Cloud Infrastructure

Linux Networking

eine Arbeit von

Michael Schneider

Shaban Rexhepi

Gruppenkürzel

cldinf-g-7

betreut von

Beat Stettler

Laurent Metzger

HS 2020

Ostschweizer Fachhochschule

Inhaltsangabe

0. Setup

Einrichten der SSH-Verbindung

Annahme: *Wir nehmen bei allen Erläuterungen an, dass eine VPN-Verbindung zum INS vorhanden ist.*

Alle Credentials zu den Hosts sind im "*Appendix 1: Credentials & Verbindungsinformation*" aufgelistet.

Leider konnten wir uns nicht direkt per SSH mit den Servern verbinden. Das Problem scheint allgemein bekannt. Deshalb wurde uns ein Jumphost zur Verfügung gestellt:

Terminal öffnen und die Verbindung zum Jumphost herstellen:

```
$ ssh ins@10.18.10.2
```

Nun ist das Login auf den Servern möglich. Die Befehle dazu sind ebenfalls im Appendix 1.

SCP Probleme

Da die Konfigurationsdateien zur Abgabe gehören, wollten wir diese mit "*scp*" runterladen. Wir haben alles ausprobiert:

- Zwischenspeichern auf dem Jumphost,
- Tunneling über den Jumphost,

nichts hat funktioniert. Nach zwei Stunden haben wir im MS-Teams nachgefragt:
Es ist nicht möglich mit "*scp*" Dateien herunterzuladen.

Bei den Performancetests haben wir die Resultate in Dateien abgespeichert. Diese liegen im "*home*" folder des root users. Da es sich um viele Dateien handelt, hielten wir es für unzumutbar diese ins .zip der Abgabe per Copy-Paste abzulegen. Die Dateien sind immer noch auf R5 und R2 vorhanden und können dort eingesehen werden.

1. Änderung von Hostname und Passwort

Hostnamen ändern

1. Auf den Maschinen R1 bis R5 werden die Hostnamen jeweils zu RX geändert, wobei man für das X die entsprechende Zahl (1 bis 5) einsetzt:

   RX: `$ sudo hostname RX`

2. Nun editieren wir die Datei `/etc/hostname` zum neuen Hostnamen:

   RX: `$ sudo nano /etc/hostname`
   → *Zeile mit CTRL + K löschen*
   → *neuen Hostnamen eintragen*
   → *Abspeichern mit CTRL + O*
   → *Bestätigen mit ENTER*
   → "*nano*" *verlassen mit CTRL + X*

3. Nun ersetzen wir den neuen Namen mit dem alten in `/etc/hosts` vor der Loopback-Adresse. Falls kein Loopback vorhanden ist, erstellen wir einen. Die Dateibearbeitung erfolgt analog zu vorhin und wird zukünftig nicht weiter ausgeführt:

   RX: `$ sudo nano /etc/hosts`
   von: *127.0.1.1* ***alter-hostname***
   zu: *127.0.1.1* ***neuer-hostname***

4. Eine Kontrolle mit einem relogin: `$ exit`

Passwort ändern

1. Wir geben folgenden Befehl in die Kommandozeile ein
   `$ passwd`
   i. Enter new UNIX password:
   ii. Retype new UNIX password:

   passwd: password updated successfully

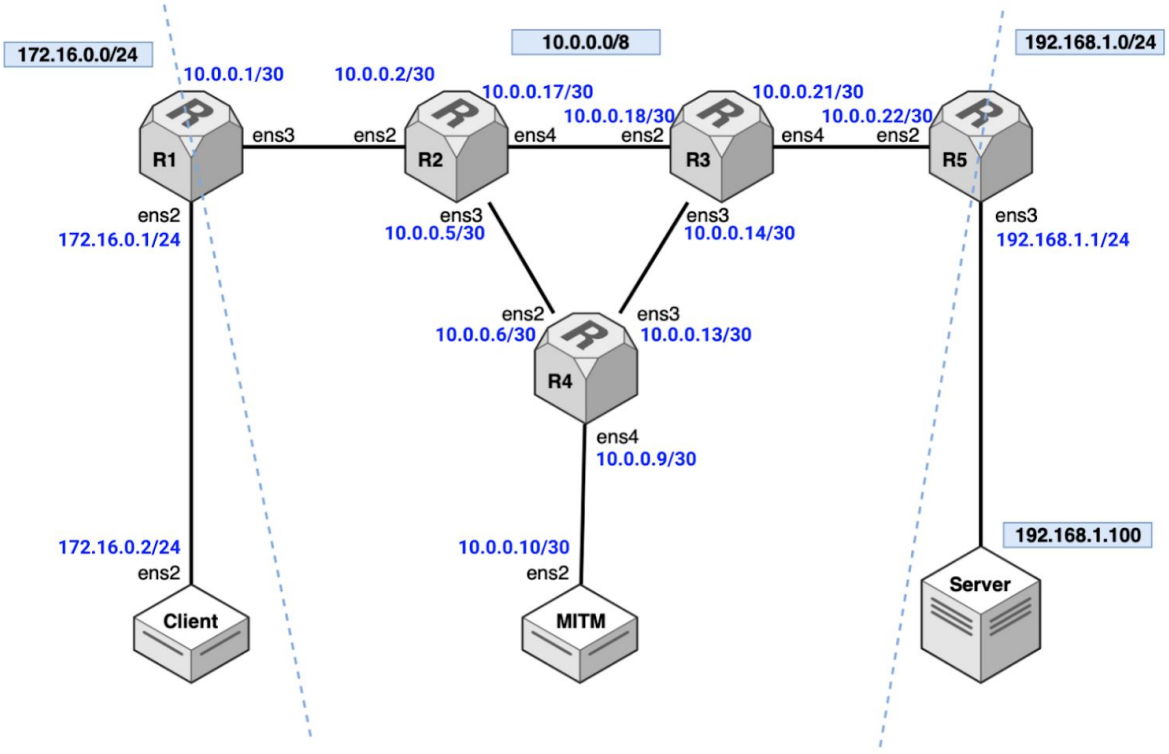
2. Design des IP-Adressplanes

Wir haben für die drei Netze drei private (nach RFC 1918) Netzräume. Wir haben uns dazu entschieden jedem Link ein kleinstmögliches Subnetz zuzuweisen. Das macht für uns das Troubleshooting einfacher, da wir der Netzadresse den Link zuordnen können.

Unserer Meinung nach ist die Wahl des Designs “aufgeräumter”. Wir füllen den Adressraum von der ersten IP an mit Hosts bzw. Netzen.

Die Netzmaske ist im 10-er Block kleinstmöglich gewählt - /30 bietet Platz für zwei Hosts.

In den B und C Klassen haben wir jeweils dem Interface des Borderrouters die erste IP vergeben. Auch hier könnte man das anders gestalten, jedoch finden wir es einfach, wenn der erste Host im Adressraum der Gateway ist.



IP-Adressplan visualisiert[1]

Im “*Appendix 2: IP-Adressplan tabellarisch”* befindet sich derselbe Plan in Tabellenform.

[1] Das Bild ist aus der Aufgabenstellung entnommen und von uns bearbeitet.

3. Vergabe der IP-Adressen durch netplan

Nun verwirklichen wir den IP-Adressplan. Dazu benutzen wir "*netplan*". Auf jeder Maschine, bis auf den Server, werden die IP-Adressen zugewiesen. Hier unser Rezept:

1. Im ersten Schritt sollte ein Backup des Konfigurationsfiles erstellt werden.

   RX: `$ cp /etc/netplan/'unser-config-file'.yaml 'unser-config-file'.yaml.bak`

2. Ändern der "*netplan*"-Standardkonfiguration[2] gemäss IP-Adressplan:

   RX: `$ nano /etc/netplan/'unser-config-file'.yaml`

   Beispielfile auf R1 in `/etc/netplan/setup.yaml:`

   ```
   network:
         version: 2
               ethernets:
                     ens2:
                           addresses:
                                 - 172.16.0.1/24
                     ens3:
                           addresses:
                                 - 10.0.0.1/30
   ```

   - IPs, Interfaces und CIDR-Notation genau übernehmen und prüfen!
   - dhcp Anweisungen sind alle zu entnehmen.
   - Einrückungen müssen, ähnlich wie in Python, beachtet werden.
   - Speziell auf dem Client muss noch ein Default Gateway gesetzt werden.

3. Nachdem die Konfiguration gespeichert ist, muss sie noch angewendet werden.

   RX: `$ sudo netplan apply`

   Auch wenn gewisse Interfaces bis anhin im "*down*" waren. Bei uns hat "*netplan*" den Status nach dem letzten Befehl auf "*up*" gesetzt. Sollte es dennoch zu Problemen mit Interfaces kommen, dann kann mit `$ sudo ip link set dev ensX up` probieren.

4. Kontrolle: Nun sollte man benachbarte Router anpingen können. Fast immer funktioniert der Ping. Die Einzelfälle, in denen es nicht funktioniert sind im nächsten Kapitel erläutert.

[2] "Netplan configuration examples - Netplan | Backend-agnostic ...." https://netplan.io/examples/. Accessed 2 Oct. 2020.

3. Einrichtung von OSPF mit BIRD

In der uns gegebenen Konfiguration konnten wir mit “*cd*” nicht nach "`/etc/bird`" navigieren. Deshalb haben wir uns als root user eingeloggt:

RX: `$ sudo -i`

`$ sudo cd /etc/bird` würde nicht funktionieren, da "*cd*" kein gewöhnliches Programm ist. Es ist ein built-in Command[3]. Deshalb müssen wir ab hier direkt als root user arbeiten, wenn wir uns in `/etc/bird` bewegen wollen. Wir haben nicht erörtern können weshalb ein normaler user nicht ins erwähnte Verzeichnis wechseln kann. Im Verzeichnis `/etc/netplan` hingegen hatten wir dieses Problem nicht.

BIRD Konfiguration

1. Analog zum letzten Kapitel, erstellen wir ein Backup von der Standardkonfiguration.

   ```
   # cp /etc/bird/'unser-config-file'.conf
   'user-config-file'.conf.bak
   ```

2. Beispielkonfiguration auf R1:

```
log syslog all; # Logging in den syslog

router id 1.1.1.1;    # Router ID setzen

protocol device {
        scan time 10;
}
protocol kernel { # Synchronisation mit Kernel-Routing-Table
        export all;
        scan time 15;  # Alle 15 Sekunden wird abgeglichen
}
protocol ospf v2 { # OSPF v2 verwenden
        import all;
        area 0 {   # Alle Interfaces gehören zum Backbone
                interface "ens3" {
                        cost 5;
                        hello 5;
                        retransmit 2;
                        wait 10;
                        dead 20;
                };
                interface "ens2" {
                        stub; # Interface passiv machen
                };
        };
}
```

[3] "Why is cd not a program? - Unix & Linux Stack Exchange." 16 May. 2012, https://unix.stackexchange.com/questions/38808/why-is-cd-not-a-program. Accessed 4 Oct. 2020.

3. Danach müssen wir den Service neu starten und den Status überprüfen:

   RX: `$ sudo systemctl restart bird.service`
   RX: `$ sudo systemctl status bird.service`

Wir haben die "*bird*" Konfiguration neben der Dokumentation auch experimentell kennengelernt. Beispielsweise haben wir den Kernel Scan Timer auf 15 gestellt (Default 20). Kosten haben wir überall auf 5 gesetzt, um später Experimente, wie mit ECMP zu vereinfachen.

Einrichten von IPV4-Forwarding

1. Wir editieren die Datei `/etc/sysctl.conf`

   RX: `$ sudo nano /etc/sysctl.conf`

2. Wir stellen sicher, dass folgender Ausdruck unkommentiert wird:

   "*net.ipv4.ip_forward = 1*"

3. Nun müssen wir die Konfiguration laden und permanent speichern (der Neustart sollte garantiert sein)

   RX: `$ sudo sysctl -p`

CPU-Limit setzten

Mit “*renice*” können wir die Priorität eines laufenden Prozesses dem Kernel neu ermitteln. Zuerst müssen wir die Process ID ermitteln und diese dann mit renice auf einen Wert (von -20 bis 19) setzten. Da OSPF ein Hintergrundprozess ist, weisen wir dem einen niedrigeren Wert von -10 zu:

RX: `$ ps aux | grep bird`

```
bird      9565  0.0  0.1  15780  2440 ?        Ss   18:02   0:00
/usr/sbin/bird -f -u bird -g bird
```

RX: `$ sudo renice -10 9565`

```
9565 (process ID) old priority 0, new priority -10
```

Die Qualitätsmassnahmen sind im nächsten Kapitel aufgeführt.

4. Problem der inkonsistenten Wegfindung

Unser erstes Ziel war es in einem Zwischenschritt auf dem 10-er Netzwerk OSPF korrekt aufzusetzen. Qualitätsmaßnahmen, die wir hierfür getroffen haben, waren:

- Kontrolle der Kernel-Routing-Table mit `$ sudo route`
- Kontrolle der OSPF-Nachbarschaften `bird> show ospf neighbor`
- Kontrolle der Topologie mit `bird> show ospf topology`
- Pingen von einem Interface zu einem anderen - in sehr vielen Variationen.

Im letzten Punkt haben wir bemerkt, dass gewisse Pings nicht funktionieren.

Beispiel 1: *ping von von 10.0.0.22 zu 10.0.0.5*

Mit `tcpdump -i` kann man den TCP-Traffic analysieren. Man sieht, dass der Request von 10.0.0.22 eingeht, aber keine Response von 10.0.0.5 im Traffic ist. Wenn man aber von R3 aus die Adresse 10.0.0.5 anpingt, sieht man, dass Hin- und Rückweg der Päckchen sich unterscheiden.

Nachtrag

Dieses Problem hat uns Zeit gekostet. Nachdem die Laborassistenten uns versicherten, dass es nicht an unserer Konfiguration liegt, haben den Nachtrag in die Doku mit Bildern auf später verschoben.

Nun als es soweit ist, können wir dieses Problem leider nicht reproduzieren. Wir entschuldigen uns es nicht ausführlicher darstellen zu können und bitten zu berücksichtigen, dass an dieser Stelle viel Zeit ins Troubleshooting investiert wurde.

5. Verifizierung weiterer Punkte

Route Failover

R3

R3: `# cat bird.conf`

```
router id 3.3.3.3;

protocol device {
        scan time 10;
}
protocol kernel {
        export all;
        scan time 15;
}

protocol ospf v2 {
        import all;
        area 0 {
                interface "ens3", "ens4" {
                        cost 5;
                        hello 5;
                        retransmit 2;
                        wait 10;
                        dead 20;
                };
                interface "ens2" {
                        cost 30;
                        hello 5;
                        retransmit 2;
                        wait 10;
                        dead 20;
                };
        };
}
```

R2

R2: `# cat bird.conf`

```
router id 2.2.2.2;

protocol device {
	scan time 10;
}

protocol kernel {
	export all;
	scan time 15;
}

protocol ospf v2 {
	import all;

	area 0 {
		interface "ens2", "ens3" {
			cost 5;
			hello 5;
			retransmit 2;
			wait 10;
			dead 20;
		};
		interface "ens4" {
			cost 30;
			hello 5;
			retransmit 2;
			wait 10;
			dead 20;
		};
	};
}
```

mtr 10.0.0.22 vor shutdown ens2 auf R4:

```
                         My traceroute  [v0.92]
R1 (10.0.0.1)                                  2020-10-02T13:25:30+0000
Keys:  Help   Display mode   Restart statistics   Order of fields   quit
                                 Packets               Pings
 Host                          Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. 10.0.0.2                    0.0%    18    0.5   0.6   0.3   1.3   0.2
 2. 10.0.0.6                    0.0%    17    2.6   1.5   0.5   3.2   0.8
 3. 10.0.0.14                   0.0%    17    6.1   2.7   1.0   7.0   1.9
 4. 10.0.0.22                   0.0%    17    1.7   3.7   1.1  11.8   3.5
```

mtr 10.0.0.22 nach shutdown ens2 auf R4:
Shutdown wurde erreicht mit dem Befehl `$ sudo ifconfig ens2 down`

```
                         My traceroute  [v0.92]
R1 (10.0.0.1)                                  2020-10-02T13:34:17+0000
Keys:  Help   Display mode   Restart statistics   Order of fields   quit
                                 Packets               Pings
 Host                          Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. 10.0.0.2                    0.0%    15    0.4   1.0   0.3   4.5   1.1
 2. 10.0.0.18                   0.0%    14    0.7   1.6   0.6   7.8   2.0
 3. 10.0.0.22                   0.0%    14    1.2   2.9   0.9   9.9   3.0
```

Rückgängigmachen der Konfiguration:

- `$ sudo ifconfig ens2 up`
- Kosten im "*bird*" zurücksetzten

Passive Interfaces

Passive Interfaces haben wir konfiguriert mit der `stub` Anweisung im `/etc/bird/bird.conf`. Es werden, wie im Kapitel 2.6 in RFC 2328[4] beschrieben, keine LSA an stub areas weitergegeben, womit unser Ziel erreicht ist:

Der Client sieht keinen OSPF Traffic.

Das haben wir beim Client und sichergestellt mit dem Befehl:

Client: `$ sudo tcpdump -i ens2`

Access Website

Nun stellen wir sicher, dass der Server über HTTP erreichbar ist mit "*curl*". Hierzu führen wir folgendes aus:

Client: `$ curl 192.168.1.100:8080`

Der Request war erfolgreich und wir bekommen ein HTML zurück:

```
<h1>Hello Docker Swarm!</h1><hr><p>A simple hello web app, in a docker image,
using debian, python and
flask!</p><p>Hostname:python-flask-hello</p><p>Requests: 156<br
\></p><p>Network Interface: eth0<br \>--> IP Address: 192.168.1.100<br \>-->
Netmask: 255.255.255.0<br \></p>
```

[4] "RFC 2328 - OSPF Version 2 - IETF Tools." https://tools.ietf.org/html/rfc2328. Accessed 2 Oct. 2020.

6. Performance

6.0 Checkup des Setups

Wir haben vor den Output der Performancetests in Dateien abzuspeichern. In einem kleinen Beispiel erläutern wir, wie das Abspeichern realisiert wird.

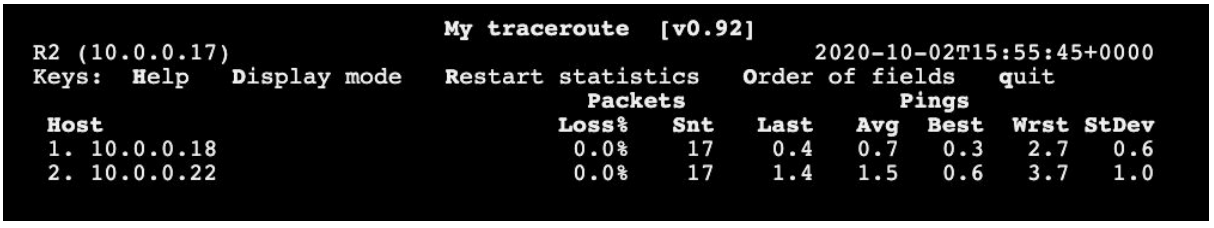
1. **R5**: Bereitstellung des Servers auf R5 mit `$ sudo iperf3 -s`. Wir wollen aber den Output in ein File speichern. Das vollziehen wir mit dem stream redirection operator. Der Operator verhält sich gemäss POSIX-Norm[5]: Ein `>>` ergänzt das angegebene File mit dem Stream Output. Falls das angegebene File nicht vorhanden ist, wird vor der Schreiboperation ein neues File erstellt.

   ```
   $ sudo iperf3 -s >> dateiname.txt
   ```

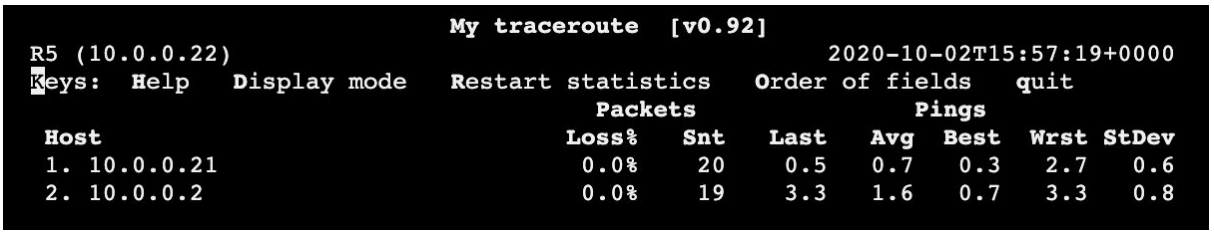
2. **R2:** Mit dem Client (R2) werden wir nun den Performancetest vollziehen. Dafür geben wir die IP-Adresse des des Interfaces ens2 des Hostrouters an. Auch diesen Output speichern wir in ein File ab.

   ```
   $ sudo iperf3 -c 10.0.0.22 >> dateiname.txt
   ```

Da wir die Wegekosten im OSPF überall auf 5 gesetzt haben, erwarten wir, dass sich die Pakete den Weg wie folgt bahnen: R2 → R3 → R5. Kontrollieren können wir das mit `mtr`, was unsere Annahme auch bestätigt:



Da, wie im "*4. Problem der inkonsistenten Wegfindung*" erwähnt, teilweise eine inkonsistente Wegfindung herrscht, prüfen wir mit "*mtr*" dazu noch den Gegenspieler:



[5] pubs.opengroup.org/onlinepubs/9699919799.2016edition/basedefs/V1_chap03.html#tag_03_318

6.1 Manipulation am Netzwerk mit traffic control

Für die Festlegung des Referenzwertes entschieden wir uns für einen fünfminütigen Performancetest. Erst mit TCP, dann mit UDP. Der Outputstream wird in Files abgespeichert.

R5: `$ sudo date >> tcp.txt && iperf3 -s >> tcp.txt`

R2:
```
$ sudo date >> tcp.txt &&
iperf3 -c 10.0.0.22 -t 300 >> tcp.txt
```

Nach 5 Minuten können wir auf R5 mit CRL+C den Test beenden.

Bei UDP ist speziell, dass "*iperf3*" automatisch eine Bandbreite von 1Mbit/s als default festlegt.[6] Deshalb erweitern wir den Befehl, neben dem `-u` für UDP, mit dem `-b` Flag und geben kein Limit für die Bandbreite (`0` steht für unlimitierte Bandbreite):

R5: `$ sudo date >> udp.txt && iperf3 -s >> udp.txt`

R2:
```
$ sudo date >> udp.txt &&
iperf3 -c 10.0.0.22 -t 300 -b 0 -u >> udp.txt
```

Nach 5 Minuten können wir auf R5 mit CRL+C den Test beenden.

Beim TCP Performance Test sendet der Client exakt fünf Minuten lang. Auf Serverseite wird ist dabei das Zeitfenster des Empfangs wegen des Delays etwas höher. In unserem Fall waren es 300.04 Sekunden. Wendet man die folgende Formel an, so haben Server und Client unterschiedliche Bandbreiten:

$$Bandbreite = \frac{Summe\ empfangener\ Daten}{\Delta\ Zeit}$$

Wir entscheiden uns arbiträr für die Bandbreite des Servers als Referenzwert.

Somit erhalten wir folgende Referenzwerte:

TCP: 5.95 Gbits/s

UDP: 2.16 Gbits/s, Jitter: 0.188ms

Die Diskrepanz zwischen UDP und TCP ist erheblich. Unsere Recherchen ergaben, dass speziell in unseren Setup mit Ubuntu-Servern andere dasselbe messen. Als Gründe werden Nebeneffekte der Linux-Kerneloptimierung auf low latency im UDP-Processing genannt.[7]

[6] "iPerf3 and iPerf2 user documentation - iPerf." https://iperf.fr/iperf-doc.php. Accessed 3 Oct. 2020.
[7] "Why is UDP slower than TCP on Ubuntu Server? - Server Fault." https://serverfault.com/questions/432101/why-is-udp-slower-than-tcp-on-ubuntu-server. Accessed 3 Oct. 2020.

6.2 Packet Loss

Als erstes simulieren wir einen packet loss von einem Prozent. Dafür verwenden wir folgenden Befehl:

R3: `$ sudo tc qdisc add dev ens4 root netem loss 1%`

Nun messen wir analog mit den Befehlen im Kapitel *0. Festlegung der Referenzwerte* die Performance, jeweils mit TCP und UDP. Danach ändern wir die Prozentzahlen auf der qdisc und messen erneut. Die Änderung der qdisc wird wie folgt realisiert:

R3: `$ sudo tc qdisc change dev ens4 root netem loss 3%`

Nach den Tests wird die Konfiguration, bzw. die qdisc wie folgt gelöscht:

R3: `$ sudo tc qdisc del dev ens4 root`

Messwerte & Vergleich zu Referenzwerten:

Rohdaten vorhanden auf R2 & R5 jeweils in `/etc/root/performance_tests/`						
packet loss	TCP [bits/s]	Δ TCP	UDP [Gbit/s]	Δ UDP	UDP Jitter [ms]	Δ UDP Jitter
1%	1.51 G	25.38%	2.35	108.5%	0.101	0.53
3%	241 M	4.05%	2.21	102.3%	0.328	1.7
10%	15.1 M	0.35%	2.16	100%	0.599	3.2
30%	439 K	0.01%	2.12	98.15%	10.974	58
50%	109 K	< 0.01%	2.23	103.2%	14.782	78
95%	*Fehler: no route to host*					

wobei immer gilt: $\Delta\, Messgrösse = \frac{Gemessener\ Wert}{Referenzwert}$

Wir gehen davon aus, dass in der letzten Messung (95% packet loss) das OSPF ebenfalls Probleme in der Konvergenz aufweiste und wahrscheinlich über das dead Intervall hinaus keine Typ 1 LSA von zwischen R5 und R3 ausgetauscht wurden.

Aus CN1 wissen wir, dass TCP, aufgrund des Congestion Control, viel empfindlicher auf Störungen im Netzwerk reagiert. Wie in der Theorie, sowie aus unseren Daten geht hervor, dass sich der packet loss in % im Verhältnis zur TCP-Bandbreite logarithmisch verhält:

TCP Veranschaulichung: Y-Achse 100% Wert ist der Referenzwert

Bei allen UDP-Messungen auf Serverseite (R5) hat "*iperf3*" wir eine Bandbreite von 0bit/s gemessen. Wir gehen davon aus, dass diese Anomalie ein Bug in "*iperf3*" ist. Aufgrund dieser Beobachtung werden wir in diesem Dokument die Tabellen nur die Messwerten des Clients verwenden. Nichtsdestotrotz sind sind die Rohdaten in R5 ebenfalls abgespeichert.

```
root@R5:~/performance_tests/packet_loss# tail one_percent_udp.txt
[  5]   8.00-9.00   sec  0.00 Bytes  0.00 bits/sec  0.075 ms  0/0 (0%)
[  5]   9.00-10.00  sec  1.88 MBytes  15.7 Mbits/sec  0.101 ms  65367/65607 (1e+02%)
[  5]  10.00-10.26  sec  0.00 Bytes  0.00 bits/sec  0.101 ms  0/0 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-10.26  sec  0.00 Bytes  0.00 bits/sec  0.101 ms  327250/328466 (1e+02%)
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
iperf3: interrupt - the server has terminated
```

Datenanomalie auf R5

Die Spalte `LOST/TOTAL` in UDP Performancetests haben wir in den Messungen komplett ignoriert, da sie sowohl clientside, wie auch serverside anomal wirken.

6.3 Delay

Delays werden wie folgt realisiert:

R3: `$ sudo tc qdisc add dev ens4 root netem delay 5ms`

Delay mit Normalverteilung, Beispiel mit *μ = 100ms, σ = 10ms ⇔ N(100ms, 10ms)*

R3: `$ sudo tc qdisc change dev ens4 root netem delay 500ms 100ms distribution normal`

Wie bereits im letzten Unterkapitel, löschen wir die qdisc nach unseren Messungen.

Messwerte & Vergleich zu Referenzwerten:

Rohdaten vorhanden auf R2 & R5 jeweils in `/etc/root/performance_tests/`						
Delay	TCP [bit/s]	Δ TCP	UDP [Gbit/s]	Δ UDP	Jitter [ms]	Δ Jitter
5 ms	2.98 G	50.08%	2.03	94%	0.127	0.68
10 ms	1.59 G	26.72%	2.02	94%	0.232	1.23
50 ms	241 M	4.05%	2.30	106%	0.258	1.37
100 ms	25.5 M	0.43%	2.19	101%	0.171	0.9
250 ms	6.07 M	0.10%	2.37	110%	0.699	3.7
500 ms	3.02 M	0.05%	2.40	111%	0.933	5
N (500ms, 100ms)	644 K	0.01%	2.04	94%	554.167	2947
N (700ms, 250ms)	162 K	< 0.01%	2.40	111%	**0.00**	**-**

- Den Jitter der letzten Messung (Normalverteilung mit *μ = 100ms, σ = 10ms*) interpretieren wir als nicht repräsentativ.
- Erneut sehen wir, dass die Bandbreite des UDPs nicht markant gestört wird. In real-time Applikationen, wie z.B. Videotelefonie, würde der Delay und Jitter die Nutzbarkeit erheblich einschränken.

2.2 Pingen mit konfigurierten Delay:

Währenddem ein Delay auf R3 konfiguriert ist, pingen wir von R2 → R5 jeweils mit 20 Paketen und speichern dies ebenfalls ab. Der Befehl hierfür lautet:

R2: `$ ping -c20 >> one_ping.txt`

Bei den konstanten Delays kann man gut beobachten, dass der Mittelwert sowohl das Minimum des Delays immer über den konfigurierten Delay liegen. Die Differenz des konfigurierten Delays und des tatsächlichen Delays stellt dabei der Roundtrip dar. Ausserdem ist zu entnehmen, dass der `tc` command nur in eine Richtung filtert:

Delay von 5ms:
```
0 packets transmitted, 20 received, 0% packet loss, time 19034ms
rtt min/avg/max/mdev = 5.551/6.372/11.590/1.337 ms
```

Delay von 10ms:
```
20 packets transmitted, 20 received, 0% packet loss, time 19037ms
rtt min/avg/max/mdev = 10.563/11.056/15.177/0.974 ms
```

Delay von 50ms:
```
20 packets transmitted, 20 received, 0% packet loss, time 19019ms
rtt min/avg/max/mdev = 50.648/51.714/57.243/1.551 ms
```

Delay von 100ms:
```
20 packets transmitted, 20 received, 0% packet loss, time 19030ms
rtt min/avg/max/mdev = 100.642/101.340/102.946/0.782 ms
```

Delay von 250ms:
```
20 packets transmitted, 20 received, 0% packet loss, time 19026ms
rtt min/avg/max/mdev = 250.628/252.164/257.146/1.779 ms
```

Delay von 500ms:
```
20 packets transmitted, 20 received, 0% packet loss, time 19025ms
rtt min/avg/max/mdev = 500.565/501.616/505.159/1.226 ms
```

Anders verhält es sich bei den normalverteilten Delays. Je mehr Päcken man abschickt, desto eher konvergiert das Arithmetische Mittel gegen die Erwartungswerte (500 und 700)

Delay von N(500ms,100ms):
```
20 packets transmitted, 20 received, 0% packet loss, time 19028ms
rtt min/avg/max/mdev = 314.604/474.334/675.009/100.572 ms
```

Delay von N(700ms,250ms):
```
20 packets transmitted, 20 received, 0% packet loss, time 19008ms
rtt min/avg/max/mdev = 380.906/720.925/1202.202/191.547 ms, pipe 2
```

6.4 Packet Corruption

Mit folgendem Befehl wird in 1% aller Pakete ein zufälliges Bit geflippt:

R3: `$ sudo tc qdisc add dev ens4 root netem corrupt 1%`

Nun führen wir Messungen durch und speichern sie in Dateien, analog zu den vorherigen Messungen. Danach ändern wie die qdisc und passen die Anzahl korrupter Pakete an:

R3: `$ sudo tc qdisc add dev ens4 root netem corrupt 3%`

Analog wie vorhin löschen wir nach Gebrauch die qdisc erneut.

Messwerte & Vergleich zu Referenzwerten:

Rohdaten vorhanden auf R2 & R5 jeweils in `/etc/root/performance_tests/`						
Korruptionsrate	TCP [bit/s]	Δ TCP	UDP [Gbit/s]	Δ UDP	Jitter [ms]	Δ Jitter
1%	1.44 GB	24.20%	2.14	99.07%	0.065	0.35
3%	254 M	4.27%	2.22	102.78%	0.082	0.44
7%	47.8 M	0.80%	2.56	118.52%	0.393	2
15%	5.62 M	0.09%	2.26	104.63%	20.104	106
50%	162 K	< 0.01%	2.43	112.50%	13.589	72

Wir haben speziell den Ping untersucht, bei der grösstmöglichen Korruptionsrate von 100%.

```
$ ping -c100 10.0.0.22
```

Wir vermuteten, dass gewisse Päckchen dennoch durchkommen, beispiel durch Korrekturen. Tatsächlich: Von 100 Päckchen kamen 4 zurück:

```
100 packets transmitted, 4 received, 96% packet loss, time
101258ms
rtt min/avg/max/mdev = 1.130/509.153/2030.851/878.552 ms, pipe 2
```

6.5 Duplicates

Duplikate werden mit `netem duplicate` definiert:

R3: `$ sudo tc qdisc add dev ens4 root netem duplicate 5%`

Für weitere Messungen ändern wir die qdisc wie in den vorherigen Aufgaben:

R3: `$ sudo tc qdisc change dev ens4 root netem duplicate 15%`

Nachdem wir unsere Messungen vollzogen haben, löschen wie die qdisc, wie in den vorherigen Aufgaben.

Messwerte & Vergleich zu Referenzwerten:

Rohdaten vorhanden auf R2 & R5 jeweils in `/etc/root/performance_tests/`						
Duplikationsrate	TCP [bit/s]	Δ TCP	UDP [Gbit/s]	Δ UDP	Jitter [ms]	Δ Jitter
5%	5.08 G	85.38%	2.18	100.93%	0.176	0.93
15%	5.04 G	84.71%	2.17	100.46%	0.316	1.6
50%	4.46 G	74.96%	2.23	103.24%	115609438510	> 10^11
100%	3.50 G	58.82%	2.15	99.54%	77244501694	> 10^11

Der Jitter UDP Datagramme ist unglaublich hoch. Die von "*iperf3*" gemessene Zeit ist sogar höher als die Messung selbst (10s). Hinzu kommt, dass die Anzahl der versendeten Datagramme (auch UDP) ebenfalls unwahrscheinlich hoch ist, bei der letzten Messung haben wir sogar einen Negativen Wert bekommen:

```
root@R2:~/performance_tests/duplicates# cat hundred_udp.txt
Sat Oct  3 02:43:01 UTC 2020
Connecting to host 10.0.0.22, port 5201
[  4] local 10.0.0.17 port 47672 connected to 10.0.0.22 port 5201
[ ID] Interval           Transfer     Bandwidth       Total Datagrams
[  4]   0.00-1.00   sec   268 MBytes  2.24 Gbits/sec  34350
[  4]   1.00-2.00   sec   261 MBytes  2.20 Gbits/sec  33470
[  4]   2.00-3.00   sec   290 MBytes  2.43 Gbits/sec  37180
[  4]   3.00-4.00   sec   252 MBytes  2.12 Gbits/sec  32270
[  4]   4.00-5.00   sec   311 MBytes  2.61 Gbits/sec  39760
[  4]   5.00-6.00   sec   225 MBytes  1.88 Gbits/sec  28740
[  4]   6.00-7.00   sec   235 MBytes  1.97 Gbits/sec  30020
[  4]   7.00-8.00   sec   219 MBytes  1.84 Gbits/sec  28090
[  4]   8.00-9.00   sec   233 MBytes  1.96 Gbits/sec  29880
[  4]   9.00-10.00  sec   267 MBytes  2.24 Gbits/sec  34140
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  4]   0.00-10.00  sec  2.50 GBytes  2.15 Gbits/sec  77244501694.589 ms  -1016506285/-1016506284 (0%)
[  4] Sent -1016506284 datagrams

iperf Done.
```

Auch auffällig bei UDP ist, dass sich die Bandbreite nicht ändert. Wir sehen in den Messungen der UDP Datagramme ein sehr unrepräsentatives Resultat.

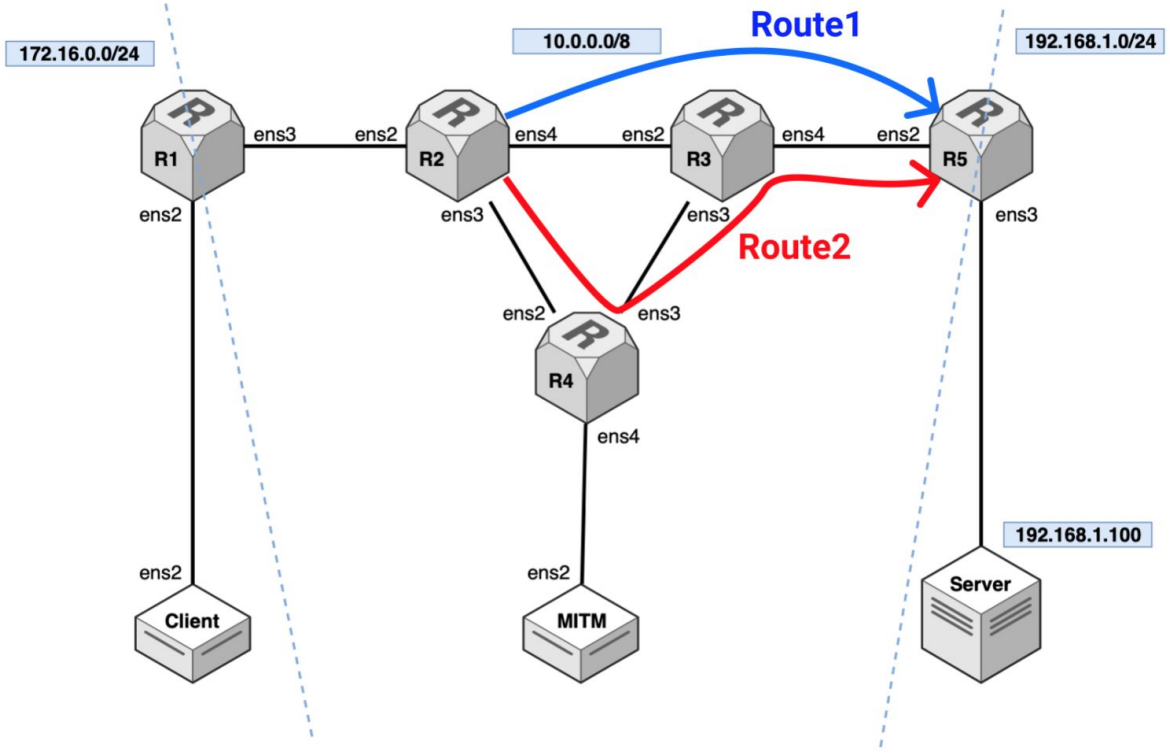
Wohingegen bei TCP sich alles verhält, wie wir es vermuteten. Die Bandbreite nimmt hier linear mit der Netzauslastung der Duplikate ab. TCP ist nicht anfällig bezüglich Duplikate.

6.6 Versuch ECMP R2 → R5 einzurichten für Performance Tests

Nach R5 existieren zwei Routen.

- Route 1: R2 → R3 → R5
- Route 2: R2 → R4 → R3 → R5

Wir dachten uns, dass es interessant wäre ein Setup zu haben, in welchem die Pfadkosten von Route 1 und Route 2 dieselbe wäre.



Uns hätte interessiert, ob es zwischen R3 und R5 mit ECMP ein Bottleneck entstanden wäre, da R2 jeweils auf ens3 und ens4 die volle Bandbreite hätte nutzen können und der Traffic im ens4 von R3 zusammenkommen würde.

In der bisherigen OSPF-Konfiguration ist es so, dass die zweite Route die teurere darstellt. Nun manipulieren wir die Kosten so, dass die beiden Routen dieselbe Metrik besitzen.

Als wir die Wegkosten so manipuliert haben, dass Route1 und Route2 dieselbe Metrik hatten, konnten wir leider mit “*tcpdump*” keinen gesplitteten Traffic beobachten können.

Wir wissen nicht ob es an unserer Konfiguration liegt oder ob die inkonsistente Wegfindung Nebeneffekte herbeigerufen hat.

Daher haben wir diesen Ansatz nicht weiter verfolgt.

7. Aufsetzen des Firewalls

Deaktivieren von Iptables

Da wir in dieser Aufgabe mit dem neueren "*nftables*" arbeiten, müssen wir sicherstellen, dass das durch "*nftables*" abgelöste "*iptables*" deaktiviert wird.

Wir tippen folgende Befehle in die Kommandozeile ein.

R5: `$ sudo iptables -F`

sowie

R5: `$ sudo ip6tables -F`

Nun sollte "iptables" deaktiviert sein.

Initialer Nmap Scan

Als Ausgangslage erstellen wir einen initialen Portscan.

Wir verbinden uns über das Webinterface mit dem Client (172.16.0.2).

Nun scannen wir den Server mit "nmap":

Client: `$ nmap 192.168.1.100`

```
Starting Nmap 7.60 ( https://nmap.org ) at 2020-10-03 17:59 UTC
Nmap scan report for 192.168.1.100
Host is up (0.0073s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE
8080/tcp open  http-proxy

Nmap done: 1 IP address (1 host up) scanned in 0.17 seconds
```

Mit Service Entdeckung:

Client: `$ nmap -sV --version-intensity 5 192.168.1.100`

```
Starting Nmap 7.60 ( https://nmap.org ) at 2020-10-03 18:06 UTC
Nmap scan report for 192.168.1.100
Host is up (0.0052s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE VERSION
8080/tcp open  http    Werkzeug httpd 0.15.2 (Python 3.4.2)

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 6.78 seconds
```

Einrichten der Firewall

Wir möchten die Firewall einrichten, welche mit "*nftables*" aufgebaut wird.

Wir editieren die Konfigurationsdatei

R5: `$ sudo nano /etc/nftables.conf`

in die folgende Form:

```
#!/usr/sbin/nft -f

flush ruleset

define SAFE_TRAFFIC_IPS = {
    172.16.0.0/24,192.168.1.0/24
}

table inet firewall {
    chain inbound {
        type filter hook input priority 0; policy drop;

        # established/related connections
        ct state established,related accept

        ip protocol 89 accept

        iifname lo accept

        tcp dport 22 accept

        log prefix "[nftables] Inbound Denied: " flags all counter drop
    }

   chain forward {
        type filter hook forward priority 0; policy drop;

        ct state established,related accept

        tcp dport { 8080 } ip saddr $SAFE_TRAFFIC_IPS accept
        udp dport { 8080 } ip saddr $SAFE_TRAFFIC_IPS accept

        tcp flags & (syn|fin) == (syn|fin) drop
        tcp flags & (syn|rst) == (syn|rst) drop
        tcp flags & (fin|rst) == (fin|rst) drop
        tcp flags & (ack|fin) == fin drop
        tcp flags & (ack|psh) == psh drop
        tcp flags & (ack|urg) == urg drop

        log prefix "[nftables] Forward Denied: " flags all counter drop
   }

   chain outbound {
        type filter hook output priority 0; policy accept;
   }
}
```

Erklärung der Konfigurationsdatei

Regel	Erläuterung
`type filter hook input priority 0; policy drop;`	*(verwerfen)* was nicht den anderen Filterkriterien entspricht.
`ct state established,related accept`	*(akzeptieren)* der bereits etablierten Verbindungen
`ip protocol 89 accept`	*(akzeptieren)* OSPF LSA Traffic.[8]
`iifname lo accept`	*(akzeptieren)* Traffic des "loopback" Interfaces
`tcp dport 22 accept`	*(akzeptieren)* der SSH Verbindungen
`log prefix "[nftables] Inbound Denied: " flags all counter drop`	*(Logging, Inbound)* aller Pakete, welche verworfen wurden
`log prefix "[nftables] Forward Denied: " flags all counter drop`	*(Logging, Forward)* aller Pakete, welche verworfen wurden
`tcp dport { 8080 } ip saddr $SAFE_TRAFFIC_IPS accept` `udp dport { 8080 } ip saddr $SAFE_TRAFFIC_IPS accept`	*(erlauben)* der Anfragen auf Port 8080, für die definierten IP Ranges ($SAVE_TRAFFIC_IPS)

`define SAFE_TRAFFIC_IPS = {`
`  172.16.0.0/24,192.168.1.0/24`
`}`
	Definierte IP's, siehe oben

`tcp flags`	Versuch, Stealth Scans zu unterbinden

[8] "Open Shortest Path First - Wikipedia." https://en.wikipedia.org/wiki/Open_Shortest_Path_First. Accessed 4 Oct. 2020.

Laden der neuen Konfiguration

Nach dem Populieren der neuen Konfigurationsdatei lässt sich diese laden mit:

R5: `$ sudo systemctl restart nftables.service`

Falls das Neustarten des Services erfolgreich sein sollte, kann dies via "*systemctl*" festgestellt werden:

R5: `$ sudo systemctl status nftables.service`

```
● nftables.service - nftables
   Loaded: loaded (/lib/systemd/system/nftables.service; enabled; vendor preset:
   Active: active (exited) since Sun 2020-10-04 11:12:52 UTC; 1h 41min ago
```

Überprüfen der Firewall

Dank unserer Regeln werden die verworfenen Pakete geloggt.

Wir sehen nun allfällige Probleme.

R5: `$ sudo tail -f /var/log/syslog`

```
Oct  x xx:xx:xx R5 systemd[1]: Starting nftables...
Oct  x xx:xx:xx R5 systemd[1]: Started nftables.
```

Die Konfiguration sieht gut aus, keine Kommunikation wurde bis anhin geblockt.

Tests

Webseite

Wir verbinden uns mit dem Client.

Als Erstes versuchen wir, die Webseite anzufordern.

```
$ curl 192.168.1.100:8080
```

```
<h1>Hello Docker Swarm!</h1><hr><p>A simple hello web app, in a docker image, using debian,
python and flask!</p><p>Hostname: pyt
hon-flask-hello</p><p>Requests: 137<br \></p><p>Network Interface: eth0<br \>--> IP
Address: 192.168.1.100<br \>--> Netmask: 255.255.255.0<br \></p>
```

Die Webseite ist nach wie vor abrufbar.

Nmap

Wir überprüfen nun, welche Ports sichtbar sind.

```
$ nmap 192.168.1.100
```

```
Starting Nmap 7.60 ( https://nmap.org ) at 2020-10-04 xx:xx UTC
Note: Host seems down. If it is really up, but blocking our ping probes, try -Pn
Nmap done: 1 IP address (0 hosts up) scanned in 3.08 seconds
```

Wie ersichtlich ist, sind keine Ports sichtbar.

Ping

Pings sollten gemäss Aufgabenstellung blockiert werden:

```
$ ping 192.168.1.100
```

```
PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
.....
```

Anscheinend erhalten wir keine Antwort.

Ein kurzer Blick in die Logfiles von R5 zeichnen folgendes Bild.

```
Oct  4 14:06:39 R5 kernel: [1466075.375599] [nftables] Forward Denied: IN=ens2 OUT=ens3
MACSRC=52:54:00:01:8b:78 MACDST=52:54:00:
11:c0:a1 MACPROTO=0800 SRC=172.16.0.2 DST=192.168.1.100 LEN=84 TOS=0x00 PREC=0x00 TTL=60
ID=15439 DF PROTO=ICMP TYPE=8 CODE=0 ID=
31279 SEQ=175
Oct  4 14:06:40 R5 kernel: [1466076.399315] [nftables] Forward Denied: IN=ens2 OUT=ens3
MACSRC=52:54:00:01:8b:78 MACDST=52:54:00:
11:c0:a1 MACPROTO=0800 SRC=172.16.0.2 DST=192.168.1.100 LEN=84 TOS=0x00 PREC=0x00 TTL=60
ID=15442 DF PROTO=ICMP TYPE=8 CODE=0 ID=
31279 SEQ=176
```

Die Pings werden erfolgreich geblockt.

Webserver Anfrage ausserhalb des Client-Netzwerkes blocken

Wir verbinden uns beispielsweise mit R4

```
$ curl 192.168.1.100:8080
```

Keine Antwort wird erhalten.

Die Logfiles sind aufschlussreich:

```
$ cat /var/log/syslog
```

```
Oct  4 14:14:42 R5 kernel: [1466558.439875] [nftables] Forward Denied: IN=ens2 OUT=ens3
MACSRC=52:54:00:01:8b:78 MACDST=52:54:00:
11:c0:a1 MACPROTO=0800 SRC=10.0.0.13 DST=192.168.1.100 LEN=60 TOS=0x00 PREC=0x00 TTL=62
ID=55313 DF PROTO=TCP SPT=59240 DPT=8080
SEQ=566590489 ACK=0 WINDOW=29200 RES=0x00 SYN URGP=0 OPT
(020405B40402080A53D060830000000001030307)
```

Die Anfrage wurde erfolgreich geblockt.

Stealth Scans (Client)

Nmap ermöglicht es dem Benutzer ebenfalls, Stealth Scans auszuführen, mit der "-Pn" Flagge.

Die Stealth Scans konnten wir auf dem Client leider nicht blocken.

Client: `$ nmap -Pn 192.168.1.100`

```
Starting Nmap 7.60 ( https://nmap.org ) at 2020-10-04 14:27 UTC
Nmap scan report for 192.168.1.100
Host is up (0.0057s latency).
Not shown: 999 filtered ports
PORT     STATE SERVICE
8080/tcp open  http-proxy

Nmap done: 1 IP address (1 host up) scanned in 8.23 seconds
```

Stealth Scans (sonst)

Die sonstigen Router erhalten keine Rückmeldung.

R1-5: `$ nmap -Pn 192.168.1.100`

```
Nmap scan report for 192.168.1.100
Host is up.
All 1000 scanned ports on 192.168.1.100 are filtered

Nmap done: 1 IP address (1 host up) scanned in 201.37 seconds
```

Firewall Logs:

```
Oct  4 14:33:29 R5 kernel: [1467685.208644] [nftables] Forward Denied: IN=ens2 OUT=ens3
MACSRC=52:54:00:01:8b:78 MACDST=52:54:00:
11:c0:a1 MACPROTO=0800 SRC=10.0.0.17 DST=192.168.1.100 LEN=44 TOS=0x00 PREC=0x00 TTL=46
ID=63334 PROTO=TCP SPT=46858 DPT=4444 SEQ
=904648651 ACK=0 WINDOW=1024 RES=0x00 SYN URGP=0 OPT (020405B4)
```

8. Man in the Middle

MITM als Proxy

Vorbedingungen

1. In einem ersten Schritt muss OSPF auf der MITM Maschine aufgesetzt sein.
    a. Die Schritte dazu sind unter "Einrichtung von BIRD (Aufsetzen von OSPF" zu finden
2. Als nächstes muss ipv4 Routing aktiviert werden
    a. Analog zu "Einrichten von IPV4-Forwarding"
3. Wir verbinden uns mit dem MITM Rechner per SSH

Ziel

Wir möchten zuerst erreichen, dass wir den Traffic des Clients einsehen können.
Das Tool "*mitmdump*"ermöglicht uns, Traffic aufzuzeichnen, welcher über unseren Computer/Server agiert.

Wir geben folgenden command ein:

R5: `$ mitmproxy --port 8080`

Der komplette Traffic, welcher über den Port 8080 einfliesst ist nun ersichtlich.

Problem 1

Der Client sendet seinen Request direkt an den Webserver, der Traffic erreicht unseren Man in the Middle (R5) nicht.

Wir nehmen an, dass der Angreifer eine Möglichkeit gefunden hat, einen Proxy (den Man in the Middle) auf dem Client zu definieren.

Mit dem folgenden Befehl

Client: `$ curl -x 10.0.0.10:8080 -L 192.168.1.100:8080`

definieren wir unseren Man in The Middle als Proxy.

Alternative MITM Taktiken

Alias

Alternativ können wir einen Alias für "*curl*" setzen, welcher jedes mal ausgeführt wird, wenn wir "*curl*" in die Kommandozeile eingeben.

```
$ nano ~/.bashrc
```

```
…
alias curl="curl -x 10.0.0.10:8080"
…
```

Config File editieren

1. Die Konfigurationsdatei für "*curl*" editieren
2. `$ nano ~/.curlrc`
3. Wir fügen folgende Zeile hinzu

```
…
proxy = 10.0.0.10:8080
..
```

Exportieren der Variablen

```
$ export https_proxy=10.0.0.10:8080
$ export http_proxy=10.0.0.10:8080
```

Diese Exports können auch persistiert werden, idem sie in

```
$ sudo -H nano /etc/environment
```

geschrieben werden.

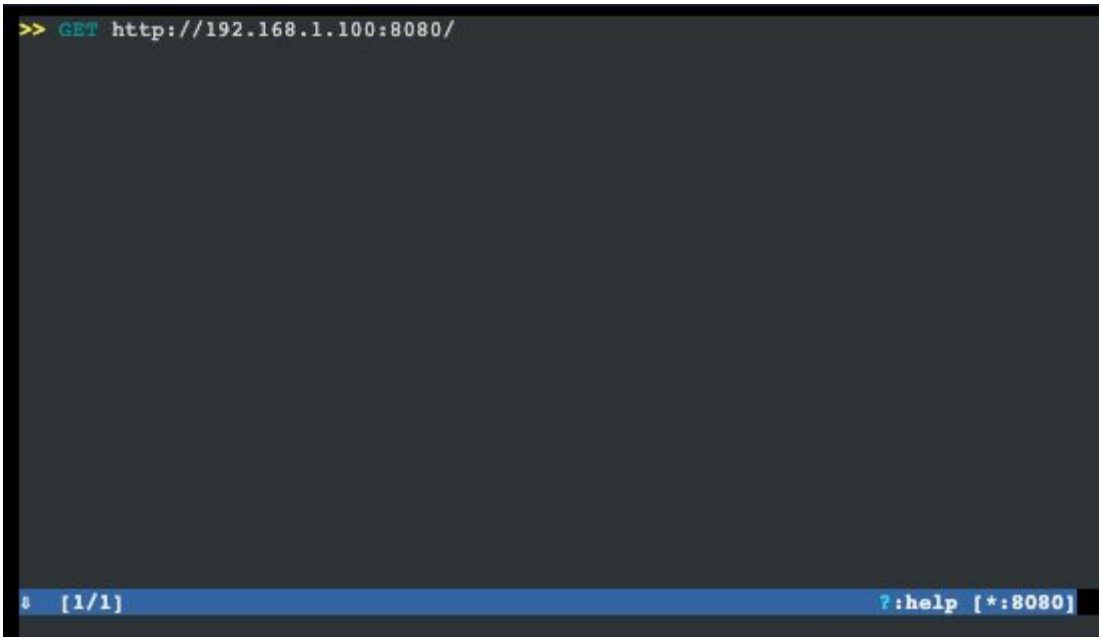
"*Curl*" benutzt nun standardmässig unseren MITM als Proxy.

Proxy als Variable mitgeben

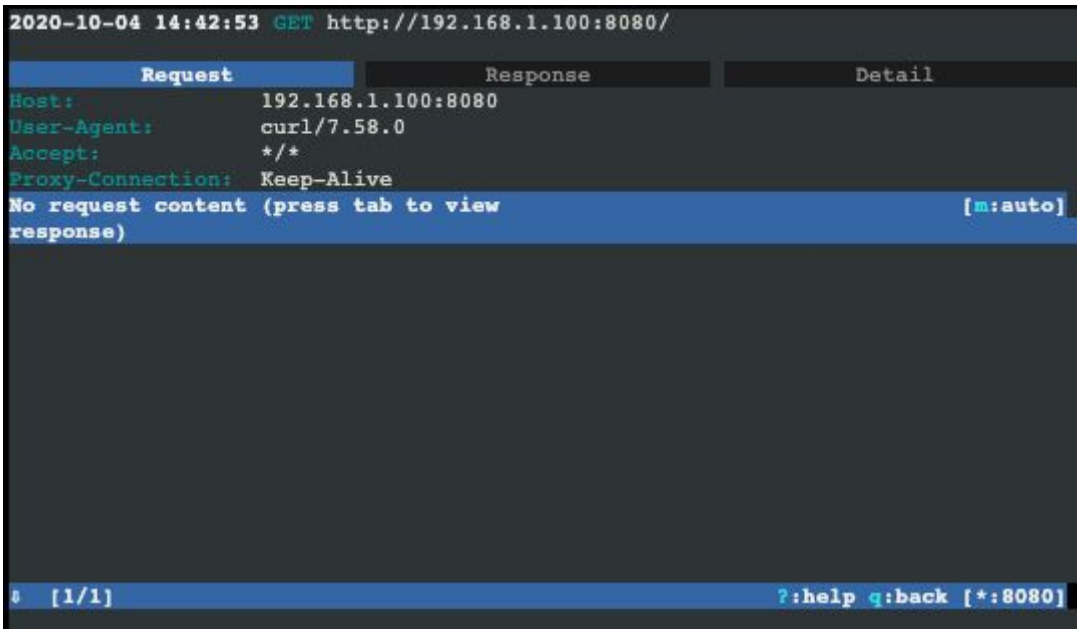
Der Proxy kann auch als Variable definiert werden und dem Programm direkt übergeben werden.

```
$ http_proxy=10.0.0.10:8080 curl 192.168.1.100:8080
```

MITM



Auf Seite des MITM (R5) sehen wir nun den Request.
Via mitmproxy haben wir die Möglichkeit den Request, wie auch die Response einzusehen, ein erster Erfolg.



Problem 2

Unsere Firewall blockiert alle Anfragen, welche nicht direkt via Client getätigt werden. Da "*mitmproxy*" nur als Mittelsmann fungiert und die Anfrage an den Server weiterleitet, erhält dieser nie eine Antwort.

```
Oct  4 14:47:48 R5 kernel: [1468543.561006] [nftables] Forward Denied: IN=ens2 OUT=ens3
MACSRC=52:54:00:01:8b:78 MACDST=52:54:00:
11:c0:a1 MACPROTO=0800 SRC=10.0.0.10 DST=192.168.1.100 LEN=60 TOS=0x00 PREC=0x00 TTL=61
ID=10386 DF PROTO=TCP SPT=43927 DPT=8080
SEQ=2180671723 ACK=0 WINDOW=29200 RES=0x00 SYN URGP=0 OPT
(020405B40402080A89E4E83700000000001030307)
```

Problem 3

Wir haben vollen Einblick im Verkehr des Clients zum Webserver. Weiter wollen wir eine Webseite "injecten". Via "*mitmdump*" lassen sich eigens geschriebene Skripte einspielen.

Schreiben des Skriptes

Mitmdump[9] besitzt eine Python API. Nach Lesen der Dokumentation[10] betreffend Scripting sind wir in der Lage den Code zu schreiben.

R5: `$ touch fakeRequest.py`
R5: `$ nano fakeRequest.py`

R5: `$ cat fakeRequest.py`

```
from mitmproxy import ctx
from mitmproxy import http
flag=False
def request(flow: http.HTTPFlow):
        ctx.log.info("Faking Destination Address")
        flow.request.host = "10.0.0.10"

def response(flow: http.HTTPFlow):
        flow.response.status_code=200
        flow.response.content=bytes("You just have been intercepted","UTF-8")
        ctx.log.info("RESPONSE SENT")
```

Dieses Programm hört auf jeden HTTP FLOW, also jede Art von Traffic und egal von welcher IP und erstellt danach eine beliebige Antwort.

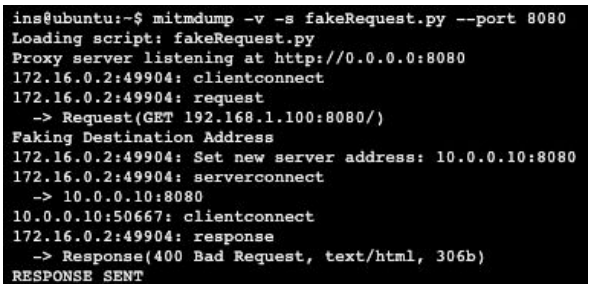
Wir haben unser Script/Addon, als nächstes müssen wir dieses "*mitmdump*" übergeben. Mit dem -s Flag könnten externe Dateien spezifiziert werden.

Run:
`$ mitmdump -s fakeRequest.py --port 8080`

---

[9] "mitmdump — mitmproxy 2.0.2 documentation." https://mitmproxy.readthedocs.io/en/v2.0.2/mitmdump.html. Accessed 3 Oct. 2020.
[10] "Overview — mitmproxy 2.0.2 documentation." https://mitmproxy.readthedocs.io/en/v2.0.2/scripting/overview.html. Accessed 3 Oct. 2020.

"*mitmdump*", verstärkt durch das Script, hört jetzt auf Web Request.



(R5 Attacker View)

Die "gefälschte" Nachricht wird vom Client empfangen.

Client: `$ curl -x 10.0.0.10:8080 -L 192.168.1.100:8080`

`You just have been intercepted`

Problematik der Firewall

"*mitmproxy/dump*" sitzt zwischen dem Client und Server und handelt als Man in the Middle.

1. "*mitmproxy*" empfängt den Request
2. "*mitmproxy*" leitet das Signal weiter an den Webserver
3. Unser Skript editiert live den Content der Antwort
4. Diese Antwort wird wieder an den Clienten zurückgegeben

Normalerweise wird Schritt 2 unterbunden, da die Firewall keine Antwort sendet. "*mitmdump*" hängt sich auf. Wir können dies korrigieren, indem wir in unserem Skript den Host anpassen. Die Anfrage des Clients wird umgeleitet und leitet auf die Angreifer Maschine um. Dies ist nicht weiter schlimm, da wir an der eigentlichen Webseite wenig interessiert sind.

```
def request(flow: http.HTTPFlow):
ctx.log.info("Faking Destination Address")
        flow.request.host = "10.0.0.10"
```

Nun wird der Traffic nicht mehr über die Firewall gesendet und unser Angriff funktioniert trotz Firewall.

Idee: MITM im OSPF Netzwerk

Wenn wir annehmen, das der MITM über ein aktives Interface ans OSPF Netz angeschlossen ist, dann kann er die Route zum Server (192.168.1.100) mit den tiefsten Kosten annoncen. So kann der MITM darauf hoffen, dass er für gewisse Clients den kürzeren Weg darstellt und damit den Traffic zu verlaufen lässt.
In unserem Setup würde das sogar funktionieren, wenn gegeben ist, dass jeder Hop dieselbe Kosten hat. Der MITM ist dem Client um ein Hop näher.

Das würde man realisieren, indem man die `/etc/bird/bird.conf` das Netz 192.168.1.0/24 einspeisen würde, den einkommenden Traffic verarbeiten und eine “gefälschte” Response zurücksenden würde. Mit fortgeschrittenen Tooling wäre es zudem möglich die Source im IP-Header auf den des Servers zu setzen.

Appendix 1: Credentials & Verbindungsinformation

Device	CLI-Befehl für SSH-Verbindung (VPN zum INS vorausgesetzt)
R1	`ssh -p 10109 ins@sr-121651.ltb.ins.hsr.ch`
R2	`ssh -p 10110 ins@sr-121651.ltb.ins.hsr.ch`
R3	`ssh -p 10111 ins@sr-121651.ltb.ins.hsr.ch`
R4	`ssh -p 10112 ins@sr-121651.ltb.ins.hsr.ch`
R5	`ssh -p 10113 ins@sr-121651.ltb.ins.hsr.ch`
MITM	`ssh -p 10115 ins@sr-121651.ltb.ins.hsr.ch`
Client	`ssh -p 10114 ins@sr-121651.ltb.ins.hsr.ch`
Server	`ssh -p 10116 ins@sr-121651.ltb.ins.hsr.ch`
Jumphost	`ssh ins@10.18.10.2`

Device	Username	Passwort
R1	ins	`7SLge45jEzGVbd9K99zwXFs7G`
R2	ins	`7SLge45jEzGVbd9K99zwXFs7G`
R3	ins	`7SLge45jEzGVbd9K99zwXFs7G`
R4	ins	`7SLge45jEzGVbd9K99zwXFs7G`
R5	ins	`7SLge45jEzGVbd9K99zwXFs7G`
MITM	ins	`7SLge45jEzGVbd9K99zwXFs7G`
Client	ins	`7SLge45jEzGVbd9K99zwXFs7G`
Jumphost	ins	`ins@lab`

Appendix 2: IP-Adressplan tabellarisch

Host	Interface	Netzadresse	Netzmaske	IPv4
R1	ens2	172.16.0.0	255.255.255.0	172.16.0.1
	ens3	10.0.0.0	255.255.255.252	10.0.0.1
R2	ens2	10.0.0.0	255.255.255.252	10.0.0.2
	ens3	10.0.0.4	255.255.255.252	10.0.0.5
	ens4	10.0.0.16	255.255.255.252	10.0.0.17
R3	ens2	10.0.0.16	255.255.255.252	10.0.0.18
	ens3	10.0.0.12	255.255.255.252	10.0.0.14
	ens4	10.0.0.20	255.255.255.252	10.0.0.21
R4	ens2	10.0.0.4	255.255.255.252	10.0.0.6
	ens3	10.0.0.12	255.255.255.252	10.0.0.13
	ens4	10.0.0.8	255.255.255.252	10.0.0.9
R5	ens2	10.0.0.20	255.255.255.252	10.0.0.21
	ens3	192.168.1.0	255.255.255.0	192.168.1.1
MITM	ens2	10.0.0.8	255.255.255.252	10.0.0.10
Client	ens2	172.16.0.0	255.255.255.0	172.16.0.2
Server	eth0@if1110	192.168.1.0	255.255.255.0	192.168.1.100

Appendix 3: Referenzblatt

Befehl	Nutzen
sudo	Ausführen von Befehlen als superuser
ssh	OpenSSH Client für remote logins
nano	terminal-based Texteditor
vim	Mächtiger terminal-based Texteditor
touch	Erstellen einer Datei oder den Timestamp ändern
cat	Inhalte von File(s) anzeigen lassen
curl	Multiprotokoltool für Datentransfer von Server zu Client
passwd	Benutzt für Passwortänderungen
ifconfig	Netzwerkkonfiguration anzeigen lassen
ip link	Status der Interfaces abfragen
cp	Kopierbefehl für Dateien und Ordner
systemctl	System Service Manager
route	Routen des Kernels anzeigen lassen.
bridc	Internet routing Daemon
tcpdump	Dumpt den TCP Traffic
iperf3	Tool um Netzwerktests durchzuführen
mtr	Netzwerkdiagnosetool - benutzt um Routen zu tracen.
date	Anzeige der aktuellen Uhrzeit
tc	tc - traffic control, zusammen benutzt mit netem (Netzemulator). Virtuelle Manipulation an Traffic
netem	
ping	Tool um ICMP ECHO_REQUEST zu senden.
iptables	Legacy Administrationtool für IPv4 (auch möglich für IPv6)
nftables	Administrationtool für filtering und classification (benutzt für Firewall)
nmap	Network exploration tool für Portscanning
mitmproxy	mitmproxy/mitmdump ist ein Sicherheits-Werkzeug, welches web traffic abfangen, untersuchen und modifizieren kann
mitmdump	