

SCHT - Lab2

Stanisław Kwiatkowski, Bartosz Ziemba

Spis treści

1. Konfiguracja ręczna	1
1.1. Podstawowa konfiguracja	1
1.2. Testowanie połączeń	2
1.2.1. UDP 50% + UDP 50%	2
1.2.2. UDP 50% + TCP	3
1.2.3. UDP 25% + UDP 25% + UDP 25%	3
1.3. Testy na pełnej sieci	3
1.3.1. UDP 25% + UDP 25% + UDP 25%	4
1.3.2. UDP 50% + UDP 50%	4
1.3.3. UDP 50% + TCP	4
2. Konfiguracja automatyczna - skrypt w Pythonie	5
2.1. Analiza programu	5
2.1.1. Wczytywanie informacji o budowie sieci do systemu	5
2.1.2. Wczytywanie informacji o żądanych parametrach strumieni	6
2.1.3. Budowanie najlepszej ścieżki na podstawie dobranych współczynników i funkcji uwzględnionych w wagach krawędzi grafu	6
2.1.4. Konfiguracja flows w ONOSie korzystając ze styku REST	7
2.2. Testowanie połączeń przy konfiguracji automatycznej	7
2.2.1. Warszawa -> Madryt, Berlin -> Madryt	7
2.2.2. Scenariusz zaawansowany 1	8
2.2.3. Scenariusz zaawansowany 2	10
3. Podsumowanie	12

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 switchy. 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.

```
mininet@mininet-vm:~/flowConfig/jsonPocket$ source configMadryt-Warszawa.sh
```

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

```
mininet> Warszawa ping Madryt -c 4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=36.2 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=34.9 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=35.3 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=36.0 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
```

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 wcześniejszymi wynikami.

1.2.1. UDP 50% + UDP 50%

```
Server listening on UDP port 5001 with pid 6535
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 55785 (sock=3) (peer 2.1.5) on 2023-11-19 17:44:27 (CET)
[ 2] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.8 port 37960 (sock=4) (peer 2.1.5) on 2023-11-19 17:44:27 (CET)
[ ID] Interval      Transfer      Bandwidth      Jitter    Lost/Total  Latency avg/min/max/stdev PPS NetPwr
[ 1] 0.0000-10.0089 sec 40.6 MBytes  34.0 Mbits/sec  0.048 ms 14804/100003 (15%) 61.106/9.047/115.735/9.499 ms 8512 pps 70
[ 1] 0.0000-10.0089 sec 1017 datagrams received out-of-order
[ 2] 0.0000-9.9542 sec 39.8 MBytes  33.5 Mbits/sec  0.093 ms 16638/100003 (17%) 69.772/18.995/130.891/9.061 ms 8375 pps 60
[ 2] 0.0000-9.9542 sec 1127 datagrams received out-of-order
```

Rys. 5: Wynik testu

1.2.2. UDP 50% + TCP

```
-----
Server listening on TCP port 5001 with pid 14026
Read buffer size: 128 KByte (Dist bin width=16.0 KByte)
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.3 port 57968 (sock=4) (peer 2.1.5) on 2023-11-19 18:15:26 (CET)
[ ID] Interval      Transfer    Bandwidth   Reads=Dist
[ 1] 0.0000-70.6636 sec 200 MBytes  23.7 Mbits/sec 47874=47734:40:13:10:12:4:1:60
```

Rys. 6: Wynik testu dla serwera TCP

```
-----
Server listening on TCP port 5001 with pid 14026
Read buffer size: 128 KByte (Dist bin width=16.0 KByte)
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.3 port 57968 (sock=4) (peer 2.1.5) on 2023-11-19 18:15:26 (CET)
[ ID] Interval      Transfer    Bandwidth   Reads=Dist
[ 1] 0.0000-70.6636 sec 200 MBytes  23.7 Mbits/sec 47874=47734:40:13:10:12:4:1:60
```

Rys. 7: Wynik testu dla serwera UDP

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

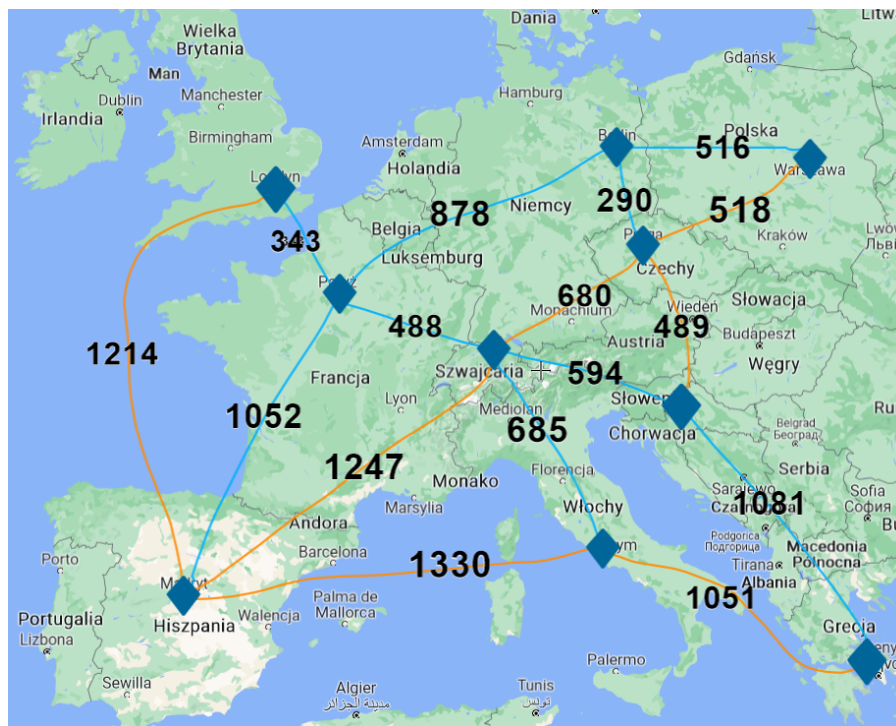
```
Server listening on UDP port 5001 with pid 4583
Read buffer size: 1.44 KByte (Dist bin width= 183 Byte)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.3 port 40748 (sock=3) (peer 2.1.5) on 2023-11-19 17:36:26 (CET)
[ 2] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.8 port 40256 (sock=5) (peer 2.1.5) on 2023-11-19 17:36:26 (CET)
[ 3] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.1 port 34000 (sock=4) (peer 2.1.5) on 2023-11-19 17:36:26 (CET)
[ ID] Interval      Transfer    Bandwidth   Jitter    Lost/Total Latency avg/min/max/stdev PPS NetPwr
[ 1] 0.0000-10.0049 sec 19.3 MBytes 16.1 Mbits/sec 0.376 ms 19222/99999 (19%) 44.411/9.018/249.907/22.946 ms 8074 pps 45
[ 1] 0.0000-10.0049 sec 6138 datagrams received out-of-order
[ 3] 0.0000-9.8980 sec 18.3 MBytes 15.5 Mbits/sec 0.387 ms 23201/99973 (23%) 60.087/22.218/270.329/25.000 ms 7756 pps 32
[ 3] 0.0000-9.8980 sec 6518 datagrams received out-of-order
[ 2] 0.0000-9.9635 sec 18.3 MBytes 15.4 Mbits/sec 0.092 ms 23158/100004 (23%) 53.830/17.055/264.107/24.510 ms 7713 pps 36
[ 2] 0.0000-9.9635 sec 6099 datagrams received out-of-order
```

Rys. 8: Wynik testu

Jak widać testy są wyniki przeprowadzonych testów są ładząco 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/stddev 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 10.9 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/stddev 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 znaczny wzrost zarówno wydajności jak i efektywności co objawiło się większą szybkością przesyłu i zmniejszoną utratą pakietów

```

Server listening on TCP port 5001 with pid 20758
Read buffer size: 128 KByte (Dist bin width=16.0 KByte)
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.4%Madryt-eth0 port 5001 connected with 10.0.0.3 port 46746 (sock=4) (peer 2.1.5) on 2023-11-19 18:58:45 (CET)
[ ID] Interval      Transfer    Bandwidth    Reads=Dist
[ 1] 0.0000-35.2488 sec  200 MBytes  47.6 Mbits/sec  25343=25044:170:13:4:6:2:4:100

```

Rys. 12: Wynik testu serwera TCP

```

Server listening on UDP port 5001 with pid 20760
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 40559 (sock=3) (peer 2.1.5) on 2023-11-19 18:58:45 (CET)
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total  Latency avg/min/max/stddev PPS NetPwr
[ 1] 0.0000-19.9848 sec  92.6 MBytes  38.9 Mbits/sec  0.250 ms  5762/199998 (2.9%) 32.354/17.027/264.819/30.678 ms 9719 pps 150
[ 1] 0.0000-19.9848 sec  4206 datagrams received out-of-order

```

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:

```

1 {
2     "devices": [
3     {
4         "name": "Londyn",
5         "ip": "10.0.0.3",
6         "links": [
7             {
8                 "port": 1,
9                 "linkId": 1
10            },
11            ....
12        ]
13    },
14    ....
15 ],
16 "links": [
17     {
18         "id": 1,
19         "node1": "Londyn",
20         "node2": "s1",
21         "delay": 0,
22         "bw": 1000
23     },
24     ....

```

```
25 ]
26 }
```

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 = $0Mb$

```
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 obciążenia 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 tym etapie zorientowaliśmy 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).

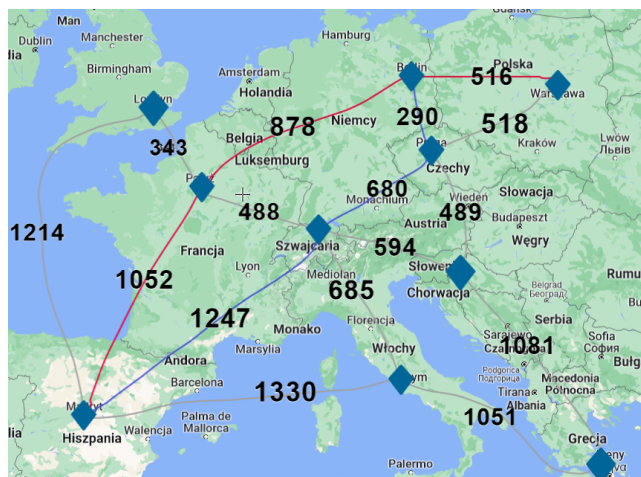
Dodatkowo, skrypt umożliwia dodawanie flag, na chwilę obecną są to *-f/-file* do podawania pliku tekstowego z parametrami zleczanych strumieni oraz *-d/-debug* do wyświetlania dodatkowych informacji dotyczących błędów.

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/stddev 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

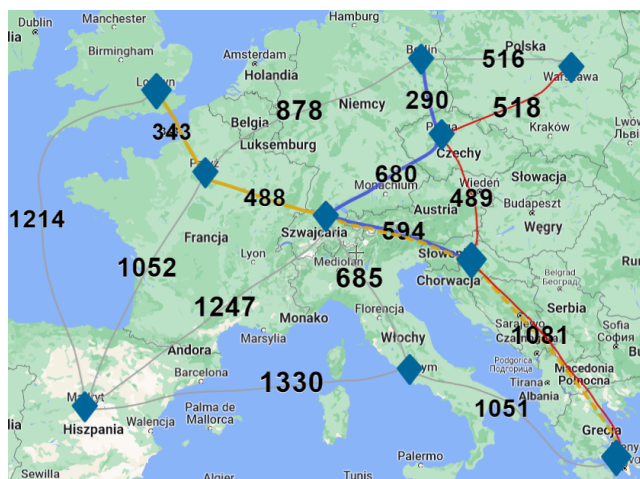
2.2.2. Scenariusz zaawansowany 1

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.

```

[ 1] local 10.0.0.3%London-eth0 port 45114 connected with 10.0.0.1 port 5001 (sock=3) (icwnd/mss/rtt-14/1448/110642)
[ ID] Interval      Transfer      Bandwidth      Write/Err      Rtry      Cwnd/RTT(var)      NetPer
[ 1] 0.0000-1.0000 sec  119 Kbytes    974 Kbits/sec  1/0            0          28K/338346(245527) us  0.368377
[ 1] 1.0000-2.0000 sec  63.6 Kbytes   521 Kbits/sec  2/0            11         8K/514375(137645) us  0.156578
[ 1] 2.0000-3.0000 sec  0.000 Bytes   0.000 bits/sec  0/0            1          8K/514375(137645) us  0.000000
[ 1] 3.0000-4.0000 sec  0.000 Bytes   0.000 bits/sec  0/0            3          2K/528703(115990) us  0.000000
[ 1] 4.0000-5.0000 sec  0.000 Bytes   0.000 bits/sec  0/0            3          5K/519976(66345) us  0.000000
[ 1] 5.0000-6.0000 sec  21.2 Kbytes    174 Kbits/sec  1/0            1          4K/620182(181545) us  0.035022
[ 1] 6.0000-7.0000 sec  21.2 Kbytes    174 Kbits/sec  1/0            1          4K/638454(119944) us  0.034451
[ 1] 7.0000-8.0000 sec  2.83 Kbytes    23.2 Kbits/sec  1/0            0          2K/627367(72844) us  0.004616
[ 1] 8.0000-9.0000 sec  2.83 Kbytes    23.2 Kbits/sec  1/0            1          1K/627367(72844) us  0.004616
[ 1] 9.0000-10.0000 sec 0.000 Bytes    0.000 bits/sec  0/0            0          1K/627367(72844) us  0.000000
[ 1] 0.0000-10.3527 sec 231 Kbytes     141 Kbits/sec  7/0            25         16K/43599(5749) us  0.405528

```

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

```

+ cat ZagrzebServer
Server listening on TCP port 5001 with pid 88782
Read buffer size: 128 KByte (Dist bin width:16.0 KByte)
TCP window size: 85.3 KByte (default)
[ 1] local 10.0.0.9%Zagreb-eth0 port 5001 connected with 10.0.0.2 port 48782 (sock=4) (peer 2.1.9)
[ ID] Interval      Transfer      Bandwidth      Reads=Dist
[ 1] 0.0000-10.1718 sec  64.0 Mbytes    52.8 Mbits/sec  7970-7526:348:61:18:5:8:1:3

```

Rys. 21: Berlin Zagrzeb TCP

Natomiast, problemy wystąpiły przede wszystkim ze strumieniem danych z Warszawy. Nawet, gdy zmieniliśmy destynację 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.

```

[ 1] local 10.0.0.1%Ateny-eth0 port 5001 connected with 10.0.0.0 port 44443 (sock=3) (peer 2.1.9) on 2023-11-19 12:52:13.591 (EST)
[ ID] Interval      Transfer      Bandwidth      Jitter      Lost/Total      Latency avg/min/max/stddev PPS NetPer
[ 1] 0.0000-1.0000 sec  856 Kbytes    7.81 Mbits/sec  0.593 ms 2014/3707 (0.0%) 384-218/76-483/633-556/185-876 ms 1761 pps 2.406527
[ 1] 0.0000-1.0000 sec  381 datagrams received out-of-order
[ 1] 1.0000-2.0000 sec  163 datagrams received out-of-order
[ 1] 2.0000-3.0000 sec  970 Kbytes    7.94 Mbits/sec  0.550 ms 8702/10088 (0.1%) 572.757/497.778/666.477/41.338 ms 1976 pps 1.733728
[ 1] 3.0000-4.0000 sec  934 Kbytes    7.65 Mbits/sec  0.906 ms 8092/10005 (0.1%) 534.673/492.887/577.998/21.567 ms 1920 pps 1.788945
[ 1] 4.0000-5.0000 sec  875 Kbytes    7.17 Mbits/sec  1.768 ms 7561/9354 (0.1%) 525.784/476.632/631.984/29.988 ms 1887 pps 1.705874
[ 1] 5.0000-6.0000 sec  609 Kbytes    4.94 Mbits/sec  1.387 ms 5958/7184 (0.3%) 778.528/625.988/896.444/59.400 ms 1225 pps 0.792521
[ 1] 6.0000-7.0000 sec  748 Kbytes    6.13 Mbits/sec  0.578 ms 10164/11696 (0.7%) 818.786/624.282/967.322/76.798 ms 1531 pps 0.944762
[ 1] 7.0000-8.0000 sec  352 datagrams received out-of-order
[ 1] 8.0000-9.0000 sec  885 Kbytes    6.80 Mbits/sec  0.605 ms 9394/11043 (0.5%) 716.319/565.322/836.805/75.269 ms 1645 pps 1.151023
[ 1] 9.0000-10.0000 sec  292 datagrams received out-of-order
[ 1] 10.0000-10.4766 sec  519 Kbytes    8.91 Mbits/sec  6.677 ms 5686/6748 (0.6%) 750.544/486.551/937.138/133.711 ms 2183 pps 1.484486
[ 1] 0.0000-10.4766 sec  120 datagrams received out-of-order
[ 1] 0.0000-10.4766 sec  811 Kbytes    6.40 Mbits/sec  1.187 ms 79617/97048 (0.7%) 629.672/16.461/967.322/153.886 ms 1664 pps 1.321169
[ 1] 0.0000-10.4766 sec  2248 datagrams received out-of-order

```

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.

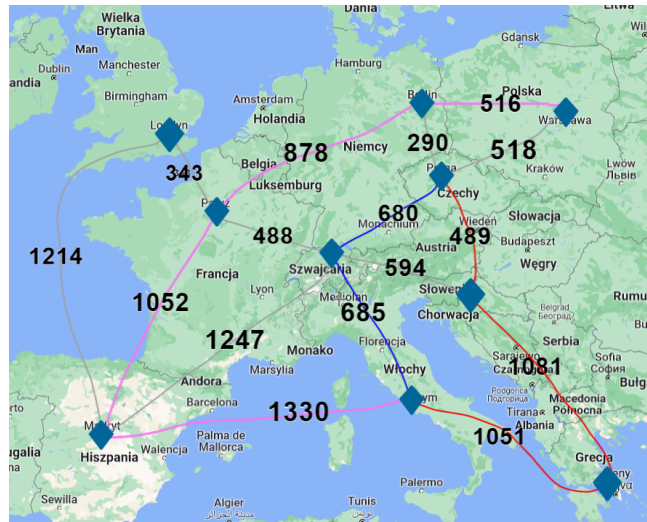
2.2.3. Scenariusz zaawansowany 2

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 faktu, 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 łączyć "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.

```

L-$ cat RzymServerTCP
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)

[ 1] local 10.0.0.7 port 5001 connected with 10.0.0.6 port
[ 2] local 10.0.0.7 port 5001 connected with 10.0.0.8 port
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-1.0000 sec  1.08 MBytes  9.06 Mbits/sec
[ 2] 0.0000-1.0000 sec  6.23 MBytes  52.3 Mbits/sec
[ 1] 1.0000-2.0000 sec  1.12 MBytes  9.43 Mbits/sec
[ 2] 1.0000-2.0000 sec  8.01 MBytes  67.2 Mbits/sec
[ 1] 2.0000-3.0000 sec  1.14 MBytes  9.52 Mbits/sec
[ 2] 2.0000-3.0000 sec  8.40 MBytes  70.5 Mbits/sec
[ 1] 3.0000-4.0000 sec  1.12 MBytes  9.43 Mbits/sec
[ 2] 3.0000-4.0000 sec  7.91 MBytes  66.4 Mbits/sec
[ 1] 4.0000-5.0000 sec  1.12 MBytes  9.38 Mbits/sec
[ 2] 4.0000-5.0000 sec  7.99 MBytes  67.1 Mbits/sec
[ 1] 5.0000-6.0000 sec  1.12 MBytes  9.38 Mbits/sec
[ 2] 5.0000-6.0000 sec  7.88 MBytes  66.1 Mbits/sec
[ 1] 6.0000-7.0000 sec  1.13 MBytes  9.46 Mbits/sec
[ 2] 6.0000-7.0000 sec  8.09 MBytes  67.9 Mbits/sec
[ 1] 7.0000-8.0000 sec  1.12 MBytes  9.43 Mbits/sec
[ 2] 7.0000-8.0000 sec  7.92 MBytes  66.4 Mbits/sec
[ 1] 8.0000-9.0000 sec  1.11 MBytes  9.30 Mbits/sec
[ 2] 8.0000-9.0000 sec  8.18 MBytes  68.6 Mbits/sec
[ 1] 9.0000-10.0000 sec 1.13 MBytes  9.44 Mbits/sec
[ 2] 9.0000-10.0000 sec  7.91 MBytes  66.4 Mbits/sec
[ 1] 10.0000-11.0000 sec 1.11 MBytes  9.30 Mbits/sec
[ 2] 10.0000-11.0000 sec  6.81 MBytes  57.1 Mbits/sec
[ 2] 0.0000-11.1568 sec 86.4 MBytes 64.9 Mbits/sec
[ 1] 11.0000-12.0000 sec 1.11 MBytes  9.31 Mbits/sec
[ 1] 12.0000-12.7040 sec 776 KBytes  9.03 Mbits/sec
[ 1] 0.0000-12.7040 sec 14.2 MBytes  9.35 Mbits/sec

(kali@kali)~$ cat RzymServerUDP
Server listening on UDP port 5001
UDP buffer size: 208 KByte (default)

[ 1] local 10.0.0.7 port 5001 connected with 10.0.0.6 port 57825
[ ID] Interval      Transfer      Bandwidth      Jitter      Lost/Total Datagrams
[ 1] 0.0000-1.0000 sec  1.92 MBytes  16.1 Mbits/sec  0.456 ms 2989/5426 (55%)
[ 1] 0.0000-1.0000 sec  54 datagrams received out-of-order
[ 1] 1.0000-2.0000 sec  1.86 MBytes  15.6 Mbits/sec  0.465 ms 7175/9536 (75%)
[ 1] 1.0000-2.0000 sec  52 datagrams received out-of-order
[ 1] 2.0000-3.0000 sec  1.88 MBytes  15.8 Mbits/sec  0.592 ms 7741/10133 (76%)
[ 1] 2.0000-3.0000 sec  18 datagrams received out-of-order
[ 1] 3.0000-4.0000 sec  1.88 MBytes  15.8 Mbits/sec  0.714 ms 7208/9602 (75%)
[ 1] 3.0000-4.0000 sec  63 datagrams received out-of-order
[ 1] 4.0000-5.0000 sec  1.84 MBytes  15.4 Mbits/sec  0.472 ms 8029/10365 (77%)
[ 1] 4.0000-5.0000 sec  53 datagrams received out-of-order
[ 1] 5.0000-6.0000 sec  1.85 MBytes  15.5 Mbits/sec  0.594 ms 7439/9787 (76%)
[ 1] 5.0000-6.0000 sec  55 datagrams received out-of-order
[ 1] 6.0000-7.0000 sec  1.97 MBytes  16.5 Mbits/sec  0.553 ms 7739/10241 (76%)
[ 1] 6.0000-7.0000 sec  48 datagrams received out-of-order
[ 1] 7.0000-8.0000 sec  1.93 MBytes  16.2 Mbits/sec  0.508 ms 7612/10061 (76%)
[ 1] 7.0000-8.0000 sec  62 datagrams received out-of-order
[ 1] 8.0000-9.0000 sec  1.82 MBytes  15.3 Mbits/sec  0.602 ms 7418/9737 (76%)
[ 1] 8.0000-9.0000 sec  46 datagrams received out-of-order
[ 1] 9.0000-10.0000 sec  1.92 MBytes  16.1 Mbits/sec  0.703 ms 7516/9955 (75%)
[ 1] 9.0000-10.0000 sec  76 datagrams received out-of-order
[ 1] 10.0000-10.4843 sec  986 KBytes  16.7 Mbits/sec  0.915 ms 3936/5160 (76%)
[ 1] 10.0000-10.4843 sec  22 datagrams received out-of-order
[ 1] 0.0000-10.4843 sec  19.8 MBytes  15.9 Mbits/sec  0.669 ms 74802/100003 (75%)
[ 1] 0.0000-10.4843 sec  549 datagrams received out-of-order

```

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 ONOS-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.