SCHT - Lab2

Stanisław Kwiatkowski, Bartosz Ziemba

Spis treści

1. Konfiguracja ręczna

By przygotować się do tej części zainstalowaliśmy na swoich maszynach kontroler ONOS oraz testowo skonfigurowaliśmy go na krótkiej trasie Warszawa-Berlin przy użyciu styku REST oraz narzędzia curl.

1.1. Podstawowa konfiguracja

W tym kroku mieliśmy za zadanie powtórzyć eksperymenty wykonywane przy użyciu narzędzia iperf z ostatniego laboratorium. By to się udało trzeba było jednak utworzyć odpowiednie reguły na odpowiednich switchach, mówiące jak zarządzać ruchem w sieci. Wykonaliśmy to wysyłając odpowiednie pliku typu json protokołem http na switche. Poniższe screeny ukazują proces konfiguracji trasy Warszawa-Madryt.

```
mininet> Warszawa ping Madryt -c 4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3066ms
```

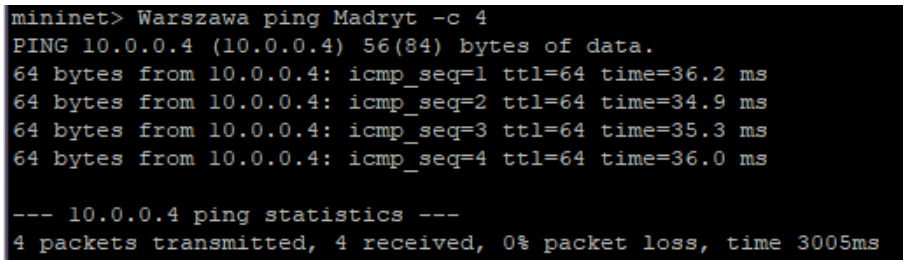
Rys. 1: Nie udany ping przed konfiguracją

Ping nie przechodzi z powodu braku reguł w tablicach.

Rys. 2: Wykonanie pliku konfiguracyjnego

SELECTOR	TREATMENT	APP NAME
ETH_TYPE:arp	imm[OUTPUT:CONTROLLER], cleared:true	*core
ETH_DST:00:00:00:00:00:08, ETH_TYPE:ipv4	imm[OUTPUT:5], cleared:false	*rest
ETH_DST:00:00:00:00:00:04, ETH_TYPE:ipv4	imm[OUTPUT:3], cleared:false	*rest
ETH_TYPE:bddp	imm[OUTPUT:CONTROLLER], cleared:true	*core
ETH_TYPE:lldp	imm[OUTPUT:CONTROLLER], cleared:true	*core

Rys. 3: Tablica przepływów dla s2 (Paryża) po konfiguracji



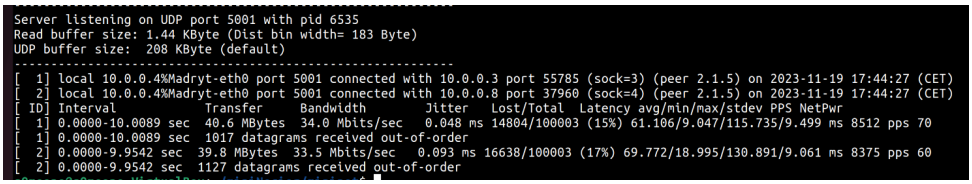
Rys. 4: Udany ping pomiędzy Warszawą a Madrytem

Jak widać udało się nam poprawnie ustawić węzły sieci na tej trasie dzięki czemu hosty Warszawa i Madryt są w stanie się komunikować. Taki sam proces konfiguracji przeprowadziliśmy dla reszty potrzebnych tras.

1.2. Testowanie połączeń

W tej części ćwiczenia przeprowadziliśmy kilka testów naszej nowo skonfigurowanej sieci przy użyciu narzędzia iperf. Testy te odpowiadają tym wykonywanym na laboratorium pierwszym. Mając to na uwadze nie będziemy się rozpisywać o charakterystyce i procesach zachodzących podczas przesyłu danych, a skupimy się tylko na poprawności połączenia i porównaniu z wczesniejszymi wynikami.

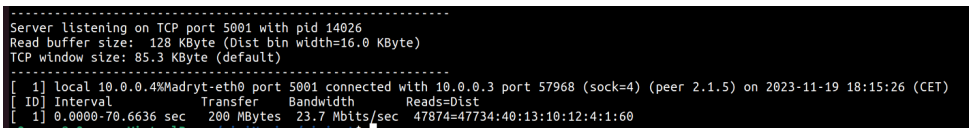
1.2.1. UDP 50% + UDP 50%



Rys. 5: Wynik testu

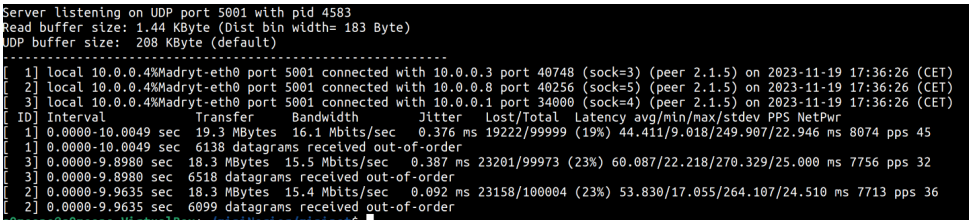
1.2.2. UDP 50% + TCP

Rys. 6: Wynik testu dla serwera TCP



Rys. 7: Wynik testu dla serwera UDP

1.2.3. UDP 25% + UDP 25% + UDP 25%

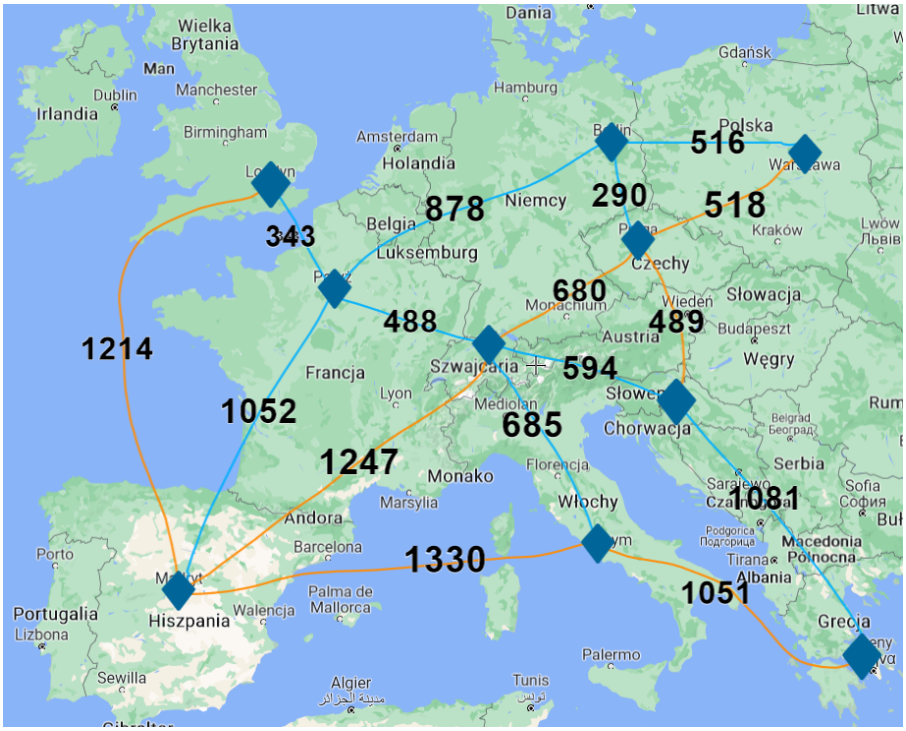


Rys. 8: Wynik testu

Jak widać testy są wyniki przeprowadzonych testów są łudząco podobne do tych otrzymanych w laboratorium pierwszym, co oznacza, że sieć została przez nas skonfigurowana poprawnie.

1.3. Testy na pełnej sieci

Z racji iż w pełnej wersji naszego grafu każda z tras na których odbywa się przesył danych jest od siebie niezależna, możemy się spodziewać znacznej poprawy wydajności, szczególnie tam gdzie na jednej drodze spotykało się wiele strumieni.



Rys. 9: Pełna wersja mapy (pomarańczowym oznaczone nowe połączenia)

1.3.1. UDP 25% + UDP 25% + UDP 25%

```
Server listening on UDP port 5001 with pid 19285
Read buffer size: 1.44 KByte (Dist bin width= 183 Byte)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.1 port 46888 (sock=3) (peer 2.1.5) on 2023-11-19 18:54:28 (CET)
[  2] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.3 port 34890 (sock=4) (peer 2.1.5) on 2023-11-19 18:54:28 (CET)
[  3] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.8 port 34457 (sock=5) (peer 2.1.5) on 2023-11-19 18:54:28 (CET)
[ ID] Interval           Transfer     Bandwidth        Jitter   Lost/Total  Latency avg/min/max/stdev PPS NetPwr
[  2] 0.0000-9.9485 sec  23.6 MBytes  19.9 Mbits/sec   0.101 ms 836/100003 (0.84%) 9.262/8.003/180.411/8.554 ms 9968 pps 269
[  2] 0.0000-9.9485 sec  1150 datagrams received out-of-order
[  1] 0.0000-9.9932 sec  23.7 MBytes  19.9 Mbits/sec   0.124 ms 773/100002 (0.77%) 18.688/17.014/193.924/8.210 ms 9930 pps 133
[  1] 0.0000-9.9932 sec  1024 datagrams received out-of-order
[  3] 0.0000-9.8443 sec  23.6 MBytes  20.1 Mbits/sec   0.158 ms 1125/100000 (1.1%) 19.867/17.040/193.814/11.787 ms 10044 pps 126
[  3] 0.0000-9.8443 sec  2713 datagrams received out-of-order
```

Rys. 10: Wynik testu

1.3.2. UDP 50% + UDP 50%

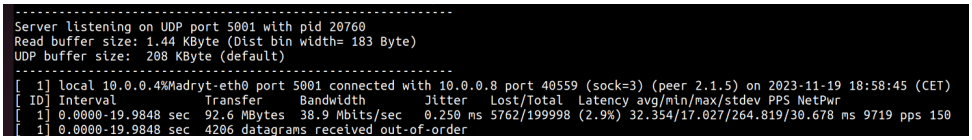
```
------------------------------------------------------------
Server listening on UDP port 5001 with pid 20023
Read buffer size: 1.44 KByte (Dist bin width= 183 Byte)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.3 port 38744 (sock=3) (peer 2.1.5) on 2023-11-19 18:56:26 (CET)
[  2] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.8 port 41343 (sock=5) (peer 2.1.5) on 2023-11-19 18:56:26 (CET)
[ ID] Interval           Transfer     Bandwidth        Jitter   Lost/Total  Latency avg/min/max/stdev PPS NetPwr
[  2] 0.0000-9.9273 sec  47.5 MBytes  40.1 Mbits/sec   0.089 ms 402/100003 (0.4%) 21.839/17.020/75.618/6.992 ms 10033 pps 230
[  2] 0.0000-9.9273 sec  1734 datagrams received out-of-order
[  1] 0.0000-9.9772 sec  47.7 MBytes  40.1 Mbits/sec   0.088 ms 22/100003 (0.022%) 10.116/8.003/57.736/4.376 ms 10021 pps 495
[  1] 0.0000-9.9772 sec  746 datagrams received out-of-order
```

Rys. 11: Wynik testu

1.3.3. UDP 50% + TCP

Istotnie tak jak zakładaliśmy poprawne skonfigurowanie pełnej sieci dało nam znaczy wzrost zarówno wydajności jak i efektywności co objawiło się większą szybkością przesyłu i zmniejszoną utratą pakietów

Rys. 12: Wynik testu serwera TCP



Rys. 13: Wynik testu dla serwera UDP

2. Konfiguracja automatyczna - skrypt w Pythonie

2.1. Analiza programu

Aby móc zrealizować funkcję konfigurowania switchy tak, aby otrzymywać każdorazowo strumienie danych o żądanych parametrach (lub jeśli to nie możliwe to o najlepszych dostępnych), musieliśmy zrealizować kolejne zadania:

- wczytywanie informacji o budowie sieci do systemu
- wczytywanie informacji o żądanych parametrach strumieni
- budowanie najlepszej ścieżki na podstawie dobranych współczynników i funkcji uwzględnionych w wagach krawędzi grafu
- konfiguracja *flows* w ONOSie korzystając ze styku REST

2.1.1. Wczytywanie informacji o budowie sieci do systemu

Skonstruowany przez nas plik json ma następujący format:

```
{
        "devices": [
        {
           "name": "Londyn",
            "ip": "10.0.0.3",
            "links": [
                {
                    "port": 1,
                    "linkId": 1
                },
                ....
            ]
        }
        ....
        ],
  "links": [
        {
            "id": 1,
            "node1": "Londyn",
            "node2": "s1",
            "delay": 0,
            "bw": 1000
        },
        ....
```

```
    ]
}
```

2.1.2. Wczytywanie informacji o żądanych parametrach strumieni

Zdecydowaliśmy się na podawanie ścieżki do pliku, w których zapisane będą oczekiwane parametry strumieni. Struktura jednej linii takiego pliku wygląda następująco:
<źródło> <cel> <TCP/UDP> <maksymalne opóźnienie> <minimalna przepustowość>

2.1.3. Budowanie najlepszej ścieżki na podstawie dobranych współczynników i funkcji uwzględnionych w wagach krawędzi grafu

Wyjściem do odpowiedniego zarządzania ruchem sieciowym w sposób efektywny i zrównoważony, jest zrozumienie procesów zachodzących w sieci podczas transmisji danych. Wiedzę tą zdobyliśmy podczas przeprowadzania testów na ostatnim laboratorium. Dlatego też wiemy, że głównymi czynnikami wpływającymi na jakość połączenia są:

- Opóźnienie -> będące sumą wszystkich opóźnień występującą pomiędzy poszczególnymi switchami na trasie
- Przepustowość -> a w zasadzie tak zwane "wąskie gardło", czyli najmniejsza przepustowość relacji występująca na trasie
- ilość aktywnych sesji -> im więcej sesji musi utrzymać dana ścieżka w sieci tym gorsza jakość

Ponad to wpływ tych parametrów różni się w zależności od protokołu jaki wybraliśmy do transmitowania danych. Mając z tyłu głowy, że nasza sieć ma być jedynie uproszczonym modelem rzeczywistości określiliśmy procentowy wpływ tych parametrów na:

- TCP:
  - Przepustowość - 50%
  - Opóźnienie - 50%
- UDP:
  - Przepustowość - 90%
  - Opóźnienie - 10%

Wiedząc jak wygląda charakterystyka naszej sieci mogliśmy utworzyć równania liniowe, służące nam do obliczania wag na poszczególnych krawędziach grafu dla odpowiedniego protokołu. Skrajne przypadki w naszej sieci:

- Maksymalne opóźnienie = $9ms$
- Minimalne opóźnienie = $3ms$
- Maksymalna przepustowość = $80Mb$
- Minimalna przepustowość = $0Mb$

```python
def calculate_tcp_score(delay, current_bw, active_tcp, active_udp):
    delay_fun = (25/3)*delay-25
    if(active_tcp==0):
        bw_fun = (-5/8)*current_bw +50
    else:
        bw_fun = (-5/8)*(current_bw/active_tcp)+50
    score = delay_fun + bw_fun
    return score
```

Rys. 14: Wyznaczanie wag w grafie dla połączenia TCP

```
def calculate_udp_score(delay, current_bw, active_tcp, active_udp):
    delay_fun = (5/3)*delay-5
    bw_fun = (-9/8) * current_bw + 90
    score = 100 + delay_fun+bw_fun
    return score
```

Rys. 15: Wyznaczanie wag w grafie dla połączenia UDP

Warto podkreślić, że zmienna $current\_bw$ jest pierwotną wartością przepustowości na danej krawędzi pomniejszona o zaplanowane na tej krawędzi sesje UDP, tak więc jeśli początkowa wartość bw na krawędzi między $s2$, a $s1$ jest równa $80Mb$ to jeśli zaplanujemy na tej trasie dwie sesje UDP wykorzystujące zasoby równe $40Mb$ to wartość zmiennej $current\_bw$ wyniesie $40Mb$

Mając odpowiednio zważone krawędzie jesteśmy w stanie skorzystać z algorytmu *Dikstry*, i znaleźć ścieżkę o najniższej wadze dla danego połączenia. Uwzględniając przy tym, czy jest to połączenie TCP, czy UDP. Kiedy znajdziemy już ścieżkę o żądanych parametrach (jeśli takiej nie ma, wysyłamy stosowny komunikat i konfigurujemy po możliwie najlepszej), zapisujemy informacje na każdej z użytych krawędzi o obecności kolejnego strumienia (obniżamy *bandwidth* w przypadku połączeń UDP i zapisujemy informacje o ilości połączeń TCP). Dzięki temu, przy szukaniu ścieżki dla kolejnego połączenia, korzystamy z rzeczywistych parametrów sieci, po uwzględnieniu obecnego obłożenia sieci.

2.1.4. Konfiguracja *flows* w ONOSie korzystając ze styku REST

Gdy posiadamy już spełniającą kryteria ścieżkę do hosta docelowego, po kolei konfigurujemy switche. W tym celu uzupełniamy szablon pojedynczej zasady przepływu, w oparciu o dane sąsiadujących urządzeń. Na typ etapie zorientwaliśmy się, że ONOS nie daje dużych możliwości różnicowania różnych strumieni pakietów poruszających się pomiędzy dwoma stałymi hostami (a właśnie takiego połączenia głównie używaliśmy w poprzednim ćwiczeniu). W związku z tym dodaliśmy kryterium *IP_PROTO*, które określa numer protokołu z którego pochodzi dany pakiet (6 dla TCP, 17 dla UDP, dodatkowo 1 dla ICMP).

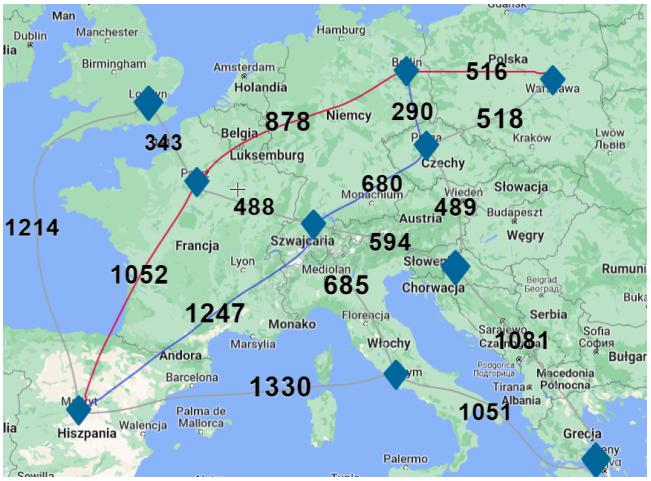
Dodatkowo, skrypt umożliwia dodawanie flag, na chwile obecną są to -f/–file do podawania pliku tekstowego z parametrami zlecanych strumieni oraz -d/–debug do wyświetlania dodatkowych informacji dotyczących błędów.

2.2. Testowanie połączeń przy konfiguracji automatycznej

Niektóre z wyników testów były by identyczne jak w przypadku testów z domyślnym kontrolerem i topologii bez cykli, więc zostały one pominięte. Dodatkowo, testy przeprowadzane z użyciem dwóch takich samych protokołów, z takimi samymi adresami źródłowymi i docelowymi również nie będą miarodajne, ponieważ wszystkie strumienie zostaną potraktowane przez reguły switchy jako jeden (problem opisany w paragrafie 2.1.4). Z uwagi na to, zmodyfikowaliśmy testy tak, aby była możliwość zauważenia poprawy i działania algorytmu.

2.2.1. Warszawa -> Madryt, Berlin -> Madryt

W tym scenariuszu poprowadziliśmy dwa połączenia UDP zajmujące 100% przepustowości dowolnego z łącz (80 Mbit/s), ale dzięki rozszerzeniu grafu i optymalizacji dróg przez nasz skrypt, połączenia zostały poprowadzone tak jak na obrazkach 16 i 17.



Rys. 16: Trasa połączeń (czerwonym z Warszawy, niebieskim z Berlina)

```
└─$ /bin/python3 /home/kali/SCHT/lab2/cli.py -f connections.txt
Attempting to configure flow rules for the path:
Warszawa -> s10 -> s8 -> s2 -> s3 -> Madryt
[INFO] Successfully added 8 flows to switches.

Attempting to configure flow rules for the path:
Berlin -> s8 -> s9 -> s4 -> s3 -> Madryt
[INFO] Successfully added 8 flows to switches.
```

Rys. 17: Wykonanie skryptu obliczającego najlepsze ścieżki

Skrypt wykrył, że w przypadku poprowadzenia tych połączeń tą samą ścieżką, nastąpiłoby przeciążenie łącza oznaczające dużą stratę pakietów, i pokierował drugie połączenie (Berlin - Madryt) inną drogą, która była minimalnie dłuższa.

Po uruchomieniu skryptów wykorzystujących iperf, otrzymaliśmy dokładnie takie same wyniki, jak w przypadku zestawienia tych połączeń w różnym od siebie czasie (utrata do 30% pakietów jest typowa dla połączenia UDP zajmującego całą szerokość łącza, co wykazaliśmy w poprzednim ćwiczeniu laboratoryjnym).

```
┌──(kali㉿kali)-[~]
└─$ cat MadrytServer
------------------------------------------------------------
Server listening on UDP port 5001 with pid 34053
Read buffer size: 1.44 KByte (Dist bin width= 183 Byte)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.8 port 55608 (sock=3) (peer 2.1.9) on 2023-11-19 10:27:10.831 (EST)
[  2] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.2 port 33395 (sock=4) (peer 2.1.9) on 2023-11-19 10:27:10.874 (EST)
[ ID] Interval           Transfer     Bandwidth        Jitter   Lost/Total  Latency avg/min/max/stdev PPS NetPwr
[  1] 0.0000-10.1389 sec  51.1 MBytes  42.3 Mbits/sec   0.342 ms 25305/78934 (32%) 179.932/21.243/278.526/33.524 ms 5289 pps 29.40
[  1] 0.0000-10.1389 sec  1347 datagrams received out-of-order
[  2] 0.0000-10.1772 sec  51.5 MBytes  42.4 Mbits/sec   0.344 ms 26468/80439 (33%) 173.801/14.432/270.514/46.933 ms 5303 pps 30.51
[  2] 0.0000-10.1772 sec  1360 datagrams received out-of-order
```

Rys. 18: Wyniki skryptów iperfowych

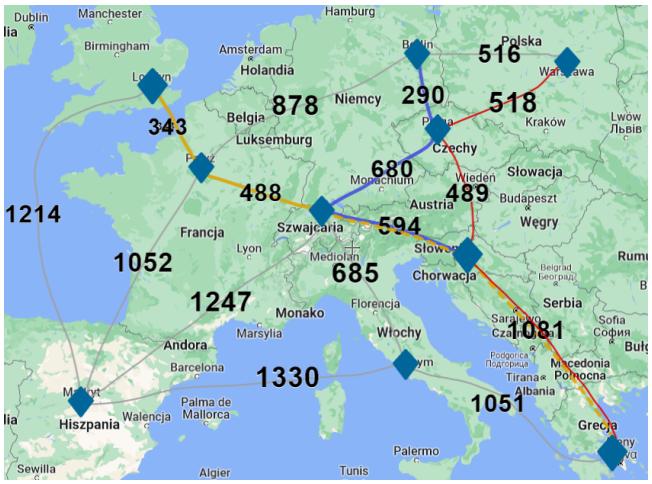
2.2.2. Scenariusz zaawansowany 1

W tym teście sprawdziliśmy ustalone ścieżki i wydajność dla następujących strumieni:

	źródło	cel	protokół	min. przepustowość	max. ping
1	Warszawa	Ateny	UDP	70	50
2	Berlin	Zagrzeb	TCP	20	50
3	Londyn	Ateny	TCP	10	10

Tabela 1: Zestawiane połączenia

Po uruchomieniu skryptu na tych danych, zestawione zostały następujące drogi



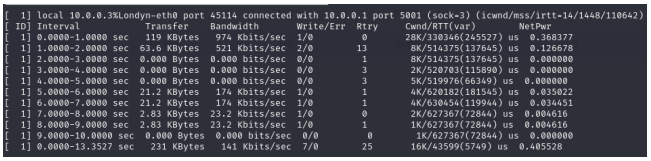
Rys. 19: Drogi zestawionych połączeń

Oczekiwana minimalna przepustowość strumienia TCP z Londynu do Aten mieści się w możliwościach sieci, niestety nie jest możliwe spełnienie nierealnie niskiego opóźnienia, o czym poinformował nas skrypt. Dodatkowo, należy zauważyć, że dzięki niskim wymaganiom dot. minimalnej przepustowości, ten strumień został poprowadzony razem z innymi strumieniami w tym teście. 1 i 2 strumień miały za duże wymagania, aby móc je w ten sposób poprowadzić, dlatego strumień nr 2 "skręcił" do Zagrzebu, dokładając sobie czasu, ale zachowując przepustowość.

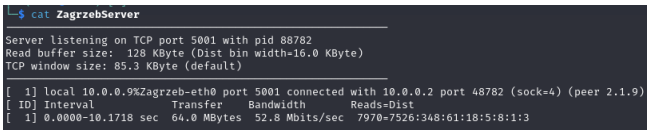
```
┌──(kali㉿kali)-[~/SCHT/lab2]
└─$ /bin/python3 /home/kali/SCHT/lab2/cli.py -f connections.txt
Attempting to configure flow rules for the path:
Warszawa -> s10 -> s9 -> s6 -> s7 -> Ateny
[INFO] Successfully added 8 flows to switches.

Attempting to configure flow rules for the path:
Berlin -> s8 -> s9 -> s4 -> s6 -> Zagrzeb
[INFO] Successfully added 8 flows to switches.

Not able to provide connection with given parameters from Londyn to Ateny.
Setting best possible connection instead:
Total delay: 17, minimum throughput: 10
Attempting to configure flow rules for the path:
Londyn -> s1 -> s2 -> s4 -> s6 -> s7 -> Ateny
[INFO] Successfully added 10 flows to switches.
```

Rys. 20: Wynik skryptu wyliczającego ścieżki

Zaczynając od najlepiej zestawionego połączenia, Berlin nadawał do Zagrzebu z zadowalającą prędkością.

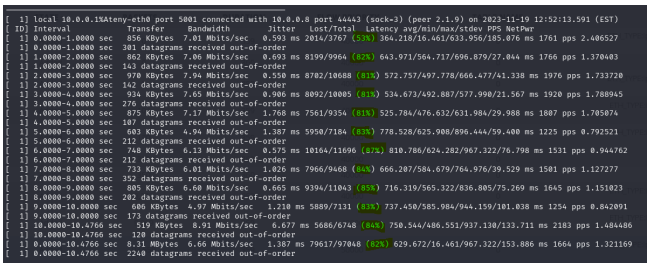


Rys. 23: Ledwo docierające pakiety UDP z Londynu do Aten



Rys. 21: Berlin Zagrzeb TCP

Natomiast, problemy wystąpiły przede wszystkim ze strumieniem danych z Warszawy. Nawet, gdy zmieniliśmy destynacje połączenia z Londynu, dając Warszawie pełen dostęp do wszystkich łącz, Wartości utraty pakietów nie zmieniły się. Być może jest to spowodowane przeciążonym switchem na wysokości Zagrzebu, który musi kierować ruchem 3 strumieni pakietów.



Rys. 22: Ledwo docierające pakiety UDP z Warszawy do Aten

Podobnie z połączeniem z Londynu, przesyłało one znikome ilości danych. W tym przypadku jednak jest to bardziej uzasadnione, gdyż pakiety na swojej drodze napotkały aż 2 inne strumienie danych.

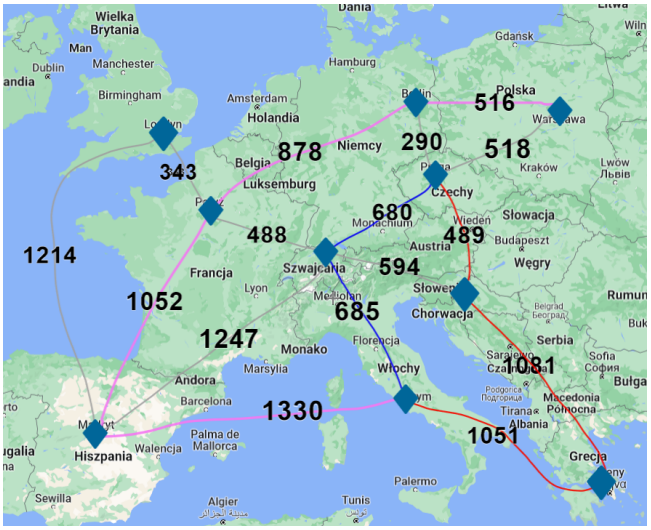
2.2.3. Scenariusz zaawansowany 2

W tym scenariuszu symulujemy następujące zlecenia strumieni:

	źródło	cel	protokół	min. przepustowość	max. ping
1	Praga	Rzym	UDP	60	25
2	Praga	Rzym	TCP	30	25
3	Warszawa	Rzym	TCP	60	100

Tabela 2: tabela zestawianych połączeń

Po wykonaniu skryptu, wyznaczona trasa wygląda następująco:



Rys. 24: Mapa tras połączeń

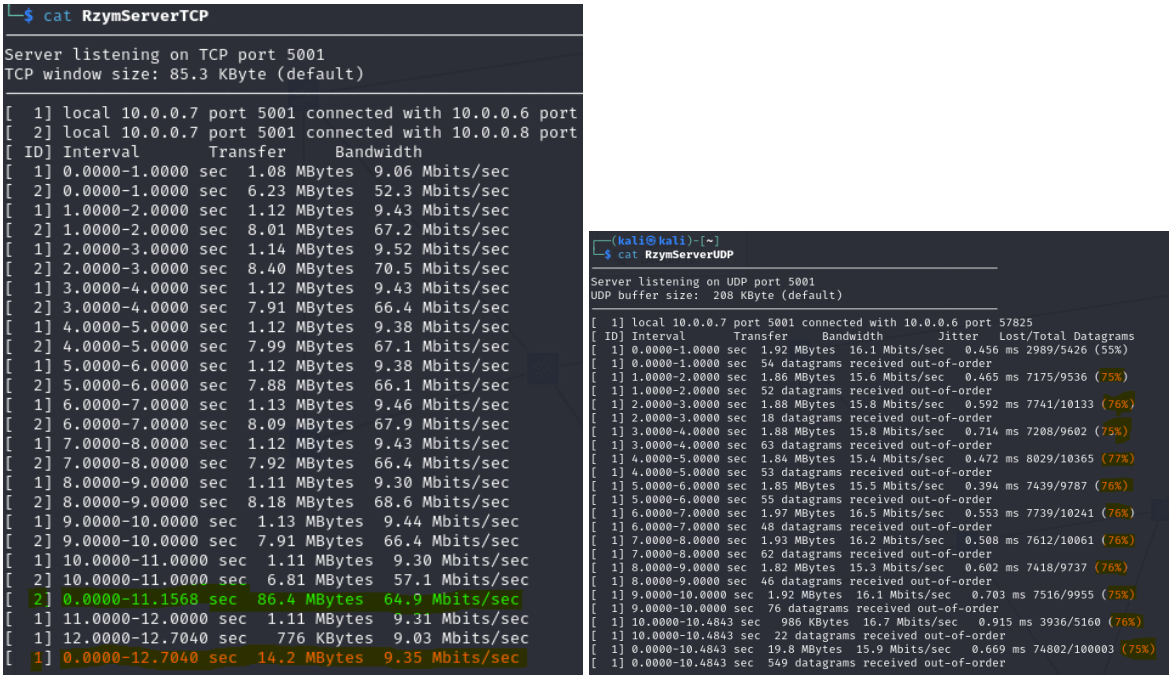
Skrypt poprowadził dwa pierwsze strumienie dość oczywistymi drogami, jednak trzecie połączenie ze względu na wysokie wymagania co do przepustowości i niskie dot. opóźnienia, został poprowadzony najdłuższą, lecz całkowicie pustą trasą. Istotne dla dalszych rozważań jest zauważenie fakty, iż są to 3 niezależne (co do łączy międzymiastowych) ścieżki.

```
└─$ /bin/python3 /home/kali/SCHT/lab2/cli.py -f connections.txt
Attempting to configure flow rules for the path:
Praga -> s9 -> s4 -> s5 -> Rzym
[INFO] Successfully added 6 flows to switches.

Attempting to configure flow rules for the path:
Praga -> s9 -> s6 -> s7 -> s5 -> Rzym
[INFO] Successfully added 8 flows to switches.

Attempting to configure flow rules for the path:
Warszawa -> s10 -> s8 -> s2 -> s3 -> s5 -> Rzym
[INFO] Successfully added 10 flows to switches.
```

Rys. 25: Wynik wywołania skryptu

Wyniki są również dosyć zaskakujące. Patrząc na połączenia TCP, mimo, że połączenie z Warszawy miało znacznie większe opóźnienie, to znacząco przewyższa przepustowość połączenia TCP z Pragi. Jest to zaskakujące zwłaszcza dla tego, że obydwa połączenia miały łącza "na własność".

Gdy spojrzymy na wyniki połączenia UDP, okazuje się, że utrata pakietów jest równie wysoka, co w poprzednim teście. Cechą wspólną jest używanie switcha przez wiele połączeń jednocześnie. Połączenie z Warszawy nie współdzieli żadnych switchy, a połączenia z Pragi używają jednego, który rozdziela je na dwie strony. Według naszej tezy właśnie to rozdzielanie pakietów na switchach jest wyjątkowo obciążające i powoduje opóźnienia oraz utraty pakietów.



Rys. 26: Wyniki przepustowości (dwóch) strumieni TCP po lewej i strumienia UDP po lewej

3. Podsumowanie

Podczas tego laboratorium mieliśmy za zadanie skonfigurować sieć wykorzystując do tego kontroler ONOS, dzięki któremu mogliśmy ustalać reguły w tablicach przepływów. Ponadto do komunikacji z ONOSem użyliśmy styku REST, wysyłając odpowiednie pliki JSON przy użyciu protokołu http. Najciekawszą i zarazem najtrudniejszą częścią tego zadania, która jest również esencją warstwy transportowej, było napisanie własnego systemu sterującego ruchem w języku Python. Przed przystąpieniem do tego zadania rozszerzyliśmy sieć na której pracowaliśmy oraz wprowadziliśmy do niej cykle, co zdecydowanie zwiększyło liczbę możliwych tras. Z początku wydawało się nam, że dzięki tej operacji nasz system będzie w stanie zapewnić lepsze warunki strumieniom i połączeniom generowanym przez iperf dzięki czemu sieć będzie wydajniejsza i bardziej efektywna. Okazało się jednak, że zwiększenie ilości ścieżek nie idące w parze z umiejętnością optymalizacji i prawidłowego zarządzania może skutkować totalnym załamaniem systemu. Dlatego też staraliśmy się stworzyć algorytm, który uwzględni zarówno potrzeby klienta jak i procesy zachodzące w sieci podczas transmisji danych. Oczywiście nie jesteśmy w stanie przewidzieć wszystkiego dlatego też idąc zgodnie z regułą 80/20 widzimy miejsca do poprawy w naszym systemie.