

Hardwarepraktikum: Aufgabe B

Student: Sebastian Hüller	Matrikelnummer: 2009968	Gruppe: 5
Student: Jonathan Stoll	Matrikelnummer: 1838730	Gruppe: 5
Student: Andre Bauer	Matrikelnummer: 1949971	Gruppe: 5
Student: Bianca Sasu	Matrikelnummer: 1976894	Gruppe: 5

1. Einführung

Zunächst galt es einige Fragen zu beantworten.

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

Beim **Packet-Switching** erfolgt eine Weiterleitung von Datenpaketen vom Sender an den Empfänger-PC. Hierbei werden alle Pakete in gleichlange Blöcke unterteilt und willkürlich geroutet, d. h. es kann sein, dass die Pakete über unterschiedliche Routen und Reihenfolge ans Ziel gelangen.

Wohingegen beim **Flow-Switching** die Route, die die Datenpakete nehmen müssen, durch die Flow-Tabelle eindeutig vorgegeben ist.

Unterstützt OpenFlow das Packetswitching?

Beim OpenFlow ermöglicht es der freiprogrammierbare Controller die Paketweiterleitung mittels Packet-Switching zu verwenden. Dazu muss nur ein entsprechendes Programm implementiert werden.

2. Versuchsumgebung / Simulation eines OpenFlow-Netzwerkes

Für die Durchführung der folgenden Aufgaben simulierten wir ein OpenFlow-Netzwerk in einer Linux-Umgebung mit Hilfe eines VirtualBox-Images.

3. Versuchsdurchführung

In den folgenden Teilaufgaben galt es schrittweise einen OpenFlow-Controller für einen Netzwerk-Hub in einen intelligenten Controller für einen Switch umzuwandeln.

Netzwerktopologie erzeugen mit Mininet

Wir starteten die virtuelle Maschine mit Ubuntu und öffneten dort das Terminal. Mit Hilfe des Befehls: `$ sudo mn --topo single,3 --mac --switch ovsk --controller remote` legten wir das Netzwerk mittels des Simulators Mininet an.

Ping-Test

Um den Host 2 von Host 1 aus anzupingen, verwendeten wir den Befehl:
`mininet> h1 ping -c3 h2`

Bekommt h2 eine Antwort? Warum?

Beim erstmaligen Verwenden des ping-Befehls bekommt der Host h2 keine Antwort vom Host h1, da die Flow-Tabelle noch keine Einträge enthält.

Flow-Tabelle füllen

Um dies zu realisieren, trugen wir manuell die Flows in die Flow-Tabelle ein.

```
$ ovs-ofctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
$ ovs-ofctl add-flow tcp:127.0.0.1:6634 in_port=2,actions=output:1
```

Durch dump-flow ist die Überprüfung der Einträge möglich:

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

Programmierung des Controllers

Bekommt h1 nun eine Antwort? Entspricht dies den Erwartungen?

Erst nach dem Eintragen der Flows in die Flow-Tabelle erhält Host 1 eine Antwort von Host 2. Nach Anwenden des "dump-flows"-Befehls wurden die von uns gemachten Einträge angezeigt. Diese wurden nun beim Verbindungsaufbau genutzt, was auch den Erwartungen entsprach.

Netzwerkverkehr beobachten mit Wireshark

Um Netzwerkverkehr aufzuzeichnen nutzen wir das Tool Wireshark. Dieses führten wir vor dem Start des POX-Controllers aus, der wie folgt aufgerufen wurde:

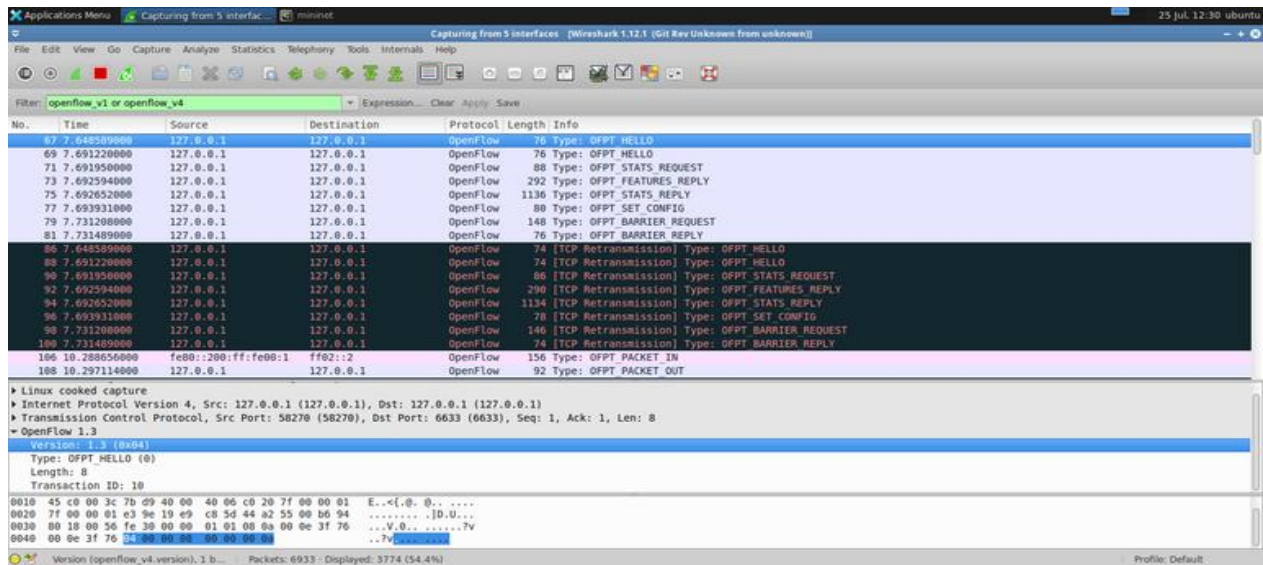
```
$ ~/pox/pox.py forwarding.tutorial_l2_hub
```

Nachdem zwischen Controller und Switch der obligatorische Versionsabgleich (Hello exchange) stattfand, war nun der Paketverkehr einsehbar.

Hinweis: Damit alle Pakete sichtbar wurden, mussten wir den Filter entsprechend anpassen, so dass nun die Pakete des Controllers - dieser läuft auf Version 4 - zum Switch angezeigt wurden.

Welche Pakete sind zu sehen und welche Bedeutung haben sie?

Der Einfachheit halber legten wir einen Screenshot an und fügten diesen bei. Zur näheren Erläuterung der Bedeutung der einzelnen Packet-Typen legten wir eine Tabelle an.



OFPT_HELLO	Der Controller erhält vom Switch die Versionsnummer
OFPT_STATS_REQUEST	Statistikinformationen werden vom Controller an den Switch gesendet
OFPT_STATS_REPLY	Switch antwortet auf die Anfrage des Controllers
OFPT_FEATURES_REPLY	Switch sendet detaillierte Informationen zu den Ports
OFPT_SET_CONFIG	Konfigurationseinstellungen des Switch werden vorgenommen
OFPT_BARRIER_REQUEST	Wird zur Gewährleistung von bestimmten Voraussetzungen genutzt
OFPT_BARRIER_REPLY	Ziel ist die Synchronisation. Wenn alle Ausführungen durchgeführt wurden, wird der Reply geschickt
OFPT_ECHO_REQUEST	Switch erbittet Bestätigung, ob Controller noch online ist

OFPT_ECHO_REPLY	Gibt Antwort auf den ECHO_REQUEST
OFPT_PACKET_IN	ARP-Request findet statt, um die Mac-Adresse zu erfragen, daraufhin sendet der Switch ein PACKET_IN zum Controller
OFPT_PACKET_OUT	Der Switch bekommt vom Controller ein PACKET_OUT und dieser legt einen Flow-Eintrag an. Nun sind sämtliche MAC-Adressen untereinander bekannt und der PING kann erfolgreich durchgeführt werden

Ein erneuter Ping-Test lieferte dasselbe Ergebnis wie der erste, es wurde dieselbe Zeit benötigt, da - aufgrund der fehlenden Flow-Einträge - vorab eine Kommunikation zwischen dem Switch und dem Controller nötig wurde, um ein Weiterversenden der Pakete zu erfragen.

Controller-Benchmark mit iperf

Im Folgenden mussten wir mit Hilfe des Benchmarks ([iperf](#) – Kommandozeile) die Geschwindigkeit zwischen dem POX-Controller und den manuellen Flowtabelleneinträgen messen. Ein iperf-TCP-Server startet sich auf einem virtuellen Rechner und ein iperf-Client auf einem anderen. Standardmäßig erfolgt die Messung in unserem Modellnetzwerk zwischen Host h1 und h3. Da jetzt die Verbindung zwischen Client und Server hergestellt wurde, kann ein Austausch der Pakete stattfinden.

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['34.1 Mbits/sec', '36.1 Mbits/sec']
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['8.11 Gbits/sec', '8.11 Gbits/sec']
```

POX

Flowtab.

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

Die Geschwindigkeit würde nun verringert werden, da durch die Implementierung im *Linux User-Space* die Pakete zuerst über diesen weitergeleitet werden und anschließend den *Kernel-Space* passieren müssen. Der Kernel-Space muss hierbei stets passiert werden, d. h. schaltet man diesem den Linux User-Space vorweg, hat dies insgesamt eine Verzögerung zur Folge. Der User-Space gibt dem Benutzer lediglich die Möglichkeit den Controller individuell zu spezifizieren und eine spezielle Filterung des Datenverkehrs zu bewerkstelligen.

4. Programmieren des intelligenten Switches

Im letzten Aufgabenteil erfolgte schließlich die Programmierung des Switches.

Für eine erfolgreiche Programmierung mussten wir folgende Schritte durchführen:

1) RYU -Controller mit Python

Zunächst deaktivierten wir den Referenz-Controller mittels des Befehls:

`$ sudo killall controller`, und schalteten den RYU-Controller ein.

Mit dem Befehl : `$ PYTHONPATH=. ./bin/ryu-manager --verbose ryu/app/hwp.py` , starten wir in einem extra Terminal den RYU-Controller.

2) Verifizieren des Verhaltens mittels tcpdump

Für die Gewährleistung der Kommunikation der PC's untereinander mussten wir drei xterm-Terminals starten und dort jeweils tcp-dump ausführen. Man konnte beobachten, dass bei ein Ping von h1 zu h3 in unserem Beispielnetzwerk sowohl h3 ein ICMP-Paket empfang, als auch h2.

3) Benchmark des HUB-Controllers

Mit dem Befehl:

`Mininet> pingall`

stellen wir sicher, dass alle Rechner auf Ping-Nachrichten antworten. Dafür müssen sowohl Mininet, als auch der RYU-Controller gestartet sein.

```
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: ['1.67 Mbits/sec', '2.44 Mbits/sec']
```

RYU

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

Die Weiterleitung der Pakete mittels des POX-Controllers (Referenz-Controller) war viel schneller als über den RYU-Controller. Beim RYU-Controller fand eine erhöhte und somit zeitraubende Kommunikation durch das OpenFlow-Protokoll zwischen den beiden Geräten statt. Der Switch fragte bei jedem Paket beim Controller nach, wo er die Daten hinschicken habe. In Wireshark konnten wir massenweise OFPT_PACKET_IN- und OFPT_PACKET_OUT-Pakete sehen.

8) Controller testen

Für die Durchführung des Aufgabenteils 4.8 mussten wir sowohl die allgemeine Konnektivität, als auch die Konnektivität auf jedem Rechner prüfen.

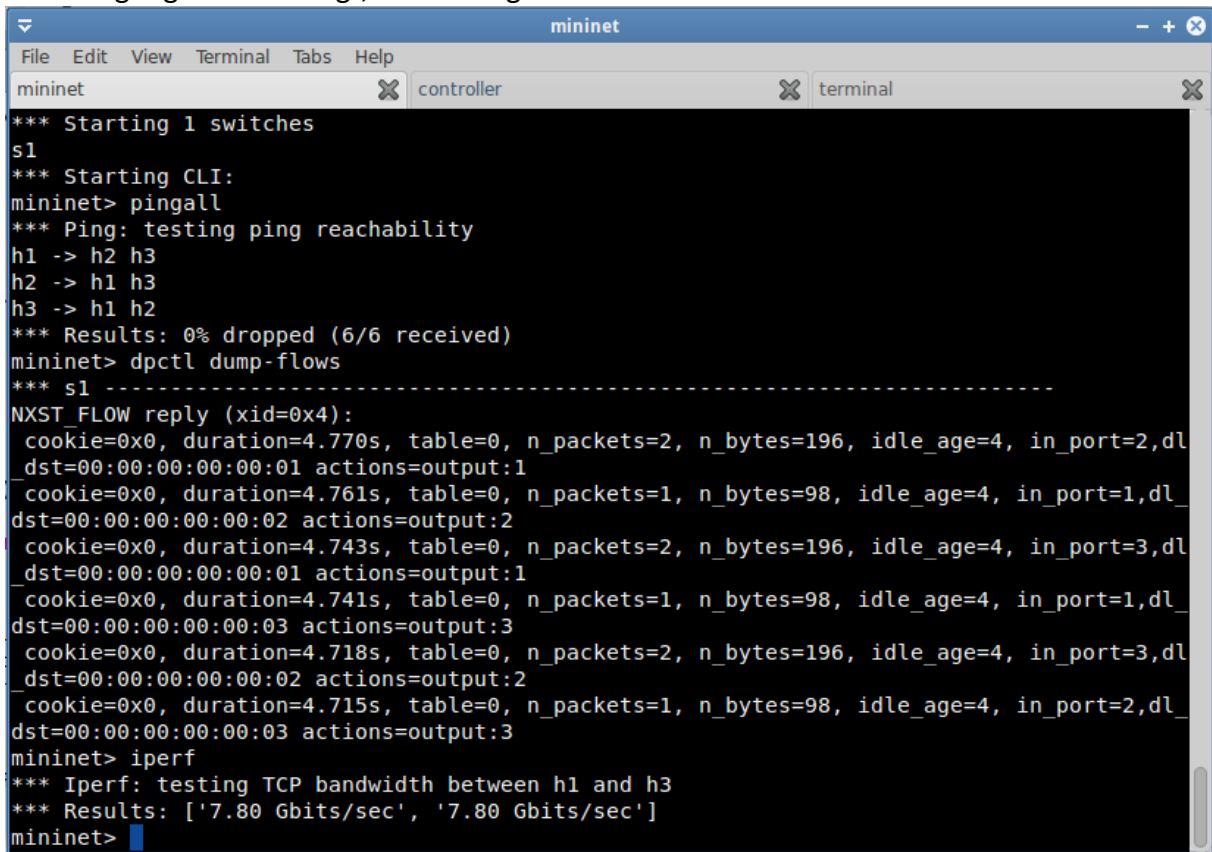
Im Nachfolgenden sind die einzelnen Terminals der Hosts 1 – 3 zu sehen, welche alle mittels tcpdump in einen lauschenden Modus versetzt wurden. Ein Ping von h1 auf h2 war erfolgreich. Die ARP-Anfrage kam auf allen drei Hosts an. Der anschließende Reply und auch der ICMP-Request wurden nur von Host 1 und Host 2 registriert.

```
root@schubum:/# [08:10] sudo tcpdump -XX -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
08:15:46.926608 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
0x0000: ffff ffff 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0000 0a00 0002 .....
08:15:46.933896 ARP, Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
0x0000: 0000 0000 0001 0000 0000 0002 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0002 0a00 0002 .....
0x0020: 0000 0000 0001 0a00 0001 .....
08:15:46.933912 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 6231, seq 1, length 64
0x0000: 0000 0000 0002 0000 0000 0001 0800 4500 .....E.
0x0010: 0054 7d8d 4000 4001 a303 0a00 0001 0a00 .....l..@.....
0x0020: 0002 0800 4962 1857 0001 a2f9 b455 0000 .....l..@.....
0x0030: 0000 7223 0a00 0000 0000 1011 1213 1415 .....r#.....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....l#%?
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 .....&()*+,-./012345
0x0060: 3637 .....67
08:15:46.938215 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 6231, seq 1, length 64
0x0000: 0000 0000 0001 0000 0000 0002 0800 4500 .....E.
0x0010: 0054 7eba 0000 4001 e7ee 0a00 0002 0a00 .....T...@.....
0x0020: 0000 0000 0001 0a00 0001 .....

root@schubum:/# [08:14] sudo tcpdump -XX -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
08:15:46.930686 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
0x0000: ffff ffff 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0000 0a00 0002 .....

mininet
File Edit View Terminal Tabs Help
mininet controller terminal
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> xterm h1 h2 h3
mininet> h1 ping -c1 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=11.6 ms
--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 11.637/11.637/11.637/0.000 ms
mininet>
```


Wir haben zunächst das Netzwerk mit mininet erzeugt und anschließend unseren erweiterten RYU-Controller gestartet. Nach einem pingall wurde für alle Richtungen im Netzwerk ein Flow-Eintrag im switch erstellt. Der nachfolgende Screenshot zeigt die Flow-Tabelle und auch den anschließend ausgeführten iperf, der wesentlich schnellere Übertragungsraten anzeigt, als der originale RYU-Controller:



```
mininet
File Edit View Terminal Tabs Help
mininet controller terminal
*** Starting 1 switches
s1
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet> dpctl dump-flows
*** s1 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=4.770s, table=0, n_packets=2, n_bytes=196, idle_age=4, in_port=2,dl
dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=4.761s, table=0, n_packets=1, n_bytes=98, idle_age=4, in_port=1,dl
dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=4.743s, table=0, n_packets=2, n_bytes=196, idle_age=4, in_port=3,dl
dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=4.741s, table=0, n_packets=1, n_bytes=98, idle_age=4, in_port=1,dl
dst=00:00:00:00:00:03 actions=output:3
 cookie=0x0, duration=4.718s, table=0, n_packets=2, n_bytes=196, idle_age=4, in_port=3,dl
dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=4.715s, table=0, n_packets=1, n_bytes=98, idle_age=4, in_port=2,dl
dst=00:00:00:00:00:03 actions=output:3
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['7.80 Gbits/sec', '7.80 Gbits/sec']
mininet>
```

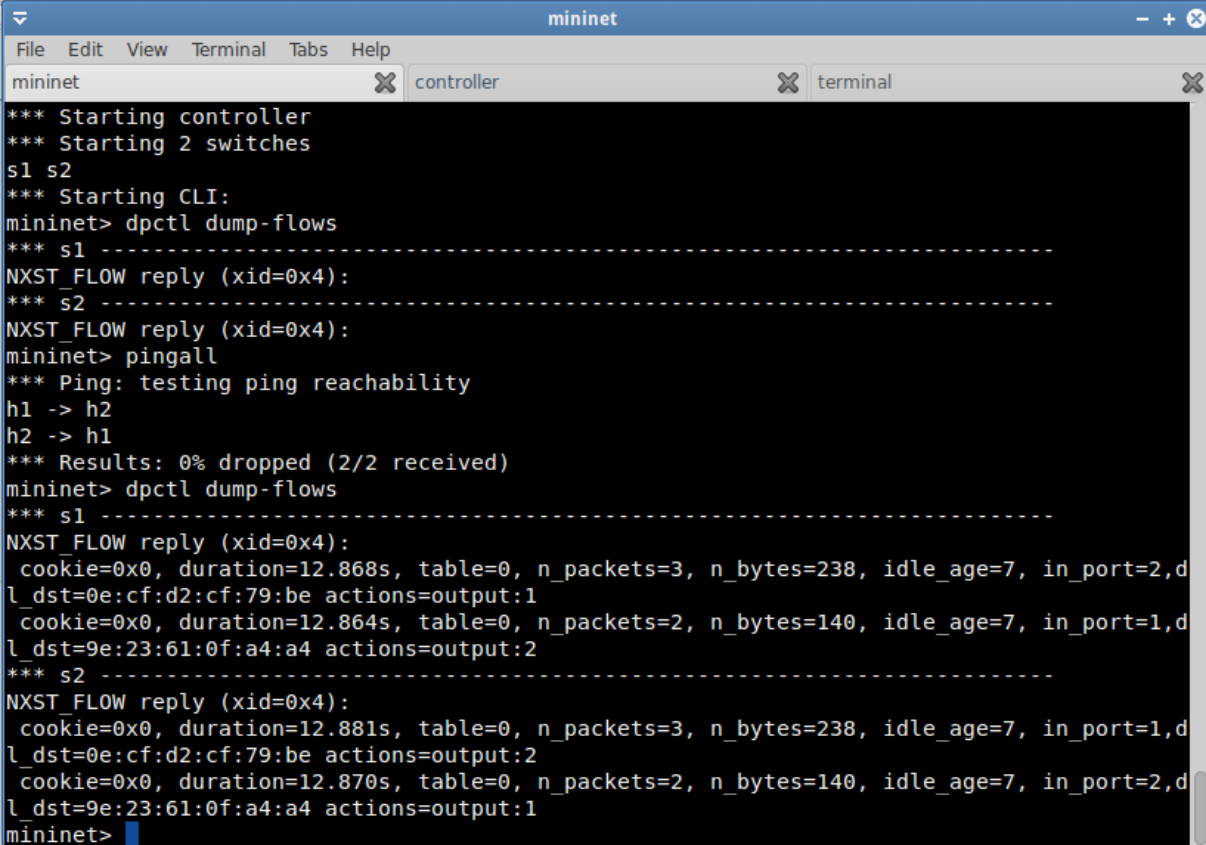

9) Unterstützung mehrerer Switches

Um die Aufgabe 4.9 zu lösen mussten wir den Controller erweitern, so dass dieser mehrere Hubs managen konnte.

Zuallererst haben wir der Mininet mit folgendem Befehl gestartet:

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

Nach der Modifizierung des Controller haben wir mit den Befehl `> pingall` die Funktionalität überprüft und konnten folgendes feststellen:



```
mininet
*** Starting controller
*** Starting 2 switches
s1 s2
*** Starting CLI:
mininet> dpctl dump-flows
*** s1 -----
NXST_FLOW reply (xid=0x4):
*** s2 -----
NXST_FLOW reply (xid=0x4):
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet> dpctl dump-flows
*** s1 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=12.868s, table=0, n_packets=3, n_bytes=238, idle_age=7, in_port=2,dl_dst=0e:cf:d2:cf:79:be actions=output:1
  cookie=0x0, duration=12.864s, table=0, n_packets=2, n_bytes=140, idle_age=7, in_port=1,dl_dst=9e:23:61:0f:a4:a4 actions=output:2
*** s2 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=12.881s, table=0, n_packets=3, n_bytes=238, idle_age=7, in_port=1,dl_dst=0e:cf:d2:cf:79:be actions=output:2
  cookie=0x0, duration=12.870s, table=0, n_packets=2, n_bytes=140, idle_age=7, in_port=2,dl_dst=9e:23:61:0f:a4:a4 actions=output:1
mininet>
```

10) Benchmark des eigenen Controllers

Zum Schluss erfolgte schließlich und endlich der Benchmark-Test unseres eigenen programmierten Controllers. Der Performance-Test wurde durch folgenden Befehl überprüft: `$ PYTHONPATH=. ./bin/ryu-manager --verbose ryu/app/hwp.py`

```
terminal
mininet controller terminal
ubuntu@sdnhubvm:~/oflops/cbench[08:49] (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
08:49:29.944 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 484 0 0 0 383 total = 0.859168 per ms
08:49:31.045 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 0 0 847 0 483 458 677 total = 2.464803 per ms
08:49:32.147 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 333 0 449 233 474 465 449 total = 2.402618 per ms
08:49:33.248 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 462 1 475 450 475 475 450 total = 2.786852 per ms
08:49:34.349 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 399 399 399 421 399 399 422 total = 2.837540 per ms
RESULT: 16 switches 4 tests min/max/avg/stddev = 2402.62/2837.54/2622.95/191.36 responses/s
ubuntu@sdnhubvm:~/oflops/cbench[08:49] (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
08:49:40.448 16 switches: flows/sec: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 total = 0.000000 per ms
08:49:41.548 16 switches: flows/sec: 22 21 22 20 22 22 20 22 20 22 20 20 22 22 0 0 total = 0.296999 per ms
08:49:42.652 16 switches: flows/sec: 21 21 21 21 21 21 21 21 21 21 21 21 21 21 0 0 total = 0.293150 per ms
08:49:43.753 16 switches: flows/sec: 31 31 32 31 31 31 31 32 31 32 31 31 31 31 0 0 total = 0.436978 per ms
08:49:44.853 16 switches: flows/sec: 23 23 23 23 23 23 23 23 23 23 23 23 23 23 0 0 total = 0.321997 per ms
RESULT: 16 switches 4 tests min/max/avg/stddev = 293.15/436.98/337.28/58.62 responses/s
ubuntu@sdnhubvm:~/oflops/cbench[08:49] (master)$
```

Test mit POX-Controller

```
terminal
File Edit View Terminal Tabs Help
mininet controller terminal
ubuntu@sdnhubvm:~/oflops/cbench[08:51] (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
08:51:30.619 16 switches: flows/sec: 0 483 0 0 0 0 408 0 0 0 0 0 232 0 0 0 total = 1.122679 per ms
08:51:31.720 16 switches: flows/sec: 258 425 183 183 133 108 424 0 58 0 0 0 399 0 0 0 total = 2.170618 per ms
08:51:32.821 16 switches: flows/sec: 299 250 225 225 235 225 250 0 225 0 0 0 259 0 208 0 total = 2.400469 per ms
08:51:33.924 16 switches: flows/sec: 248 250 250 250 250 250 250 0 250 0 0 0 250 0 225 0 total = 2.472174 per ms
08:51:35.025 16 switches: flows/sec: 225 225 240 224 224 224 224 83 249 0 0 0 236 0 225 0 total = 2.378370 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev = 2170.62/2472.17/2355.41/112.18 responses/s
ubuntu@sdnhubvm:~/oflops/cbench[08:51] (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
08:51:40.284 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
08:51:41.385 16 switches: flows/sec: 7 7 7 8 0 0 0 0 0 0 0 0 0 0 0 0 total = 0.029000 per ms
08:51:42.486 16 switches: flows/sec: 6 6 5 6 0 0 0 6 6 6 6 6 6 0 0 0 total = 0.059000 per ms
08:51:43.586 16 switches: flows/sec: 10 10 5 10 0 0 0 10 10 10 10 10 10 0 0 10 total = 0.105000 per ms
08:51:44.688 16 switches: flows/sec: 8 8 5 7 0 0 0 8 8 8 8 8 8 0 0 8 total = 0.083938 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev = 29.00/105.00/69.23/28.37 responses/s
ubuntu@sdnhubvm:~/oflops/cbench[08:51] (master)$
```

Test mit POX C++ Controller

```
terminal
File Edit View Terminal Tabs Help
mininet controller terminal
ubuntu@sdnhubvm:~/oflops/cbench[08:37] (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
08:37:58.499 16 switches: flows/sec: 0 17 17 0 0 17 255 17 0 0 0 255 103 255 255 0 0 total = 1.184946 per ms
08:37:59.604 16 switches: flows/sec: 0 0 18 273 0 0 0 494 0 0 0 0 0 274 17 0 total = 1.071997 per ms
08:38:00.708 16 switches: flows/sec: 15 258 16 0 16 0 395 16 0 396 0 0 0 258 15 total = 1.381390 per ms
08:38:01.813 16 switches: flows/sec: 189 258 0 16 412 258 0 0 0 16 0 258 16 16 0 0 total = 1.436960 per ms
08:38:02.917 16 switches: flows/sec: 16 16 258 0 0 16 465 0 0 0 274 0 378 16 total = 1.434423 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev = 1072.00/1436.96/1331.19/151.28 responses/s
ubuntu@sdnhubvm:~/oflops/cbench[08:38] (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
08:46:25.274 16 switches: flows/sec: 137 77 65 99 77 51 135 67 103 71 43 41 85 55 55 101 total = 1.261999 per ms
08:46:26.375 16 switches: flows/sec: 88 50 48 90 44 80 78 46 84 48 46 90 46 42 42 76 total = 0.997997 per ms
08:46:27.476 16 switches: flows/sec: 100 66 62 66 76 90 48 40 38 38 38 40 68 68 36 36 total = 0.909991 per ms
08:46:28.580 16 switches: flows/sec: 46 50 96 42 42 78 78 62 56 60 48 38 72 80 46 76 total = 0.966581 per ms
08:46:29.680 16 switches: flows/sec: 42 66 94 44 42 40 56 80 70 86 74 38 36 56 38 54 total = 0.915993 per ms
RESULT: 16 switches 4 tests min/max/avg/stdev = 909.99/998.00/947.64/36.45 responses/s
ubuntu@sdnhubvm:~/oflops/cbench[08:46] (master)$
```

Test mit erweitertem RYU- Controller

Was ist schneller? Was fällt im Vergleich auf?

Vergleich der Controller mit cbench:

	RYU	POX	POX C++
Throughput (Option-t)	1331	2622	2355
Latency	947	337	69

Angaben in „durchschnittliche Antworten pro Sekunde“.

Bei Throughput war der POX-Controller der Schnellste.

Wenn man die Latenz betrachtet bemerkt man, dass der RYU-Controller der Schnellste ist.

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

Damit ein Switch auf Schicht 3 des OSI-Modells arbeitet, sind folgende Zusatzfunktionen einzubauen:

- Ziel-IP-Adresse ändern
- Einbau einer Forwarding-Table
- Subnetting
- Optional NAT

Mit diesen Funktionen ist es möglich Netze miteinander zu verbinden oder voneinander zu trennen.