; Pettit, Justin ; Pfaff, Ben ; Casado, Martín ; McKeown, Nick ; Shenker, Scott: NOX: towards an operating system for networks. In: *ACM SIGCOMM Computer Communication Review* 38 (2008), Nr. 3, S. 105–110

[HD98] Hinden, Bob ; Deering, Dr. Steve E.: *Internet Protocol, Version 6 (IPv6) Specification.* RFC 2460. `http://dx.doi.org/10.17487/RFC2460`. Version: Dezember 1998 (Request for Comments)

[HH06] H. Holbrook, Inc. B. Cain Acopia Networks B. Haberman JHU A. Arastra: Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast. 2006. – Forschungsbericht

[HPD+15] Haleplidis, Evangelos ; Pentikousis, Kostas ; Denazis, Spyros ; Salim, Jamal H. ; Meyer, David ; Koufopavlou, Odysseas: *Software-Defined Networking (SDN): Layers and Architecture Terminology.* RFC 7426. `http://dx.doi.org/10.17487/RFC7426`. Version: Januar 2015 (Request for Comments)

[IKM14] Iyer, Aakash ; Kumar, Praveen ; Mann, Vijay: Avalanche: Data center multicast using software defined networking. In: *2014 sixth international conference on communication systems and networks (COMSNETS)* IEEE, 2014, S. 1–8

[KM05] In: Katrinis, Kostas ; May, Martin: *11. Application-Layer Multicast.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–32047–0, 157–170

[KRV+15] Kreutz, Diego ; Ramos, Fernando M. ; Verissimo, Paulo ; Rothenberg, Christian E. ; Azodolmolky, Siamak ; Uhlig, Steve: Software-defined networking: A comprehensive survey. In: *Proceedings of the IEEE* 103 (2015), Nr. 1, S. 14–76

[Lim12] Limoncelli, Thomas A.: Openflow: a radical new idea in networking. In: *Queue* 10 (2012), Nr. 6, S. 40

[LLT+17] Lin, Ying-Dar ; Lai, Yuan-Cheng ; Teng, Hung-Yi ; Liao, Chun-Chieh ; Kao, Yi-Chih: Scalable multicasting with multiple shared trees in software defined networking. In: *Journal of Network and Computer Applications* 78 (2017), S. 125–133

[MVTG14] Medved, Jan ; Varga, Robert ; Tkacik, Anton ; Gray, Ken: Opendaylight: Towards a model-driven sdn controller architecture. In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014* IEEE, 2014, S. 1–6

[OFI$^+$07] Ooms, Dirk ; Feldman, Nancy ; Imai, Yuji ; Livens, Wim P. ; Boivie, Dr. Richard H.: *Explicit Multicast (Xcast) Concepts and Options*. RFC 5058. `http://dx.doi.org/10.17487/RFC5058`. Version: November 2007 (Request for Comments)

[ONF12] ONF: Software-Defined Networking: The New Norm for Networks / Open Networking Foundation. Version: April 2012. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`. 2012. – Forschungsbericht

[RNPCE$^+$15] Rodriguez-Natal, Alberto ; Portoles-Comeras, Marc ; Ermagan, Vina ; Lewis, Darrel ; Farinacci, Dino ; Maino, Fabio ; Cabellos-Aparicio, Albert: LISP: a southbound SDN protocol? In: *IEEE Communications Magazine* 53 (2015), Nr. 7, S. 201–207

[SHYC15] Shen, Shan-Hsiang ; Huang, Liang-Hao ; Yang, De-Nian ; Chen, Wen-Tsuen: Reliable multicast routing for software-defined networks. In: *2015 IEEE Conference on Computer Communications (INFOCOM)* IEEE, 2015, S. 181–189

[SNNS07] Simpson, William A. ; Narten, Dr. T. ; Nordmark, Erik ; Soliman, Hesham: *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. `http://dx.doi.org/10.17487/RFC4861`. Version: September 2007 (Request for Comments)

[TD18] Tran, Cuong N. ; Danciu, Vitalian: Privacy-preserving multicast to explicit agnostic destinations. In: *The Eighth International Conference on Advanced Communications and Computation (INFOCOMP 2018)*, IARIA XPS Press, 2018, S. 60–65