HSR Cloud Infrastructure Lab 9

Software Defined Network

Emanuel Duss, Roland Bischofberger

2014-12-04

Inhaltsverzeichnis

1 Häufig benutzte Befehle

Folgende Commands haben wir oft genutzt:

Docker Daemon starten:

```
sudo systemctl start docker
```

Open Virtual Switch Daemon starten:

```
sudo systemctl start ovs-vswitchd
```

Mininet starten (ein Switch, 3 Hosts mit einem Controller):

```
sudo mn --topo single,3 --mac --controller remote
```

Switch `s1` soll ein OpenFlow Controller sein:

```
sudo ovs-vsctl set bridge s1 protocols=OpenFlow13
```

Docker Images anzeigen:

```
sudo docker images
```

Docker laufende Instanzen anzeigen

```
sudo docker ps
```

Docker Container Committen

```
sudo docker commit <ID>
```

Docker Images starten:

```
sudo docker run -ti -p 6633:6633 9dd865973dcc bash
```

Bestehende Flows auf dem Switch `s1` löschen:

```
sudo ovs-ofctl --protocols=OpenFlow13 del-flows s1
```

OpenFlow Controller RYU starten:

```
ryu-manager --verbose ryu/app/hub.py
```

Host `h1` pingt Host `h2` in Mininet:

```
mininet> h1 ping h2
```

Alle Hosts pingen sich gegenseitig?

```
mininet> pingall
```

2 Implementing a simple Hub

2.1 Ziel

Ein Hub, dessen Traffic immer über den Controller geht.

2.2 Implementation

In diesem Fall haben wir einfach im PacketInHandler jedes mal wenn ein Paket reinkam, die Actions auf Flood gesetzt und somit dem Switch mitgeteilt, dass er flooden soll.

2.3 Code

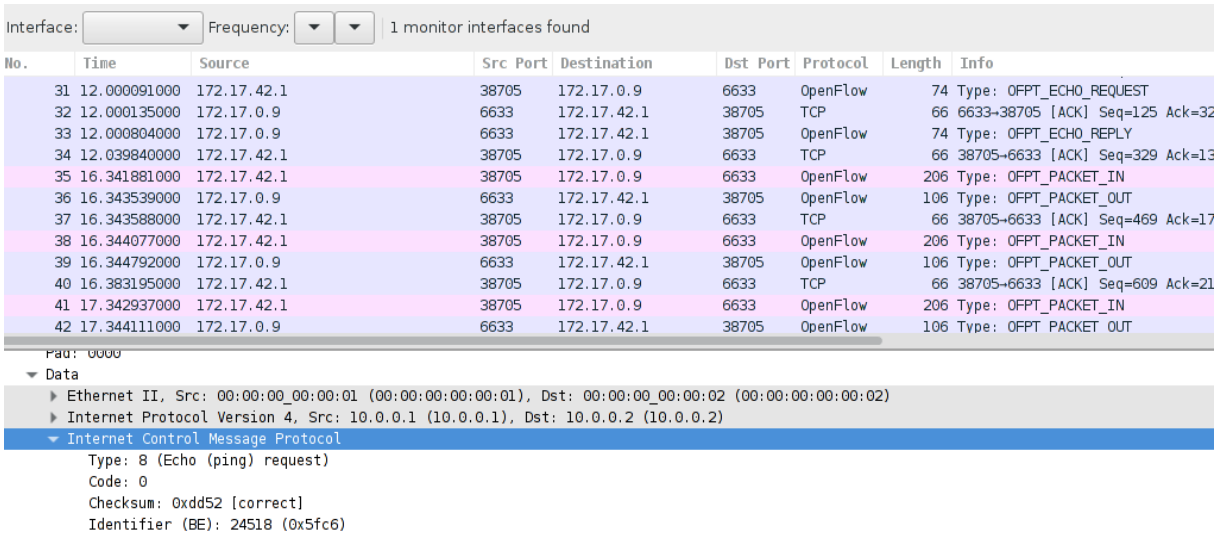
Code von `hub.py`

```
#!/usr/bin/env python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleHub13(app_manager.RyuApp):
        OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

        def __init__(self, *args, **kwargs):
                super(SimpleHub13, self).__init__(*args, **kwargs)

        @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
        def switch_features_handler(self, ev):
                datapath = ev.msg.datapath
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                match = parser.OFPMatch()
                actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
                self.add_flow(datapath, 0, match, actions)

        def add_flow(self, datapath, priority, match, actions):
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
                mod = parser.OFPFlowMod(datapath=datapath, priority=priority, match=match, instructi
                datapath.send_msg(mod)


        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def _packet_in_handler(self, ev):
                msg = ev.msg
                datapath = msg.datapath
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                in_port = msg.match['in_port']
```

```
out_port = ofproto.OFPP_FLOOD
actions = [parser.OFPActionOutput(out_port)]

data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id, in_port=in_port
datapath.send_msg(out)
dpid = datapath.id
self.logger.info("packet in %s %s", dpid, in_port)
```

2.4 Beschreibung

In Wireshark ist gut ersichtlich, dass der Traffic immer über den Controller fliesst. Jeder ARP Request und jedes ICMP Frame wird in ein OpenFlow Paket eingekapselt und durch den Controller verschickt. Der Hub lernt niemals eien Flow:



3 Implementing a Hub mit “abengespitztem” Flow

3.1 Ziel

Ein Hub, welcher den Traffic an den Controller schickt, falls er noch keinen Flow dafür hat.

3.2 Implementation

Bei diesm Hub wurde der Flow niedergeschrieben, dass jedes Paket einfach mit der Action flooden bahandelt werden soll.

Dieser Flow wurde jedoch erst beim ersten mal als ein Paket reinkam niedergeschrieben. Deshalb sieht man in der folgenden Pingübersicht, dass der erste Ping länger dauert, da der Flow noch nicht auf den Switch geschrieben wurde.

3.3 Code

Code von `hub_flood.py`:

```
#!/usr/bin/env python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleHub13(app_manager.RyuApp):
        OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

        def __init__(self, *args, **kwargs):
                super(SimpleHub13, self).__init__(*args, **kwargs)

        @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
        def switch_features_handler(self, ev):
                datapath = ev.msg.datapath
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                match = parser.OFPMatch()
                actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
                self.add_flow(datapath, 0, match, actions)

        def add_flow(self, datapath, priority, match, actions):
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
                mod = parser.OFPFlowMod(datapath=datapath, priority=priority, match=match, instructi
                datapath.send_msg(mod)


        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def _packet_in_handler(self, ev):
                msg = ev.msg
                datapath = msg.datapath
```

```
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']

out_port = ofproto.OFPP_FLOOD
actions = [parser.OFPActionOutput(out_port)]

match = parser.OFPMatch()
self.add_flow(datapath, 1, match, actions)

data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id, in_port=in_por
datapath.send_msg(out)
dpid = datapath.id
self.logger.info("packet in %s %s", dpid, in_port)
```

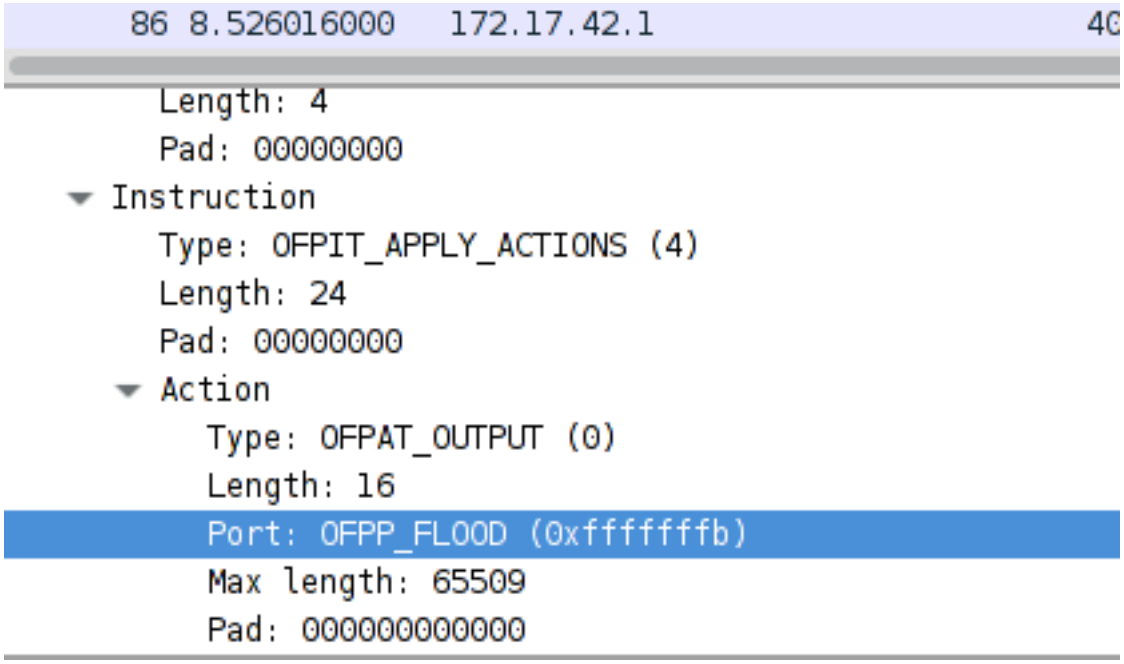
3.4 Test in Mininet

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=3.18 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.323 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.115 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.106 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.100 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3999ms
rtt min/avg/max/mdev = 0.100/0.765/3.183/1.211 ms
```

In der ICMP Sequenz 1 sieht man gut, dass die Round Trip Time höher war als bei den restlichen Pings, da der Flow erst nach diesem auf das Gerät beschrieben wurde. Danach werden die ICMP Pakete durch den Hub direkt an alle Ports gefloodet.

3.5 Beschreibung

In Wireshark ist gut ersichtlich, wie der Controller dem Hub sagt, er soll das Paket an alle Ports Flooden:



Nachfolgende Pakete gehen nicht mehr über den Controller, sondern werden direkt geflutet.

4 Implementing a Switch

4.1 Ziel

Ein Switch, welcher den Traffic an den Controller schickt, falls er noch keinen Flow dafür hat. Zusätzlich zum vorherigen Hub werden hier die MAC Adressen dem Port zugeordnet.

4.2 Implementation

Wir verwendeten eine Mac-zu-Port-Zuordnungsliste, damit sich der Controller merken kann hinter welchem Port sich eine Macadresse befindet und dies dann in einem Flow dem Switch mitteilt. Dies wurde bereits so im Handbuch von Ryu gelöst.

Auf eine genauere Beschreibung wird hier verzichtet, da der Multi Tenant Switch eine Erweiterung dieses Switches ist und dort auf die Details eingegangen wird.

4.3 Code

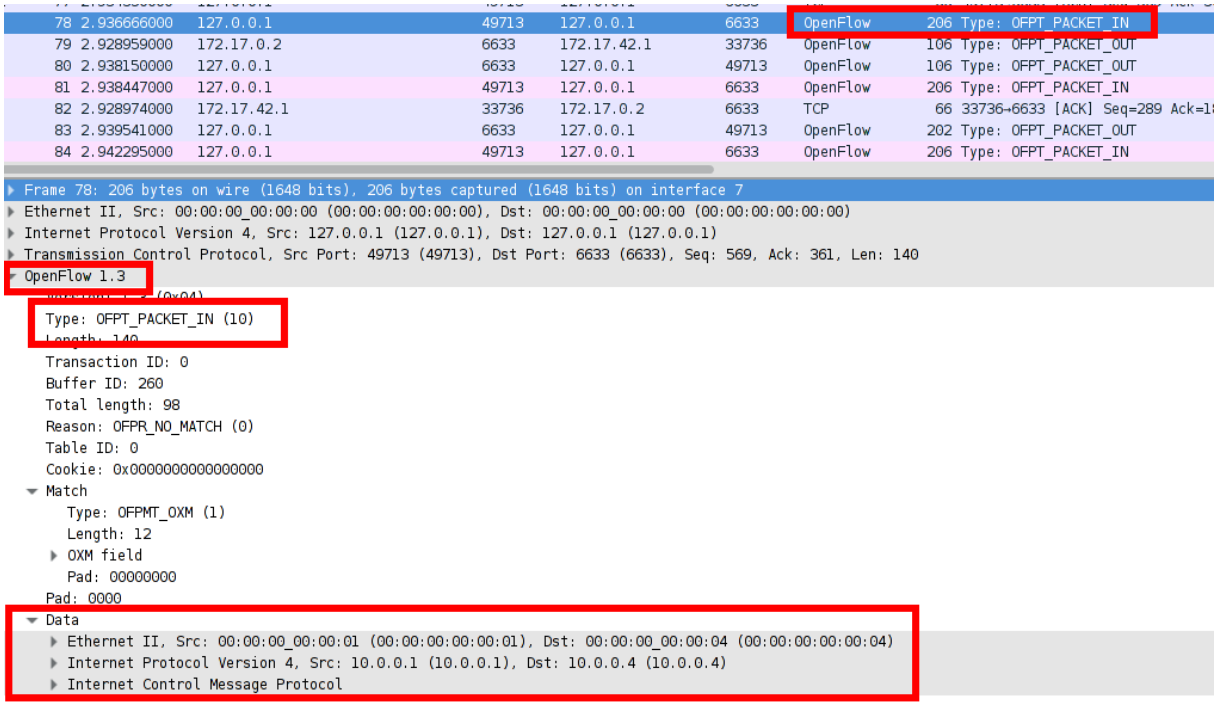
Code von `switch.py`:

```
#!/usr/bin/env python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleHub13(app_manager.RyuApp):
        OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

        def __init__(self, *args, **kwargs):
                super(SimpleHub13, self).__init__(*args, **kwargs)
                self.mac_to_port = {}

        @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
        def switch_features_handler(self, ev):
                datapath = ev.msg.datapath
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                match = parser.OFPMatch()
                actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
                self.add_flow(datapath, 0, match, actions)

        def add_flow(self, datapath, priority, match, actions):
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
                mod = parser.OFPFlowMod(datapath=datapath, priority=priority, match=match, instructi
                datapath.send_msg(mod)


        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def _packet_in_handler(self, ev):
```

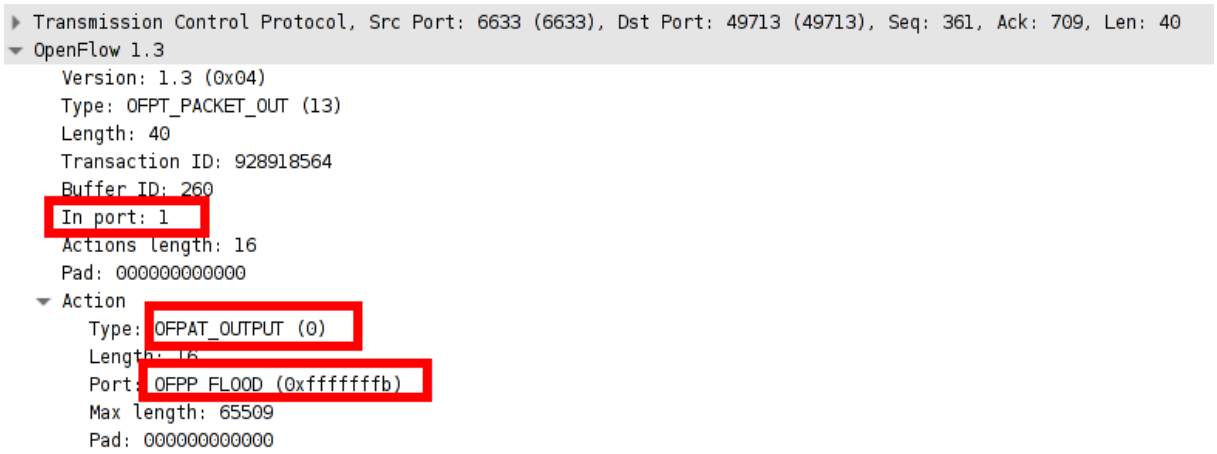
```
msg = ev.msg
datapath = msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']

pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]

dst = eth.dst
src = eth.src

dpid = datapath.id
self.mac_to_port.setdefault(dpid,{})

self.mac_to_port[dpid][src] = in_port # SRC Adresse und Port mappen

if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
else:
        out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]

# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id, in_port=in_por
datapath.send_msg(out)

self.logger.info("packet in %s %s", dpid, in_port)
```

4.4 Beschreibung

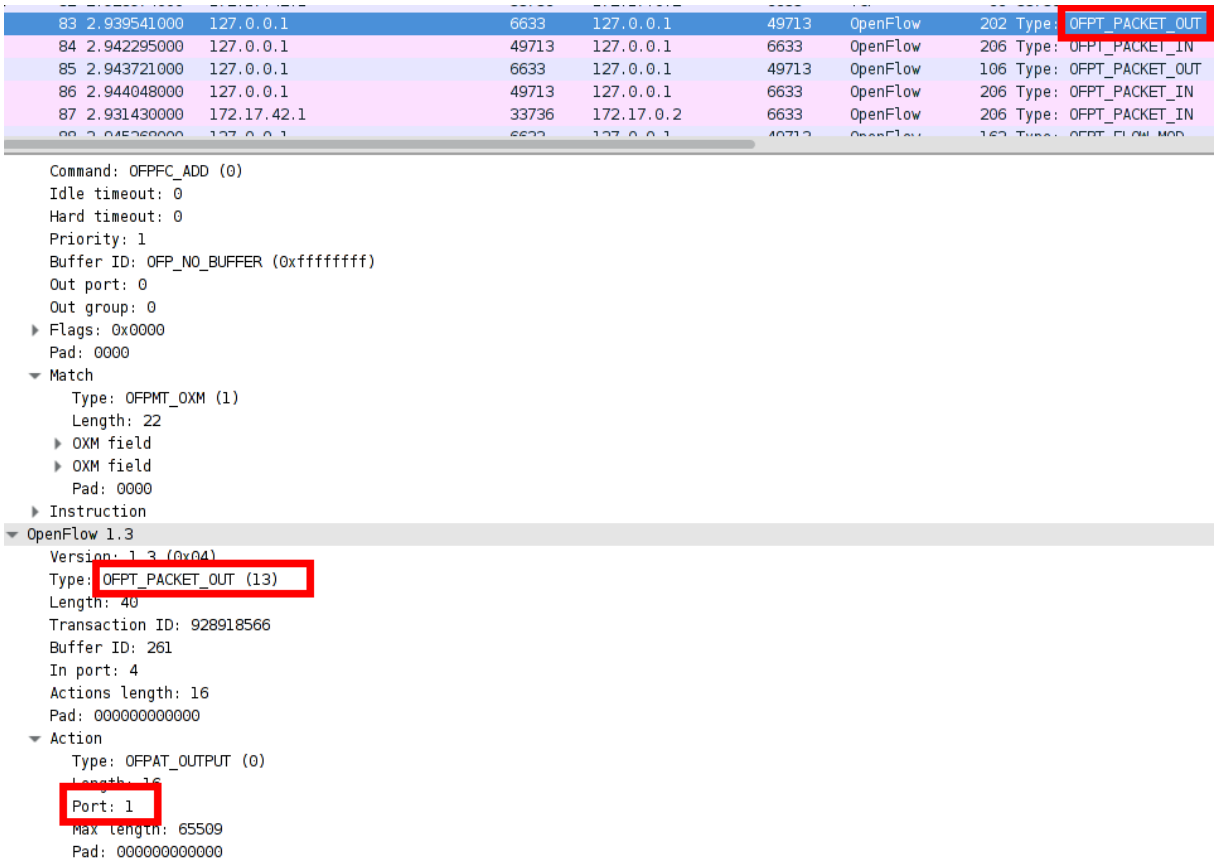
Das erste Paket geht über den Controller und wird in ein OpenFlow Header eingepackt (Hier im Wireshark Screenshot vom Host h1 zum Host h4):



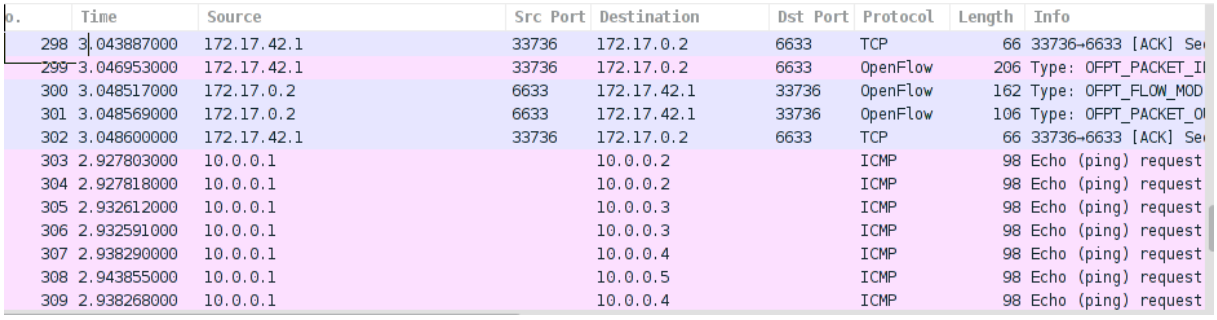
Der Switch sieht das Paket und weiss nicht an welchen Port es soll. Deshalb wird das Paket gefloodet:



Der Switch merkt sich die MAC Adresse vom Host 1. Beim ICMP Replay vom Host h4 kann die Antwort direkt auf den Port 1 ausgegeben werden. Zudem wird dies in einem Flow auf den Switch geschrieben:



Schlussendlich gehen die Pings direkt ohne über den Controller zu den Hosts:



5 Implementing a Policy Based Controller

5.1 Ziel

Auf dem Controller sollen Regeln geschrieben werden, welche definieren, welche Hosts mit anderen Hosts kommunizieren dürfen.

5.2 Implementation

Wir haben wie bereits beim einfachen Switch eine Mac-zu-Port-Zuordnungsliste verwendet.

5.3 Annahmen

Das erste Paket von einer Src-Adresse wird gedropt, da die Src-Adresse noch nicht in der Mac-zu-Port-Zuordnungsliste steht.

Wir nehmen jedoch an, dass ein Host der ans Netz kommt bereits viel Netzwerkverkehr verursacht und somit dem Switch die Mac zu Port Zuordnung bereits bekannt ist, wenn wichtiger Verkehr fliessen sollte.

5.4 Umsetzung

Um festzulegen welche Hosts zu welchen anderen Hosts Pakete senden dürfen, haben wir ein CSV File mit dem Namen serverlist.csv erstellt.

Dieses hat folgende Struktur:

```
HostSource;Liste mit Hosts an welche der HostSource Pakete senden darf.
```

Beispiel:

```
00:00:00:00:00:01;00:00:00:00:00:02;00:00:00:00:00:03;00:00:00:00:00:04
```

Somit darf Host 1 auf die Hosts 2, 3 und 4 zugreifen, jedoch nicht auf die Hosts 5 und 6

Dieses CSV lesen wir ein und speichern die Liste der Hosts jeweils in einem Dictionary mit dem Key der MacAdresse des Source Hosts.

Dies hier ist eine sinnbildliche veranschaulichung davon:

```
[00:00:00:00:00:01]{00:00:00:00:00:02,00:00:00:00:00:03,00:00:00:00:00:04}
```

Der Switch hat eine Liste die die Zuordnung von Mac-Adresse zu den Switchports macht.

Im PacketInHandler behandeln wir folgende Fälle:

- Ein Paket darf von der Src-Adresse an die Dst-Adresse gesendet werden und die Dst-Adresse ist nicht in der Mac-zu-Port-Zuordnungsliste –> Aktion: Drop
- Das Paket darf von der Src-Adresse an die Dst-Adresse gesendet werden und ist bereits in der Liste der bekannten Ports. –> Aktion: Forward to Port und Flow schreiben
- Ein Broadcast-Paket und dem Ethertype ARP kommt an –> Aktion: Flood und Flow schreiben
- Alle anderen Pakete werden gedropt und das gleich als Flow nieder geschrieben

Mininet wird wie folgt gestartet:

```
sudo mn --topo=single,6 --mac --controller remote,ip=127.0.0.1
```

Während den Arbeiten haben wir jeweils den geclonten Docker container committed, damit unsere Änderungen nicht verloren gehen.

5.5 Erweiterungsmöglichkeiten

Wir haben uns bewusst dafür entschieden, einen ARP Request an die Broadcastadresse an alle Hosts zu flooden. In einem weiteren Schritt könnte man dies auch noch unterbinden, indem man auflöst ob der PC hinter der IP Adresse wirklich angesprochen werden darf vom ARP-Request-Steller.

Auch möglich wäre es, die Konfiguration mit einem JSON zu machen und somit "Gruppen" zu ermöglichen, mit welchen man die Hosts übersichtlicher verwalten könnte.

5.6 Code

Inhalt von `serverlist.csv`:

```
00:00:00:00:00:01;00:00:00:00:00:02;00:00:00:00:00:03;00:00:00:00:00:04
00:00:00:00:00:02;00:00:00:00:00:01;00:00:00:00:00:03;00:00:00:00:00:04
00:00:00:00:00:03;00:00:00:00:00:01;00:00:00:00:00:02;00:00:00:00:00:04;00:00:00:00:00:05;00:00:00:0
00:00:00:00:00:04;00:00:00:00:00:01;00:00:00:00:00:02;00:00:00:00:00:03;00:00:00:00:00:05;00:00:00:0
00:00:00:00:00:05;00:00:00:00:00:03;00:00:00:00:00:04;00:00:00:00:00:06
00:00:00:00:00:06;00:00:00:00:00:03;00:00:00:00:00:04;00:00:00:00:00:05
```

Inhalt von `tierarch.py`:

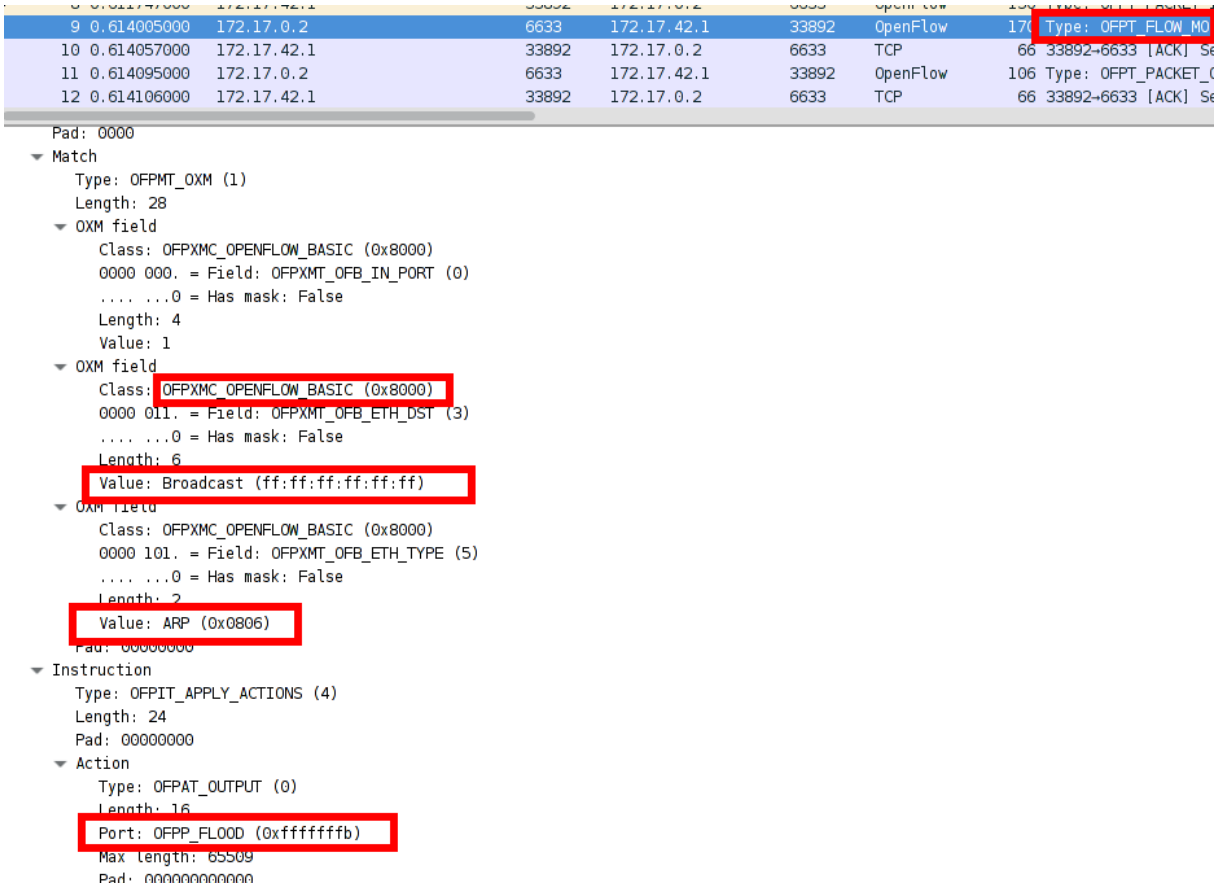
```
#!/usr/bin/env python

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import arp

import csv

class SimpleHub13(app_manager.RyuApp):
        OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
        serverlist = "serverlist.csv"
        acl = {}

        def __init__(self, *args, **kwargs):
                super(SimpleHub13, self).__init__(*args, **kwargs)
                self.mac_to_port = {}
                self.get_data()

        def get_data(self):
                with open(self.serverlist, 'rb') as f:
                        reader = csv.reader(f, delimiter=';')
                        for row in reader:
                                host = row[0]
                                allowed = row[1:len(row)]
                                self.acl[host] = allowed
                for i in self.acl:
                        print i
                        print self.acl[i]
```

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority, match=match, instructio
        datapath.send_msg(mod)


@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]
        pkt_arp = pkt.get_protocol(arp.arp)

        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid,{})

        self.mac_to_port[dpid][src] = in_port # SRC Adresse und Port mappen

        # self.logger.info(dst in self.acl[src])
        # self.logger.info("dst = %s / acl[src] = %s", dst, self.acl[src])

        actions = []
        tuhe = ""

        out_port = ""
        if dst in self.acl[src] and dst in self.mac_to_port[dpid]:
                out_port = self.mac_to_port[dpid][dst]
                actions = [parser.OFPActionOutput(out_port)]
                match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
                self.add_flow(datapath, 1, match, actions)
                tuhe = "[Write Flow] Allowed Traffic: Correct ACL and DST MAC on known Port.
        elif dst == "ff:ff:ff:ff:ff:ff" and pkt_arp:
                out_port = ofproto.OFPP_FLOOD
                match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_type=0x0806)
                actions = [parser.OFPActionOutput(out_port)]
                self.add_flow(datapath,1,match,actions)
                tuhe = "[Write Flow] Allowed Traffic: Broadcast ARP Request."
```

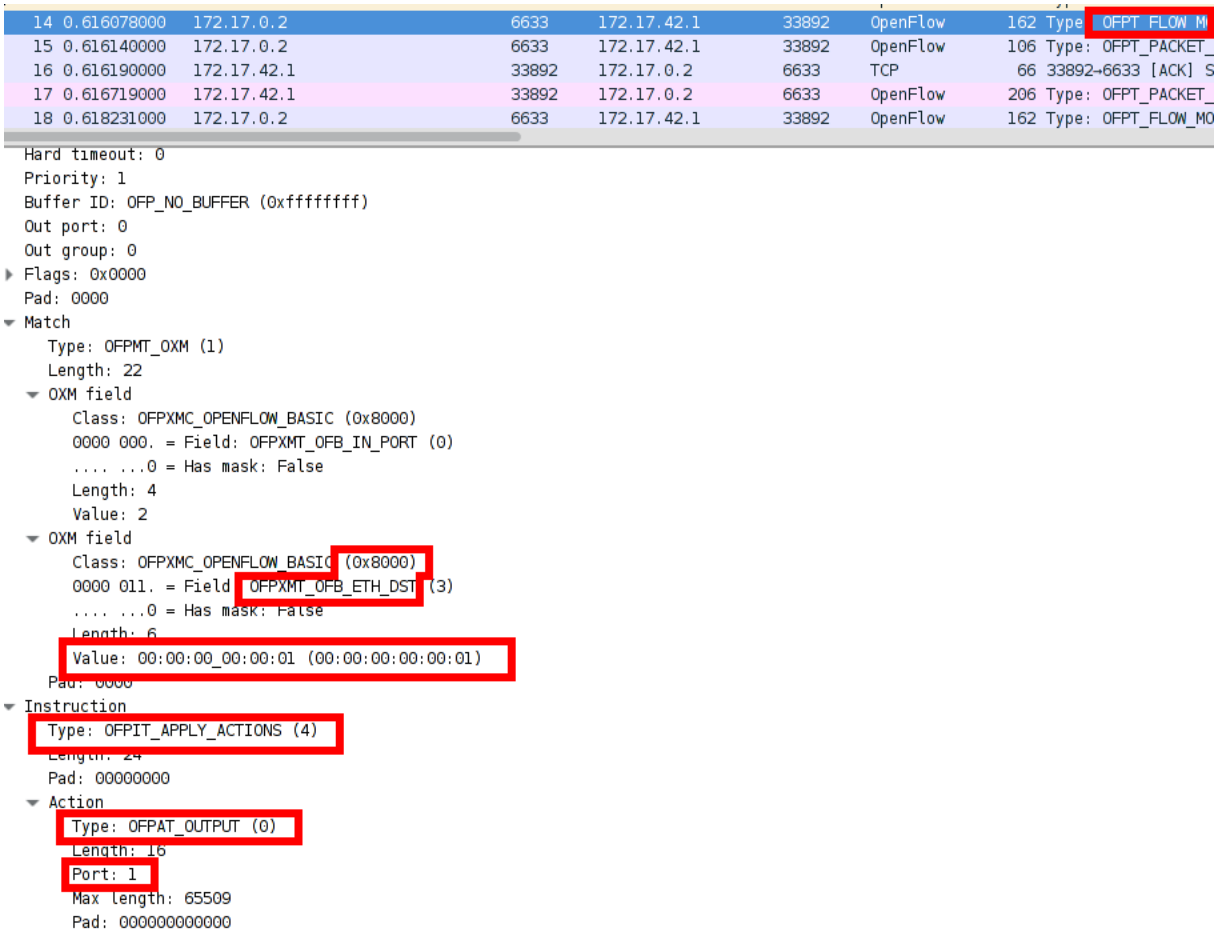
```
                elif dst not in self.acl[src]:
                        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
                        self.add_flow(datapath,1,match,actions)
                        tuhe = "[Write Flow] Deny Traffic; Wrong ACL"
                else:
                        tuhe = "[No Flow] Write no Flow"

                # install a flow to avoid packet_in next time
                # if out_port != ofproto.OFPP_FLOOD:

                data = None
                if msg.buffer_id == ofproto.OFP_NO_BUFFER:
                        data = msg.data

                out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id, in_port=in_por
                datapath.send_msg(out)

                # self.logger.info("packet in %s %s", dpid, in_port)
                self.logger.info("[*] From %s (Port %s) to %s (Port %s)", src, in_port, dst, out_por
                self.logger.info("    Action: %s", tuhe)

#if __name__ == '__main__':
        # get_data()
```

5.7 Beschreibung

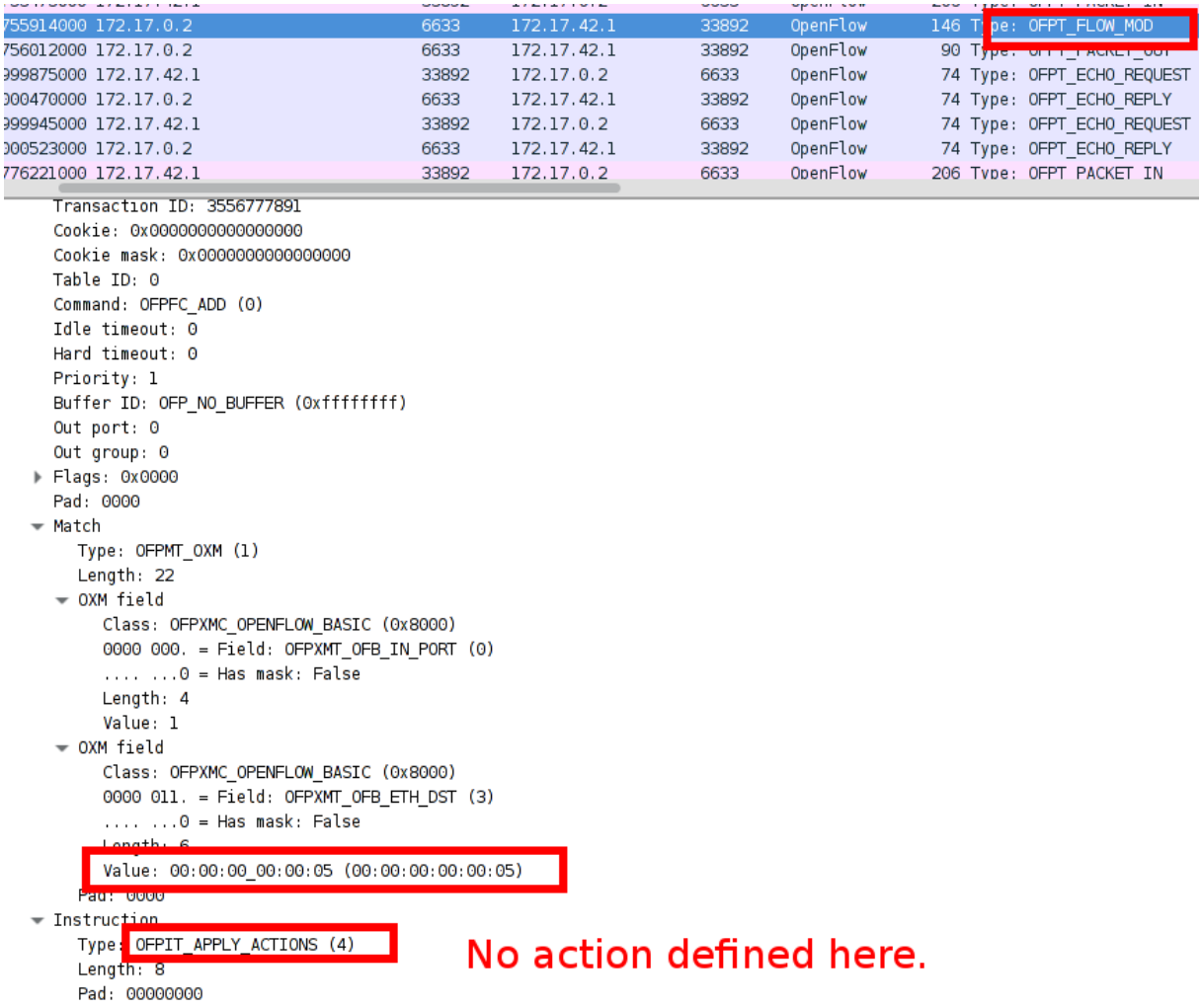
Bei den ersten Packeten handelt es sich um ein ARP Broadcast (Ethertype `0x0806`). Als Action wird dem Switch ein entsprechender Flow heruntergeschrieben (vgl. Punkt 3 bei der Beschreibung vom `PacketInHandler`):



Kontaktiert ein Host den Host h1, und es ist erlaubt (also h1, h2, h3, h4) wird folgender Flow abengespitzt (vgl. Punkt 2 bei der Beschreibung vom `PacketInHandler`):

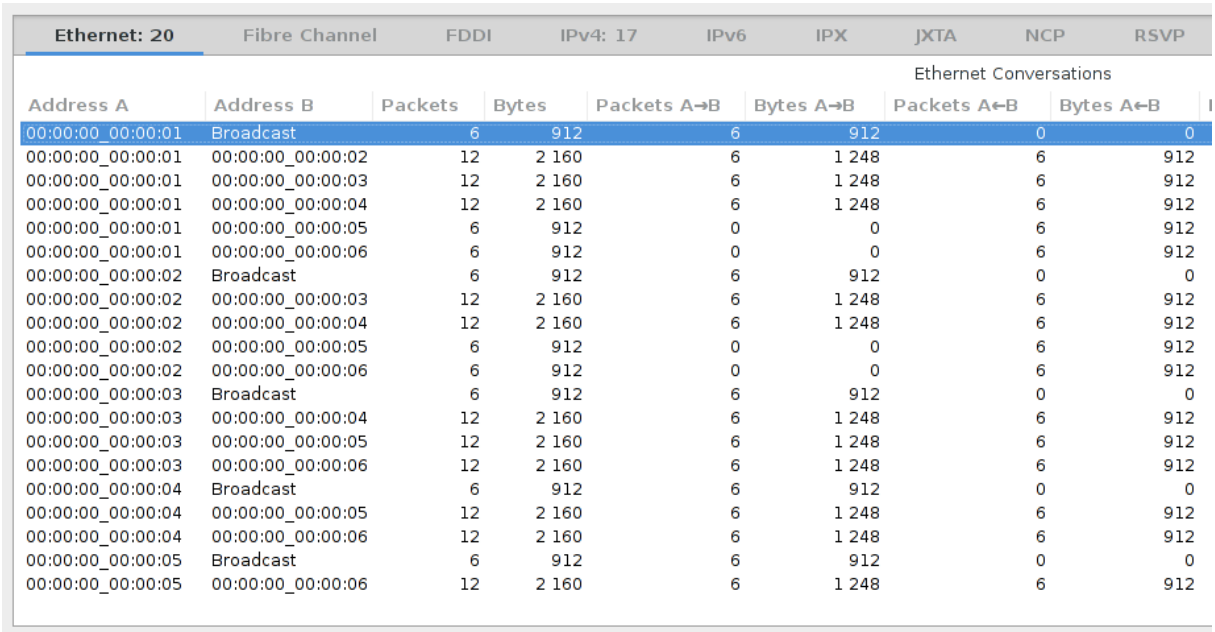


Kommt Netzwerkverkehr beim Cotroller vorbei, welcher nicht erlaubt wäre (beispielsweise zwischen Host h2 und Host h5), wird eine leere Action angewendet, was dem droppen eines Packets entspricht (vgl. Punkt 1 bei der Beschreibung vom `PacketInHandler`):



In der Conversation View von Wireshark ist zudem auch sehr gut ersichtlich, dass nur die erlaubten Hosts miteinander kommunizieren:

Layer 2:

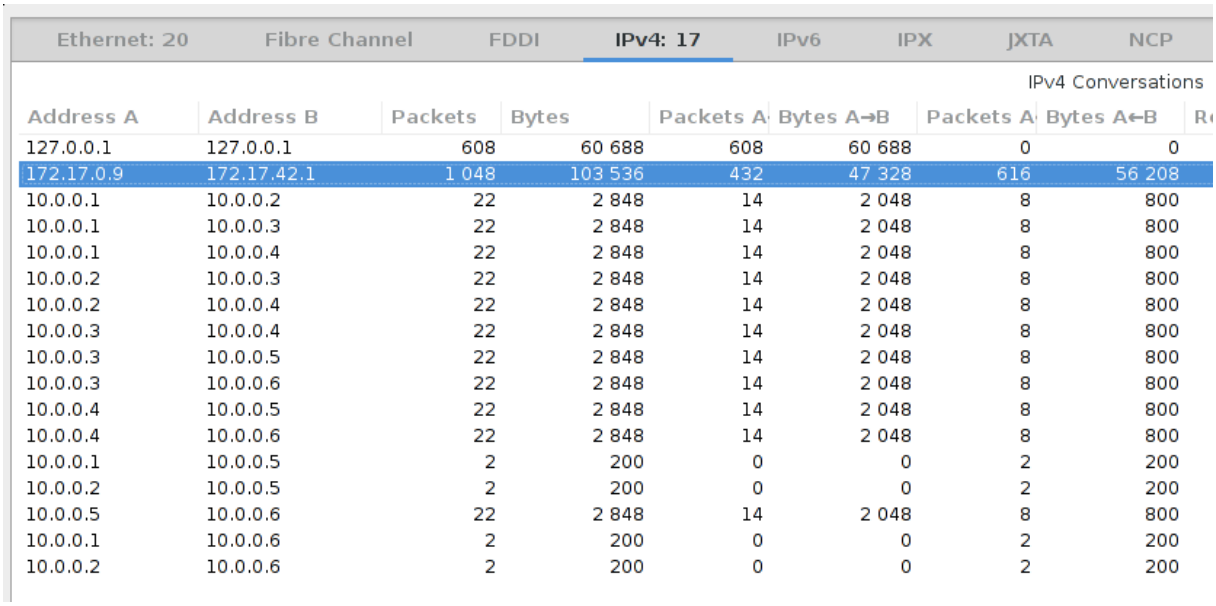


Ethernet: 20 | Fibre Channel | FDDI | IPv4: 17 | IPv6 | IPX | JXTA | NCP | RSVP

Ethernet Conversations

Address A	Address B	Packets	Bytes	Packets A→B	Bytes A→B	Packets A←B	Bytes A←B
00:00:00_00:00:01	Broadcast	6	912	6	912	0	0
00:00:00_00:00:01	00:00:00_00:00:02	12	2 160	6	1 248	6	912
00:00:00_00:00:01	00:00:00_00:00:03	12	2 160	6	1 248	6	912
00:00:00_00:00:01	00:00:00_00:00:04	12	2 160	6	1 248	6	912
00:00:00_00:00:01	00:00:00_00:00:05	6	912	0	0	6	912
00:00:00_00:00:01	00:00:00_00:00:06	6	912	0	0	6	912
00:00:00_00:00:02	Broadcast	6	912	6	912	0	0
00:00:00_00:00:02	00:00:00_00:00:03	12	2 160	6	1 248	6	912
00:00:00_00:00:02	00:00:00_00:00:04	12	2 160	6	1 248	6	912
00:00:00_00:00:02	00:00:00_00:00:05	6	912	0	0	6	912
00:00:00_00:00:02	00:00:00_00:00:06	6	912	0	0	6	912
00:00:00_00:00:03	Broadcast	6	912	6	912	0	0
00:00:00_00:00:03	00:00:00_00:00:04	12	2 160	6	1 248	6	912
00:00:00_00:00:03	00:00:00_00:00:05	12	2 160	6	1 248	6	912
00:00:00_00:00:03	00:00:00_00:00:06	12	2 160	6	1 248	6	912
00:00:00_00:00:04	Broadcast	6	912	6	912	0	0
00:00:00_00:00:04	00:00:00_00:00:05	12	2 160	6	1 248	6	912
00:00:00_00:00:04	00:00:00_00:00:06	12	2 160	6	1 248	6	912
00:00:00_00:00:05	Broadcast	6	912	6	912	0	0
00:00:00_00:00:05	00:00:00_00:00:06	12	2 160	6	1 248	6	912

Layer 3:



Ethernet: 20 | Fibre Channel | FDDI | **IPv4: 17** | IPv6 | IPX | JXTA | NCP

IPv4 Conversations

Address A	Address B	Packets	Bytes	Packets A	Bytes A→B	Packets A	Bytes A←B	R
127.0.0.1	127.0.0.1	608	60 688	608	60 688	0	0	
172.17.0.9	172.17.42.1	1 048	103 536	432	47 328	616	56 208	
10.0.0.1	10.0.0.2	22	2 848	14	2 048	8	800	
10.0.0.1	10.0.0.3	22	2 848	14	2 048	8	800	
10.0.0.1	10.0.0.4	22	2 848	14	2 048	8	800	
10.0.0.2	10.0.0.3	22	2 848	14	2 048	8	800	
10.0.0.2	10.0.0.4	22	2 848	14	2 048	8	800	
10.0.0.3	10.0.0.4	22	2 848	14	2 048	8	800	
10.0.0.3	10.0.0.5	22	2 848	14	2 048	8	800	
10.0.0.3	10.0.0.6	22	2 848	14	2 048	8	800	
10.0.0.4	10.0.0.5	22	2 848	14	2 048	8	800	
10.0.0.4	10.0.0.6	22	2 848	14	2 048	8	800	
10.0.0.1	10.0.0.5	2	200	0	0	2	200	
10.0.0.2	10.0.0.5	2	200	0	0	2	200	
10.0.0.5	10.0.0.6	22	2 848	14	2 048	8	800	
10.0.0.1	10.0.0.6	2	200	0	0	2	200	
10.0.0.2	10.0.0.6	2	200	0	0	2	200	

5.8 Output Controller

Zum besseren Betrachten der Actions, haben wir Meldungen auf die Konsole gelogt, damit man den Netzwerkverkehr besser nachvollziehen kann:

```
[*] From 00:00:00:00:00:01 (Port 1) to ff:ff:ff:ff:ff:ff (Port 4294967291)
    Action: [Write Flow] Allowed Traffic: Broadcast ARP Request.
EVENT ofp_event->SimpleHub13 EventOFPPacketIn
[*] From 00:00:00:00:00:02 (Port 2) to 00:00:00:00:00:01 (Port 1)
    Action: [Write Flow] Allowed Traffic: Correct ACL and DST MAC on known Port.
EVENT ofp_event->SimpleHub13 EventOFPPacketIn
[*] From 00:00:00:00:00:01 (Port 1) to 00:00:00:00:00:02 (Port 2)
    Action: [Write Flow] Allowed Traffic: Correct ACL and DST MAC on known Port.
EVENT ofp_event->SimpleHub13 EventOFPPacketIn
[*] From 00:00:00:00:00:03 (Port 3) to 00:00:00:00:00:01 (Port 1)
    Action: [Write Flow] Allowed Traffic: Correct ACL and DST MAC on known Port.

[...]

[*] From 00:00:00:00:00:02 (Port 2) to 00:00:00:00:00:04 (Port 4)
    Action: [Write Flow] Allowed Traffic: Correct ACL and DST MAC on known Port.
EVENT ofp_event->SimpleHub13 EventOFPPacketIn
[*] From 00:00:00:00:00:05 (Port 5) to 00:00:00:00:00:02 (Port )
    Action: [Write Flow] Deny Traffic; Wrong ACL
EVENT ofp_event->SimpleHub13 EventOFPPacketIn
[*] From 00:00:00:00:00:06 (Port 6) to 00:00:00:00:00:02 (Port )
    Action: [Write Flow] Deny Traffic; Wrong ACL
EVENT ofp_event->SimpleHub13 EventOFPPacketIn
[*] From 00:00:00:00:00:03 (Port 3) to ff:ff:ff:ff:ff:ff (Port 4294967291)
    Action: [Write Flow] Allowed Traffic: Broadcast ARP Request.

[...]
```

Die in der Beschreibung vom `PacketInHandler` beschriebenen Szenarien sind im Output des Controllers gut ersichtlich.

6 Referenzen

- Um die Aufgaben zu lösen, haben wir das Buch "Using OpenFlow 1.3 - RYU SDN Framework" vom RYU Projektteam zur Hilfe genommen.
- Manpage von `ovs-ctl(8)`

7 Anhang

Im Anhang (im Zipfile) befinden sich folgende Dateien:

- `document.pdf`: Diese Dokumentation
- `01_hub.pcapng`: Wireshark Capture zum Hub
- `02_hub_flow.pcapng`: Wireshark Capture zum Hub mit Flow abenspitzen
- `03_switch.pcapng`: Wireshark Capture zum Switch
- `04_tenant.pcapng`: Wireshark Capture zur 3-Tier-Architektur
- `hub_flow.py`: OpenFlow Controller für den simplen Hub mit Flow abenspitzen
- `hub.py`: OpenFlow Controller für den simplen Hub
- `tierarch.py`: OpenFlow Controller für die 3-Tier-Architektur
- `switch.py`: OpenFlow Controller für den Switch