



Universität Würzburg
Lehrstuhl für Informatik III
Prof. Dr.-Ing. P. Tran-Gia

Sommersemester 2016

Bericht zum Hardwarepraktikum

„Internet-Technologien“

Aufgabe B: OpenFlow

Stanislav Lange und Lam Dinh-Xuan



Stand: 28. Juli 2016

Ala Eddine Ben Yahya MN : 2073907

Ramadan Shweiki MN : 2081693

Andrej Fink MN : 2068082

1 Einführung :

Openflow ist ein Protokoll, das zu einen Server ermöglicht zu sagen zu Netzwerk-Switches, wo Pakete zu senden sind. Bei einem konventionellen Netzwerk hat jeder Switch seine eigene Software, die entscheidet, was zu tun ist. Mit Openflow sind die Paket-moving Entscheidungen zentralisiert, so dass das Netzwerk unabhängig von der einzelnen Switches programmiert werden kann.

Bei einem Konventionellen Switch, packet forwarding (the data path) and high-level routing (the control path) treten auf dem gleichen Gerät.

Ein Openflow-Switch trennt Data-path von Control-path. Data-path befindet sich auf dem Switch selbst; ein separater Controller trifft die high-level routing Entscheidungen.

Der Switch und der Controller kommunizieren mittels des Openflow-Protokoll. Diese Methodik, Software-definierten Netzwerk (SDN), ermöglicht eine effizientere Nutzung von Netzwerkressourcen als bei konventionellen Netzwerken möglich ist. Openflow ist schon häufig angewendet in Bereiche wie VM(Virtual Machine) Mobilität, unternehmenskritische Netzwerke, und nächste Generation von IP-basierten Mobilfunknetzen .

Frage: Was ist der Unterschied zwischen Packet- und Flow-Switching?

! Packet-Switching:

- Aufteilung der Nachricht in Pakete begrenzter Länge
- Paketieren/ depaketieren im Ursprungs- und Zielknoten notwendig
- günstiger hinsichtlich der Speicherorganisation
- kurze effektive Übertragungszeit über mehrere Netzknoten hinweg

! Flow-Switching:

- Enthält Quelle und Ziel
- IP: Port # IP: Port (TCP/UDP)
- ermöglicht rerouting während der Laufzeit
- Erreicht schnellere Laufzeit durch Nutzung von Hash-Tabellen - Kann flows/packets fallen lassen

Packet-Switching unterteilt die Nachricht in Pakete begrenzter Länge. Die Datenpakete können in unterschiedlicher Reihenfolge am Ziel ankommen, da sie verschiedene Pfade nehmen können. Beim Flow-Switching ist durch die Flow-Tabelle der Pfad vorgegeben, den die Datenpakete gehen müssen.

Unterstützt OpenFlow das Packet-Switching?

Openflow unterstützt Packet-switching durch freiprogrammierbaren Controller, der Pakete mittels Packet-Switching weiterleitet . Dazu muss nur ein entsprechendes Programm implementiert werden.

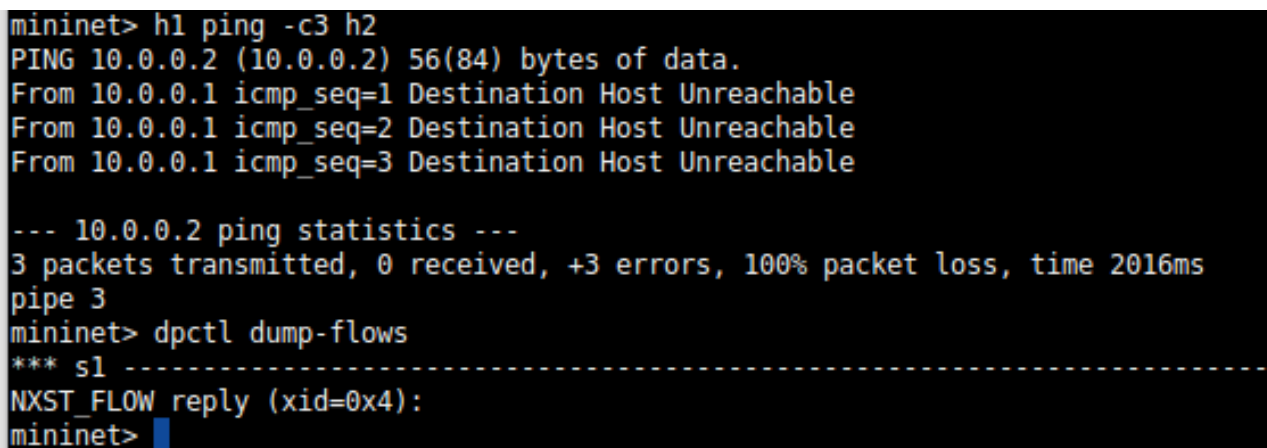
3.3 Ping-Test :

Frage: Bekommt h2 eine Antwort? Warum? Warum nicht?

h1 bekommt keine Antwort von h2, da die Flow-Tabelle noch leer ist .

```
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2015ms
pipe 3
```



```
mininet> h1 ping -c3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2016ms
pipe 3
mininet> dpctl dump-flows
*** s1 -----
NXST_FLOW reply (xid=0x4):
mininet> 
```

Abbildung 1: leere Tabelle

3.4 Flow-Tabelle füllen :

```
ubuntu@sdnhubvm:~[01:51]$ ovs-ofctl add-flow tcp:127.0.0.1:6634
in_port=1,actions=output:2
ubuntu@sdnhubvm:~[01:51]$ ovs-ofctl add-flow tcp:127.0.0.1:6634
in_port=2,actions=output:1
```

Diese Flow-Einträge sorgen dafür, dass Pakete von Port 1 an den Port 2 weitergeleitet werden und umgekehrt. Um zu verifizieren, dass die Einträge auch wirklich in die Flow-Tabelle geschrieben wurde :

```
mininet> dpctl dump-flows
```

erhalten wir folgende Ausgabe in unserem Terminal:

```
mininet> dpctl dump-flows
*** s1 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=99.378s, table=0, n_packets=0, n_bytes=0, idle_age=99,
  in_port=1 actions=output:2
  cookie=0x0, duration=84.923s, table=0, n_packets=0, n_bytes=0, idle_age=84,
  in_port=2 actions=output:1
```

Frage: Bekommt h1 nun eine Antwort? Kontrolliert die Flow-Tabelle und schaut euch die Statistiken für jeden Eintrag an. Entspricht dies den Erwartungen?

h1 pingt h2 dreimal erfolgreich an.

```
mininet> h1 ping -c3 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=0.482 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.073 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.074 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.073/0.209/0.482/0.193 ms
```

3.5 Netzwerkverkehr beobachten mit Wireshark :

```
$ ~/pox/pox.py forwarding.tutorial_l2_hub
```

Mit diesem Befehl startet der Controller, der sich wie ein Hub verhält, und demnach keine Flow-Einträge anlegt. Der Controller ist bereit nachdem folgende Ausgabe erscheint :

```
ubuntu@sdnhubvm:~[01:51]$ ~/pox/pox.py forwarding.tutorial_l2_hub
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
```

Frage: Welche Pakete sind zu sehen und welche Bedeutung haben sie?

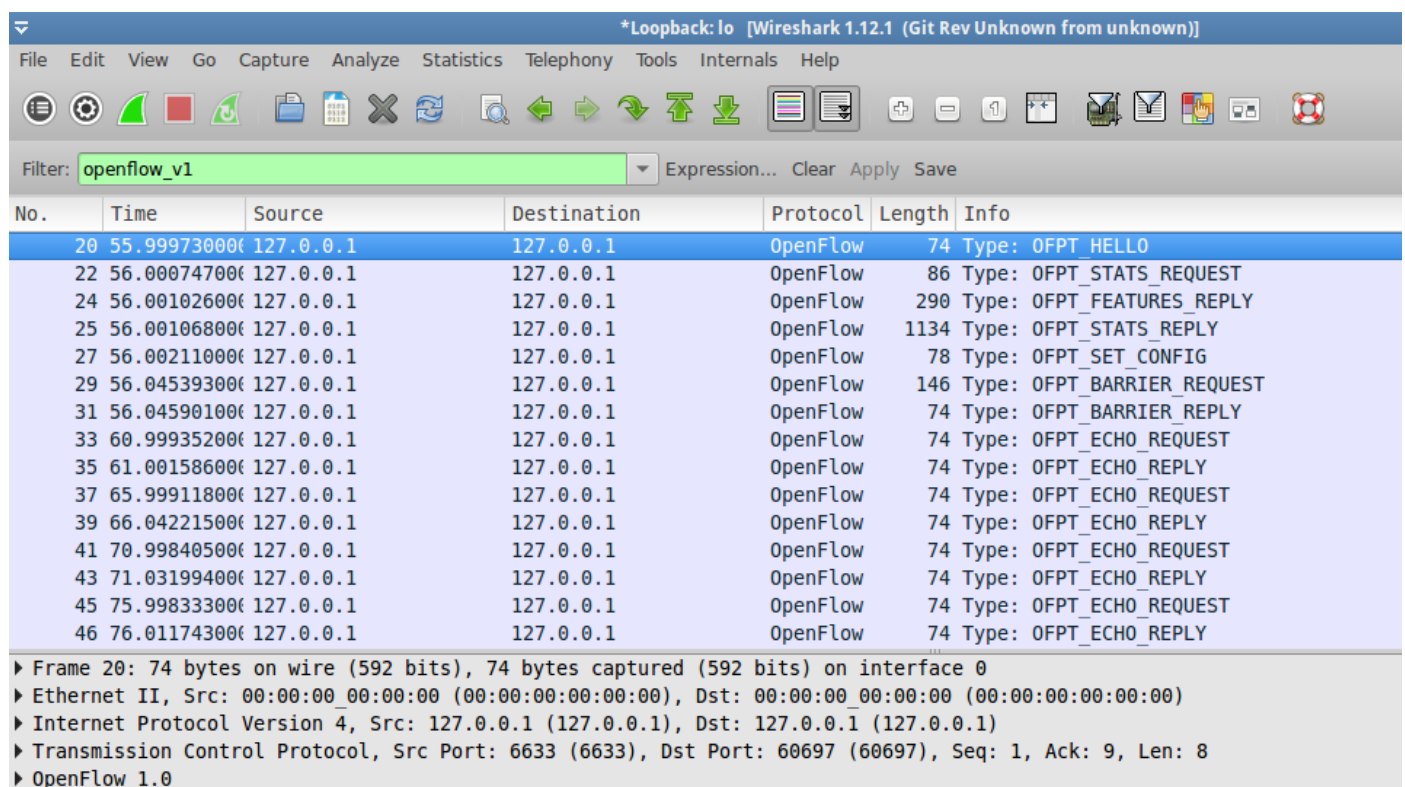


Abbildung 2 : wireshart mit filter openflow_v1

Packet	Beschreibung
OFPT_HELLO	Switch schickt seine Versionsnummer an den Controller
OFPT_STATS_REQUEST	Statistikinformationen, die vom Controller spezifiziert werden
OFPT_STATS_REPLY	Switch zeigt Informationen an, die in stats_request spezifiziert wurden
OFPT_FEATURES_REPLY	Switch schickt Details zu den Ports
OFPT_SET_CONFIG	Controller fordert Switch auf Flows zu senden
OFPT_BARRIER_REQUEST	Barrier request/reply messages werden vom Benutzer genutzt, um sicherzustellen, dass Voraussetzungen für bestimmte Nachrichten vorhanden sind, bzw. um Benachrichtigungen für erledigte Prozesse zu erhalten
OFPT_BARRIER_REPLY	
OFPT_ECHO_REQUEST	Anfragen, ob Switch noch da ist
OFPT_ECHO_REPLY	Antwort auf Echo Request

No.	Time	Source	Destination	Protocol	Length	Info
45	64.183905000	10.0.0.1	10.0.0.2	OpenFlow	182	Type: OFPT_PACKET_IN
46	64.202996000	127.0.0.1	127.0.0.1	OpenFlow	90	Type: OFPT_PACKET_OUT
48	64.203858000	10.0.0.2	10.0.0.1	OpenFlow	182	Type: OFPT_PACKET_IN
49	64.204497000	127.0.0.1	127.0.0.1	OpenFlow	90	Type: OFPT_PACKET_OUT
51	65.186983000	10.0.0.1	10.0.0.2	OpenFlow	182	Type: OFPT_PACKET_IN
53	65.233448000	127.0.0.1	127.0.0.1	OpenFlow	90	Type: OFPT_PACKET_OUT
55	65.235125000	10.0.0.2	10.0.0.1	OpenFlow	182	Type: OFPT_PACKET_IN
57	65.237979000	127.0.0.1	127.0.0.1	OpenFlow	90	Type: OFPT_PACKET_OUT

Abbildung 3 : Wireshark nach ping .

Packetname	Beschreibung
OFPT_PACKET_IN	h1 sendet einen ARP-Request um die MAC-Adresse von h2 zu erfahren. Der Switch sendet dann ein packet_in zum Controller.
OFPT_PACKET_OUT	Der Controller sendet ein packet_out an alle Ports. H2 antwortet auf den ARP-Request und der Controller legt einen flow an. Jetzt kennt h1 die MAC Adresse und kann den Ping durchführen.

3.6 - Controller-Benchmark mit iperf:

Bevor dass wir pox Controller starten, messen wir die Geschwindigkeit zwischen zwei Hosts h1 und h2 im Netzwerk durch den Befehl:

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
Waiting for iperf to start up...*** Results: ['13.3 Gbits/sec', '13.3 Gbits/sec']
```

Danach starten wir pox Controller und messen :

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['21.0 Mbits/sec', '23.1 Mbits/sec']
```

Der tutorial_l2_hub ist so viel langsamer als der selbstgeschriebene.

Frage: \$admin-hiwi will nun einen OpenFlow Software Switch im Linux User-Space implementieren. Wie würde sich das zuvor beobachtete Ergebnis verändern? Warum?

Durch diese Befehl konnten wir dasselbe Mininet aber mit user-space Switch

```
$ sudo mn --topo single,3 --mac --controller remote --switch user
```

Es wird langsamer.

Mit dem User-Space Switch, müssen Pakete überqueren von User-Space auf Kernel-Space und zurück . Der User-Space-Switch ist einfacher zu ändern, aber langsamer für die Simulation.

4 - Programmieren des intelligenten Switches:

4.1 Ryu-Controller mit Python

Wir beenden der pox Controller mit ctrl+c.

Und dann schließen ,reinigen , und nochmal starten wir die Mininet , mit der Befehle :

```
mininet> exit
ubuntu@sdnhubvm:~[01:49]$ sudo mn -c
ubuntu@sdnhubvm:~[01:49]$ sudo mn --topo single,3 --mac --switch ovsk
--controller remote
```

In einem zweiten Terminal started wir ryu Controller :

```
ubuntu@sdnhubvm:~[01:59]$ cd ~/ryu/
ubuntu@sdnhubvm:~/ryu[02:00] (master)$ PYTHONPATH=. ./bin/ryu-manager
--verbose ryu/app/hwp.py
loading app ryu/app/hwp.py
loading app ryu.controller.ofp_handler
instantiating app ryu/app/hwp.py of HWP
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK ofp_event
  PROVIDES EventOFPPortStatus TO {'HWP': set(['main'])}
  PROVIDES EventOFPPacketIn TO {'HWP': set(['main'])}
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPHello
  CONSUMES EventOFPErrormsg
  CONSUMES EventOFPPortDescStatsReply
BRICK HWP
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPPacketIn
connected socket:<eventlet.greenio.GreenSocket object at 0x7f7af71135d0>
address:('127.0.0.1', 60702)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x7f7af7113950>
```



```

move onto config mode
switch features ev version: 0x1 msg_type 0x6 xid 0xcc457126
OFPSwitchFeatures(actions=4095,capabilities=199,datapath_id=1,n_buffers=256,
n_tables=254,ports={1:
OFPPhyPort(port_no=1,hw_addr='ba:97:9c:10:8e:c5',name='s1-
eth1',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 2:
OFPPhyPort(port_no=2,hw_addr='66:07:37:6f:94:2c',name='s1-
eth2',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 3:
OFPPhyPort(port_no=3,hw_addr='0e:4e:16:3b:fc:f0',name='s1-
eth3',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 65534:
OFPPhyPort(port_no=65534,hw_addr='16:be:c4:5a:f1:4f',name='s1',config=1,stat
e=1,curr=0,advertised=0,supported=0,peer=0))}}
move onto main mode

```

4.2 Verifizieren des Verhaltens mittels tcpdump:

```

Node: h1
root@sdnhubvm:~[02:06]$ ping -c1 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=7.96 ms

--- 10.0.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.969/7.969/7.969/0.000 ms
root@sdnhubvm:~[02:07]$ ping -c1 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=10.6 ms

--- 10.0.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 10.655/10.655/10.655/0.000 ms
root@sdnhubvm:~[02:07]$ ping -c1 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.

Node: h2
0x0020: 0001 0000 6d4e 10f7 0001 d5d6 9557 0000 ....mN.....W..
0x0030: 0000 57b8 0000 0000 0000 1011 1213 1415 ..W.....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!"#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
02:07:43.014085 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 4347, seq 1, length 64

Node: h3
0x0020: 0001 0000 6d4e 10f7 0001 d5d6 9557 0000 ....mN.....W..
0x0030: 0000 57b8 0000 0000 0000 1011 1213 1415 ..W.....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!"#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
02:07:43.014095 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 4347, seq 1, length 64
0x0000: 0000 0000 0003 0000 0000 0001 0800 4500 .....E.
0x0010: 0054 5d70 4000 4001 c935 0a00 0001 0a00 .T]p@.0..5.....
0x0020: 0003 0800 6fe5 10fb 0001 dfd6 9557 0000 ....o.....W..
0x0030: 0000 431d 0000 0000 0000 1011 1213 1415 ..C.....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!"#$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67

```

Abbildung 4 : xterm h1,h2,h3 .

4.3 Benchmark des HUB-Controllers :

Wir kontrollieren zunächst in Mininet mit pingall dass wir alle Hosts erreichen. Dieser Befehl liefert unsfolgende Ausgabe:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

wir können alle Hosts erreichen.

Frage: Vergleicht das Ergebnis mit den Ergebnissen zu dem Referenz-Controller zuvor. Was fällt euch auf?

iperf liefert folgende Ausgabe:

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['13.1 Mbits/sec', '14.2 Mbits/sec']
```

Dieser Hub ist ungefähr halb so schnell wie der POX-Controller.

4.4 Programmierung des Controllers

Folgende Änderungen haben wir an der Funktion packet in handler() gemacht.

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

    msg = ev.msg

    datapath = msg.datapath
    ofproto = datapath.ofproto

    in_port = msg.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.logger.debug("sw:%s src:%s dst:%s port:%s", dpid, src, dst, in_port)
self.mac_to_port[dpid][src] = in_port
knownDst = 0
if dst in self.mac_to_port[dpid]:
    knownDst=1
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = ofproto.OFPP_FLOOD
actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]
data = None

if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data
out = datapath.ofproto_parser.OFPPacketOut(
    datapath=datapath, buffer_id=msg.buffer_id,
    in_port=msg.in_port, actions=actions, data=data)

if knownDst == 1:
    self.add_flow(datapath=datapath, in_port=msg.in_port, dst=dst
, actions=actions)

datapath.send_msg(out)

```

4.8 Controller testen

```

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['13.7 Gbits/sec', '13.8 Gbits/sec']
mininet> dpctl dump-flows
*** s1 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=171.869s, table=0, n_packets=3, n_bytes=182,
  idle_age=166, in_port=2, dl_dst=00:00:00:00:00:01 actions=output:1
  cookie=0x0, duration=171.857s, table=0, n_packets=66927, n_bytes=4418966,
  idle_age=163, in_port=3, dl_dst=00:00:00:00:00:01 actions=output:1
  cookie=0x0, duration=171.840s, table=0, n_packets=3, n_bytes=182,
  idle_age=166, in_port=1, dl_dst=00:00:00:00:00:02 actions=output:2
  cookie=0x0, duration=171.827s, table=0, n_packets=2, n_bytes=140,

```

```
idle_age=166, in_port=2, dl_dst=00:00:00:00:00:03 actions=output:3
cookie=0x0, duration=171.825s, table=0, n_packets=2, n_bytes=140,
idle_age=166, in_port=3, dl_dst=00:00:00:00:00:02 actions=output:2
cookie=0x0, duration=171.809s, table=0, n_packets=134870,
n_bytes=8603950004, idle_age=163, in_port=1, dl_dst=00:00:00:00:00:03
actions=output:3
```

4.9 Unterstützung mehrerer Switches

Wir beenden erst mininet , und die alte Configuration löschen durch :

```
mininet> exit
ubuntu@sdnhubvm:~[01:49]$ sudo mn -c
```

Und dann starten wir eine neu mit mehreren switches durch :

```
$ sudo mn --topo linear --switch ovsk --controller remote
```

Jetzt testen wir wieder dieses Netz mit dem pingall Befehl in der Mininet-Konsole und erhalten:

```
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Alle Hosts sind erreichbar.

4.10 Benchmark des eigenen Controllers

Zum Abschluss testen wir die Leistungsfähigkeit unseres Controllers mit dem Programm cbench.

Zuerst schließen wir Mininet :

```
mininet> exit
```

Danach beenden wir andere Controller starten unsere neu Controller, ohne verbose:

```
$ PYTHONPATH=. ./bin/ryu-manager ryu/app/hwp.py
```

Anschließend starten wir cbench mit:

```
$ cd ~/oflops/cbench
$ ./cbench -l5
```

Wir erhalten für unseren Controller folgende Ergebnisse:

```
buntu@sdnhubvm:~/oflops/cbench[05:48] (master)$ ./cbench -l5
cbench: controller benchmarking tool
  running in mode 'latency'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
  starting test with 0 ms delay after features_reply
  ignoring first 1 "warmup" and last 0 "cooldown" loops
  connection delay of 0ms per 1 switch(es)
  debugging info is off
05:48:11.412 16 switches: flows/sec: 445 377 386 367 365 443 315 299
531 379 361 447 409 573 247 333 total = 6.276893 per ms
05:48:12.512 16 switches: flows/sec: 478 338 444 342 496 382 316 432
513 330 370 422 387 466 364 418 total = 6.497994 per ms
05:48:13.613 16 switches: flows/sec: 458 400 484 440 466 458 388 422
461 378 338 440 388 466 383 378 total = 6.747980 per ms
05:48:14.714 16 switches: flows/sec: 378 388 458 466 400 356 484 434
382 440 364 459 444 338 363 422 total = 6.575928 per ms
05:48:15.815 16 switches: flows/sec: 422 444 356 338 364 422 382 356
364 434 434 338 382 421 444 382 total = 6.282975 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev =
6282.97/6747.98/6526.22/167.04 responses/s
```

Modified Hwp controller in mode Latency

```
$ ./cbench -l5 -t
```

Wir erhalten:

```
ubuntu@sdnhubvm:~/oflops/cbench[06:24] (master)$ ./cbench -l5 t
cbench: controller benchmarking tool
  running in mode 'latency'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
  starting test with 0 ms delay after features_reply
  ignoring first 1 "warmup" and last 0 "cooldown" loops
  connection delay of 0ms per 1 switch(es)
  debugging info is off
06:27:46.052 16 switches: flows/sec: 353 363 359 439 409 453 447 477
277 259 343 277 343 281 349 423 total = 5.851988 per ms
06:27:47.153 16 switches: flows/sec: 430 362 466 356 430 356 388 358
```

```

430 355 428 368 356 326 406 368 total = 6.182975 per ms
06:27:48.254 16 switches: flows/sec: 368 388 362 430 356 430 406 406
356 399 362 409 430 388 368 430 total = 6.287925 per ms
06:27:49.355 16 switches: flows/sec: 394 406 388 326 430 362 368 340
379 382 388 332 362 406 430 356 total = 6.048976 per ms
06:27:50.456 16 switches: flows/sec: 390 368 406 430 362 388 430 430
362 356 406 430 388 368 356 430 total = 6.299943 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev =
6048.98/6299.94/6204.95/100.90 responses/s

```

Modified Hwp controller in mode throughput :

```

ubuntu@sdnhubvm:~/oflops/cbench[06:27] (master)$ ./cbench -l5 -t
cbench: controller benchmarking tool
  running in mode 'throughput'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
  starting test with 0 ms delay after features_reply
  ignoring first 1 "warmup" and last 0 "cooldown" loops
  connection delay of 0ms per 1 switch(es)
  debugging info is off
06:27:54.848 16 switches: flows/sec: 273 273 17 49 515 17 515 273 257
437 787 15 273 515 1043 31 total = 5.289899 per ms
06:27:55.949 16 switches: flows/sec: 274 532 413 274 290 532 548 48 274
274 305 290 532 32 532 516 total = 5.665762 per ms
06:27:57.051 16 switches: flows/sec: 290 16 274 516 48 32 516 516 274
774 290 737 290 32 274 274 total = 5.149046 per ms
06:27:58.152 16 switches: flows/sec: 274 306 532 638 516 532 274 32 774
274 32 32 258 32 274 119 total = 4.895329 per ms
06:27:59.253 16 switches: flows/sec: 633 48 290 274 32 790 274 274 274
216 258 258 532 290 258 516 total = 5.216134 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev =
4895.33/5665.76/5231.57/277.77 responses/s

```

Pox controller in mode latency :

```

ubuntu@sdnhubvm:~/oflops/cbench[06:24] (master)$ ./cbench -l5 -t
cbench: controller benchmarking tool
  running in mode 'latency'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
  starting test with 0 ms delay after features_reply

```

```

ignoring first 1 "warmup" and last 0 "cooldown" loops
connection delay of 0ms per 1 switch(es)
debugging info is off
06:24:37.367 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
total = 0.000000 per ms
06:24:38.468 16 switches: flows/sec: 450 447 447 446 445 444 448 448
447 447 446 444 444 447 445 448 total = 7.142886 per ms
06:24:39.570 16 switches: flows/sec: 353 349 349 343 342 341 348 348
348 348 347 341 341 348 342 348 total = 5.535967 per ms
06:24:40.672 16 switches: flows/sec: 408 407 406 405 405 400 407 407
405 406 405 402 401 404 403 408 total = 6.478948 per ms
06:24:41.773 16 switches: flows/sec: 401 399 398 397 396 392 400 400
397 399 396 393 395 397 395 400 total = 6.354949 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev =
5535.97/7142.89/6378.19/571.14 responses/s

```

Pox controller in mode throughput :

```

ubuntu@sdnhubvm:~/oflops/cbench[06:24] (master)$ ./cbench -l5 -t
cbench: controller benchmarking tool
  running in mode 'throughput'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
  starting test with 0 ms delay after features_reply
  ignoring first 1 "warmup" and last 0 "cooldown" loops
  connection delay of 0ms per 1 switch(es)
  debugging info is off
06:24:55.435 16 switches: flows/sec: 0 0 1971 384 0 408 0 383 0 0 0 0 0 0
0 0 0 total = 3.136515 per ms
06:24:56.536 16 switches: flows/sec: 0 0 1427 1448 433 1423 0 1445 0 0
0 0 0 0 0 0 total = 6.175549 per ms
06:24:57.637 16 switches: flows/sec: 0 0 1274 1225 1225 1282 0 1225 0 0
0 0 0 0 0 0 total = 6.230707 per ms
06:24:58.737 16 switches: flows/sec: 0 0 875 899 899 974 509 874 484
434 434 0 0 0 0 total = 6.381770 per ms
06:24:59.838 16 switches: flows/sec: 0 0 715 749 753 749 749 749 771
774 749 0 0 0 0 total = 6.757831 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev =
6175.55/6757.83/6386.46/227.31 responses/s

```


Frage: Was ist schneller?

	POX controller	Eigene Controller
Latency mode	6378.19	6204.95
Throughput mode	6386.46	5231.57

POX Controller ist schneller .

POX-C++ in latency mode :

```
untu@sdnhubvm:~/oflops/cbench[06:19] (master)$ ./cbench -l5 t
cbench: controller benchmarking tool
  running in mode 'latency'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
  starting test with 0 ms delay after features_reply
  ignoring first 1 "warmup" and last 0 "cooldown" loops
  connection delay of 0ms per 1 switch(es)
  debugging info is off
06:20:44.937 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
total = 0.000000 per ms
06:20:46.037 16 switches: flows/sec: 373 373 372 371 372 370 371 364
370 372 370 371 368 369 366 366 total = 5.917994 per ms
06:20:47.140 16 switches: flows/sec: 334 333 333 333 333 333 333 321
333 333 333 333 333 333 333 329 total = 5.312957 per ms
06:20:48.240 16 switches: flows/sec: 318 318 318 318 318 317 317 313
317 317 316 317 316 317 316 315 total = 5.067985 per ms
06:20:49.341 16 switches: flows/sec: 235 234 234 234 234 232 234 226
234 234 233 234 232 234 232 231 total = 3.726981 per ms
RESULT: 16 switches 4 tests min/max/avg/stddev =
3726.98/5917.99/5006.48/800.89 responses/s
```

POX-C++ in throughput mode :

```
ubuntu@sdnhubvm:~/oflops/cbench[06:20] (master)$ ./cbench -l5 -t
cbench: controller benchmarking tool
  running in mode 'throughput'
  connecting to controller at localhost:6633
  faking 16 switches offset 1 :: 5 tests each; 1000 ms per test
  with 100000 unique source MACs per switch
  learning destination mac addresses before the test
```


starting test with 0 ms delay after features_reply
ignoring first 1 "warmup" and last 0 "cooldown" loops
connection delay of 0ms per 1 switch(es)
debugging info is off

```
06:21:01.175 16 switches: flows/sec: 0 0 0 0 0 0 1 0 758 0 714 0 0 0 0
0 total = 1.472738 per ms
06:21:02.276 16 switches: flows/sec: 0 0 0 0 0 0 1199 0 1180 0 1199 0
420 0 0 0 total = 3.997588 per ms
06:21:03.376 16 switches: flows/sec: 608 608 583 583 0 0 649 0 629 0
798 457 650 0 0 0 total = 5.564755 per ms
06:21:04.477 16 switches: flows/sec: 525 545 525 525 0 0 550 0 525 0
525 537 550 0 0 0 total = 4.806476 per ms
06:21:05.578 16 switches: flows/sec: 549 549 544 533 0 0 549 0 549 0
525 549 549 0 0 0 total = 4.895202 per ms
RESULT: 16 switches 4 tests min/max/avg/stddev =
3997.59/5564.76/4816.01/556.06 responses/s
```

Frage: Was fällt im Vergleich auf?

Bei dem POX-C++-Switch werden nicht alle Switches gleichmäßig ausgelastet.
Der Controller ist langsamer als beide anderen Controllers.

Frage: Welche Anpassungen sind notwendig, um den Switch in einen Router umzuwandeln?

Würden wir unserem Controller noch Funktionalitäten geben, einem Paket
einer andere Ziel-IP zu geben. Zusätzlich brauchen wir eine "Forwarding-Table" um
zwischen verschiedenen IP-Räumen Pakete zu senden. Diese kann dann auch zur Findung
schneller Routen zwischen den Netzwerken benutzt werden. So könnten wir aus unserem
Switch einen Router machen.