



Universidade do Minho
Escola de Engenharia

MESTRADO EM ENGENHARIA DE TELECOMUNICAÇÕES E INFORMÁTICA
SERVIÇOS DE REDE & APLICAÇÕES MULTIMÉDIA

DOCENTE: BRUNO ALEXANDRE FERNANDES DIAS

TRABALHO PRÁTICO Nº2 – REDE APLICACIONAL DE STREAMING LIVE

Rui Filipe Ribeiro Freitas - pg47639@alunos.uminho.pt
Tiago João Pereira Ferreira - pg47692@alunos.uminho.pt

28 de junho de 2022

Índice

1	Introdução	2
2	Desenvolvimento	3
2.1	Controlador do Serviço (SC)	3
2.2	Servidor de streaming	5
2.3	Elementos de relay	6
2.4	Elementos de edge	8
2.5	Elementos clientes	10
3	Testes	13
4	Conclusão	17

1 Introdução

Os serviços de streaming live têm um papel essencial na atualidade de praticamente todas as pessoas. Isto visto que este tipo de serviços disponibiliza aos seus utilizadores a possibilidade de estar em constante atualização das notícias ao seu redor, bem como conteúdos de lazer e pessoais.

Neste trabalho é apresentada a implementação de uma rede aplicacional com o objetivo de disponibilizar aos seus utilizadores diferentes *streams* de conteúdo em tempo real. A realização deste teve como objetivo a construção de uma topologia física para servir como ambiente para realizar os testes assim como realizar todo o código necessário para implementar uma rede overlay.

De modo a sermos capazes de cumprir com os objetivos deste trabalho foi necessário pôr em prática conhecimentos adquiridos ao longo do semestre na unidade curricular assim como realizar alguma pesquisa de eventuais dificuldades encontradas.

2 Desenvolvimento

Nesta secção do relatório é apresentada a parte mais prática do projeto onde se encontra a discussão das estratégias escolhidas, as opções tomadas e os mecanismos adotados de cada elemento da rede.

2.1 Controlador do Serviço (SC)

O Controlador do Serviço (SC) é o elemento responsável pelo funcionamento da rede aplicacional. Este tem como funções iniciar o sistema, controlar a topologia (indicar aos elementos relay, edge e cliente quais os seus vizinhos) e catalogar as *streams* disponíveis. O SC é também o único elemento que possui um endereço público.

Quando inicia o sistema, a primeira operação que o SC realiza é ler do ficheiro de configuração a topologia e colocar todos os elementos no estado "OFF" (figura 1). Depois de carregada a topologia este fica à “escuta” de pedidos por partes dos diferentes elementos. Sempre que alguém se conecta, este recebe um pacote de pedido de conexão. Com isto, o SC atribui o estado “ON” ao elemento e envia-lhe os respetivos vizinhos.



```
1 def loadTopology():
2     """
3     It reads the config file and stores the topology in a dictionary
4     """
5     with open(configFile) as f:
6         lines = f.readlines()
7         for line in lines:
8             arguments = line.split(" ")
9             if(arguments[0] == "T"):
10                topology[arguments[3]] = "OFF"
```

Figura 1: Função *loadTopology()*.

Para controlar o estado de cada elemento na topologia é usado o pacote *beacon* que é enviado entre todos os elementos da rede com uma certa frequência. Através deste pacote é possível determinar quais os elementos que se encontram ligados.

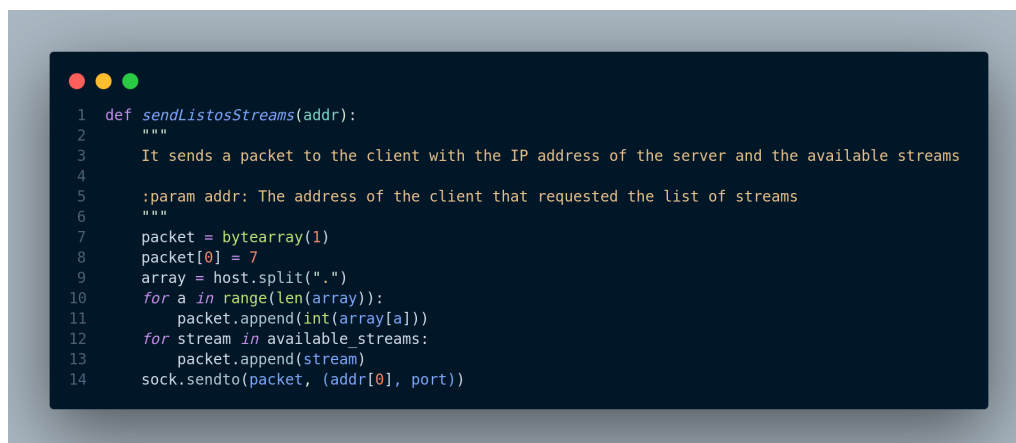
Na figura seguinte encontra-se o código da função que controla o estado de cada elemento.



```
1 def isAlive(addr):
2     """
3     It checks if the node is alive by checking if the time difference between the current time and the
4     last time the node sent a beacon is greater than 2 seconds. If it is, it means that the node is
5     offline and it is removed from the topology
6
7     @param addr The address of the client that sent the message.
8     """
9     while True:
10         sleep(1)
11         if(time.time() - times_beacon[addr[0]] > 2):
12             topology[addr[0]] = "OFF"
13             del times_beacon[addr[0]]
14             for ip_topology in topology.keys():
15                 if(topology[ip_topology] == "ON"):
16                     sendNeighbours(ip_topology)
17             sys.exit(1)
```

Figura 2: Função *isAlive()*.

Sempre que um cliente pede as *streams* disponíveis ao SC, este responde com um pacote com as streams que se encontram disponíveis no servidor (figura 3).



```
1 def sendListofStreams(addr):
2     """
3     It sends a packet to the client with the IP address of the server and the available streams
4
5     :param addr: The address of the client that requested the list of streams
6     """
7     packet = bytearray(1)
8     packet[0] = 7
9     array = host.split(".")
10    for a in range(len(array)):
11        packet.append(int(array[a]))
12    for stream in available_streams:
13        packet.append(stream)
14    sock.sendto(packet, (addr[0], port))
```

Figura 3: Função *sendListofStreams()*.

2.2 Servidor de streaming

Relativamente ao servidor de streaming, que na nossa implementação é único, este tem como principal objetivo manter as transmissões das *streams* e fornecer ao controlador de serviços a lista de *streams* a ser partilhada.

A implementação deste elemento tem 2 funções principais, uma com o objetivo de transmitir as *streams* para os relays e outra que tem como objetivo enviar a lista de *streams* para o SC. Para além destas existem 3 que são realizadas por todos os elementos da rede, que são o envio de *beacons*, o pedido de vizinhos e o pedido de conexão à rede. Como as 2 primeiras são características somente do servidor estas vão ser apresentadas de seguida.

A primeira função é a de enviar a lista de *streams* para o SC em que esta função cria um pacote com o tipo 2, adiciona o IP do servidor, o número de *streams* disponíveis e o nome dessas *streams*. Após isso é enviado por uma socket UDP para o IP do SC que é público.

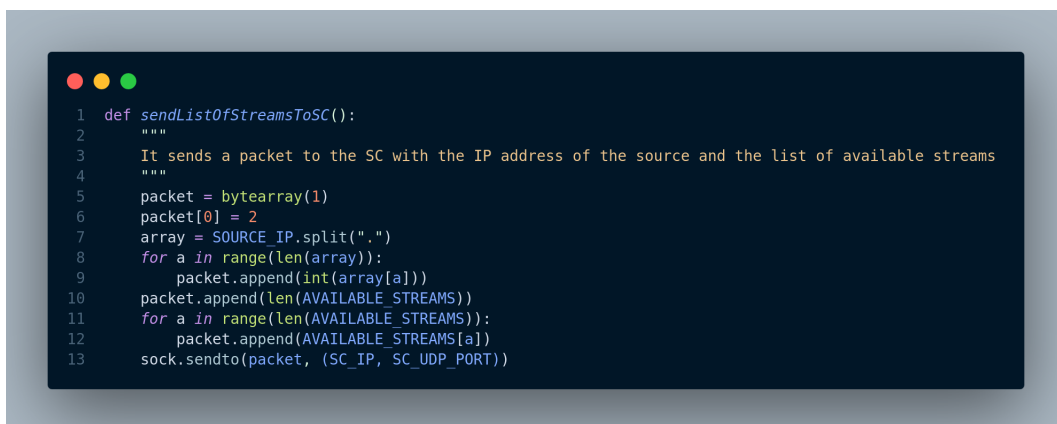


Figura 4: Função *sendListOfStreamsToSC()*.

A função mais relevante no servidor de streaming é então a de transmitir as *streams*. Esta função, apresentada na próxima figura, recebe o pacote proveniente do relay, o seu endereço IP e a frequência de transmissão. De acordo com o pacote recebido, este acede à *stream* desejada e cria um pacote com o tipo 13, o endereço IP do servidor, o ID da *stream*, o ID do pedido, o número da frame a transmitir e por fim o conteúdo, que neste caso é enviado 1 byte por pacote. Após criado o pacote é

enviado através da socket UDP com a frequência fornecida.



```
1 def sendStream(packet_received, addr, frequency):
2     """
3     It reads a byte from the file, creates a packet with the byte, and sends the packet to the address
4     and port specified
5
6     :param packet_received: the packet received from the client
7     :param addr: the IP address of the sender
8     :param frequency: the time between each packet sent
9     """
10    stream_id = packet_received[5]
11    requestID = packet_received[6]
12    stream = "streams/stream" + str(stream_id) + ".txt"
13    file = open(stream, "rb")
14    frame_number = 0
15    #print(stream_id)
16    while True:
17        byte = file.read(1)
18        packet = bytearray(1)
19        packet[0] = 13
20        array = SOURCE_IP.split(".")
21        for a in range(len(array)):
22            packet.append(int(array[a]))
23        packet.append(stream_id)
24        packet.append(requestID)
25        packet.append(frame_number)
26        frame_number += 1
27        if(frame_number >= 256):
28            frame_number = 0
29        packet.append(ord(byte))
30        sock.sendto(packet, (addr[0], UDP_PORT))
31        sleep(frequency)
```


Figura 5: Função *sendStream()*.

2.3 Elementos de relay

Relativamente aos elementos de relay, estes são responsáveis por interligar o servidor de *streaming* aos elementos edge. Primeiramente realizam a conexão ao SC, recebem os vizinhos e adicionam estes ao seu dicionário. Após isso, aguardam por um pedido de transmissão por parte do edge e, caso um servidor seja seu vizinho envia para este o pedido de transmissão, caso contrário, envia para um relay vizinho esse pedido de modo a que caso este seja vizinho de um servidor, comunique com ele. Após receber o pacote da *stream* desejada, este reencaminha o pacote para o edge que realizou o pedido de transmissão.

A implementação deste elemento tem 3 funções principais em que 2 delas são bastantes similares. As funções *askServerForStream()* e *askRelayNeighbourForStream()*

têm o mesmo objetivo que é de pedir ao servidor a transmissão da *stream*, no entanto, caso o servidor não seja vizinho, este tem de enviar o pedido para um relay vizinho. O que altera nestas 2 funções é o facto de quando o relay faz o pedido ao relay vizinho é necessário fornecer qual é o ID do pedido de modo que depois seja possível enviar o pedido para o edge correto. De seguida é apresentada a função *askServerForStream()*.



```
1 def askServerForStream(id_stream):
2     """
3     It sends a request to the server for a stream
4
5     :param id_stream: the id of the stream you want to ask for
6     """
7     global requestID
8     packet = bytearray(1)
9     packet[0] = 12
10    array = SOURCE_IP.split(".")
11    for a in range(len(array)):
12        packet.append(int(array[a]))
13    packet.append(id_stream)
14    for x in neighbours_dict:
15        if x.find("S") != -1:
16            sock.sendto(packet, (neighbours_dict[x], UDP_PORT))
```

Figura 6: Função *askServerForStream()*.

Com a *stream* recebida é necessário proceder ao seu reencaminhamento para o edge que realizou o pedido. Esta transmissão é realizada na função *sendStreamToEdge()* apresentada na próxima figura. Esta função recebe o pacote proveniente do servidor e cria um novo pacote do tipo 14 com o IP do relay, o ID da stream, o número da frame, o conteúdo da *stream* e por fim o ID do pedido. Após adicionados estes campos é realizado o envio do pacote através de uma socket UDP para o edge.


```

1  def sendStreamToEdge(packet_received):
2      """
3      It sends the stream to the edge that requested it
4
5      :param packet_received: the packet received from the server
6      """
7      packet = bytearray(1)
8      packet[0] = 14
9      array = SOURCE_IP.split(".")
10     for a in range(len(array)):
11         packet.append(int(array[a]))
12     packet.append(packet_received[5])    # stream id
13     packet.append(packet_received[6])    # frame number
14     packet.append(packet_received[7])    # byte
15     for edge in edgeWhoRequestedStream.keys():
16         array = edgeWhoRequestedStream[edge]
17         if(packet_received[5] == array[0] and array[2] == packet_received[8]):
18             ip = socket.inet_ntoa(array[3:7])
19             packet.append(array[1])
20             sock.sendto(packet, (ip, UDP_PORT))
21             packet.pop()

```

Figura 7: Função *sendStreamToEdge()*.

2.4 Elementos de edge

No que toca aos elementos de edge, estes são parecidos com os elementos relay, no entanto, estes apenas permitem a retransmissão de streams entre os relays e os clientes. Como nos outros elementos, este começa por fazer um pedido de conexão ao SC, onde recebe como resposta os seus vizinhos que são guardados num dicionário. Após isso, o edge fica à espera de pedidos de retransmissão por parte de um cliente.

Quando este recebe um pedido por parte de um cliente, este guarda o seu endereço IP e respetivo PS (que corresponde ao ID do pedido), para que mais tarde possa saber para onde retransmitir o pacote com a *stream*, de seguida o edge envia para todos os seus relays vizinhos um pedido de transmissão da *stream* pedida pelo cliente.

```

1 def askRelayForStream(packet_received):
2     """
3     It sends a request to all the relays to ask for a stream
4
5     @param packet_received the packet received from the source
6     """
7     packet = bytearray(1)
8     packet[0] = 11
9     array = SOURCE_IP.split(".")
10    for a in range(len(array)):
11        packet.append(int(array[a]))
12    packet.append(packet_received[5]) # id_stream
13    for x in neighbours_dict:
14        if x.find("R") != -1:
15            PS4relay = random.randint(1, 255)
16            if(packet_received[6] not in idStreamDict.keys()):
17                idStreamDict[PS4relay] = packet_received[6]
18            packet.append(PS4relay) # PS
19            sock.sendto(packet, (neighbours_dict[x], UDP_PORT))
20            packet.pop()

```

Figura 8: Função *askRelayForStream()*.

Caso seja recebido um pacote proveniente do relay com uma *stream* para ser enviada para um cliente, o edge chama a função *sendStreamToClient()*, onde é comparado o PS do pacote recebido com o PS anteriormente guardado quando o cliente efetuou o seu pedido, se os PS forem iguais é então enviado o pacote com a *stream* para o respetivo cliente. Na figura seguinte encontra-se o código da função *sendStreamToClient()*.

```

1 def sendStreamToClient(packet_received):
2     """
3     It sends the stream to the client who requested it
4
5     @param packet_received the packet received from the relay
6     """
7     packet = bytearray(1)
8     packet[0] = 15
9     array = SOURCE_IP.split(".")
10    for a in range(len(array)):
11        packet.append(int(array[a]))
12    packet.append(packet_received[5])    # stream id
13    packet.append(packet_received[6])    # frame number
14    packet.append(packet_received[7])    # byte
15    for client in clientWhoRequestedStream.keys():
16        try:
17            if(idStreamDict[packet_received[8]] == clientWhoRequestedStream[client]):
18                sock.sendto(packet, (client, UDP_PORT))
19        except:
20            continue

```

Figura 9: Função `sendStreamToClient()`.

2.5 Elementos clientes

Relativamente aos elementos clientes, estes são os utilizadores finais da rede de *streaming* da nossa solução pois são estes que escolhem quando querem receber uma *stream* assim como qual das *streams* disponíveis querem receber.

A implementação deste elemento tem 3 funções principais, em que uma delas corresponde a uma thread que está à espera de receber um input do utilizador para este realizar o pedido de uma *stream*, outra que pede as *streams* disponíveis ao SC e por último a função que realiza o pedido da *stream* aos edges vizinhos. Para além destas, este elemento também realiza a conexão ao SC, o pedido de vizinhos e o constante envio de beacons.

A primeira função que corresponde ao pedido de *streams* ao SC é a apresentada na figura seguinte. Esta função cria um pacote com o tipo 6 e o IP do cliente. Visto que o IP do SC é público, estes falam entre si diretamente.

```

1 def askForStreamsAvailable():
2     """
3     It sends a packet to the SC asking for the streams available
4     """
5     packet = bytearray(1)
6     packet[0] = 6
7     array = SOURCE_IP.split(".")
8     for a in range(len(array)):
9         packet.append(int(array[a]))
10    sock.sendto(packet, (SC_IP, SC_UDP_PORT))

```

Figura 10: Função *askForStreamsAvailable()*.

Com a lista de *streams* obtida, o cliente pode então escolher qual *stream* quer receber. Para isto foi criada uma thread que está constantemente à espera de um input por parte do utilizador como apresentado a seguir.

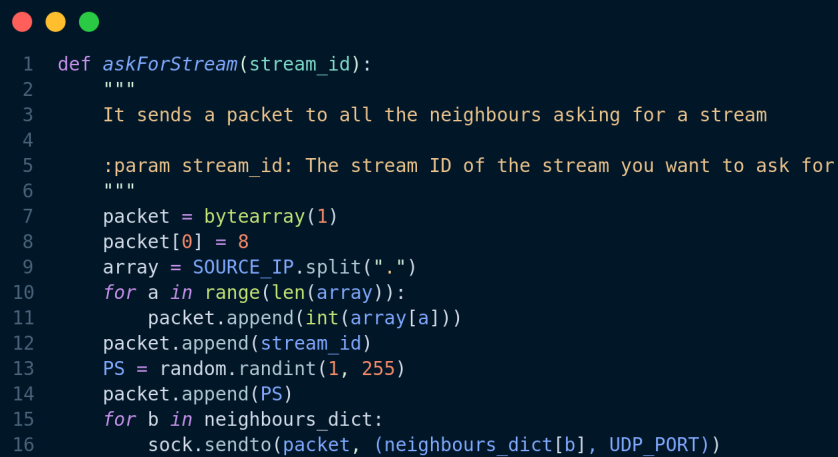
```

1 def inputStreamFromUser():
2     """
3     It asks the user to choose a stream
4     """
5     stream_selected = input("Choose the stream you want to watch:")
6     askForStream(int(stream_selected))
7     print("Selected stream: " + str(stream_selected))

```

Figura 11: Função *inputStreamFromUser()*.

Por último, com a *stream* seleccionada é procedido ao envio do pedido da *stream* para os edges vizinhos, realizado na função *askForStream()* como demonstrado na figura seguinte. Esta função recebe o id da *stream* que deve ser pedido e cria um pacote com o tipo 8, o endereço IP do cliente, o ID da *stream* passado como argumento e um ID do pedido que é obtido aleatoriamente. Após o pacote estar completo é então enviado para os edges vizinhos o pacote com o pedido de transmissão da *stream*.



```
1 def askForStream(stream_id):
2     """
3     It sends a packet to all the neighbours asking for a stream
4
5     :param stream_id: The stream ID of the stream you want to ask for
6     """
7     packet = bytearray(1)
8     packet[0] = 8
9     array = SOURCE_IP.split(".")
10    for a in range(len(array)):
11        packet.append(int(array[a]))
12    packet.append(stream_id)
13    PS = random.randint(1, 255)
14    packet.append(PS)
15    for b in neighbours_dict:
16        sock.sendto(packet, (neighbours_dict[b], UDP_PORT))
```

Figura 12: Função *askForStream()*.

3 Testes

De modo a comprovar a solução implementada foram realizados alguns testes. Foram ligados 3 clientes, 2 edges, 4 relays, 1 servidor e 1 controlo de serviços. Na próxima figura é possível observar a topologia de teste simulada com os vários serviços conectados sublinhados a azul e as ligações aos seus vizinhos sublinhados a preto.

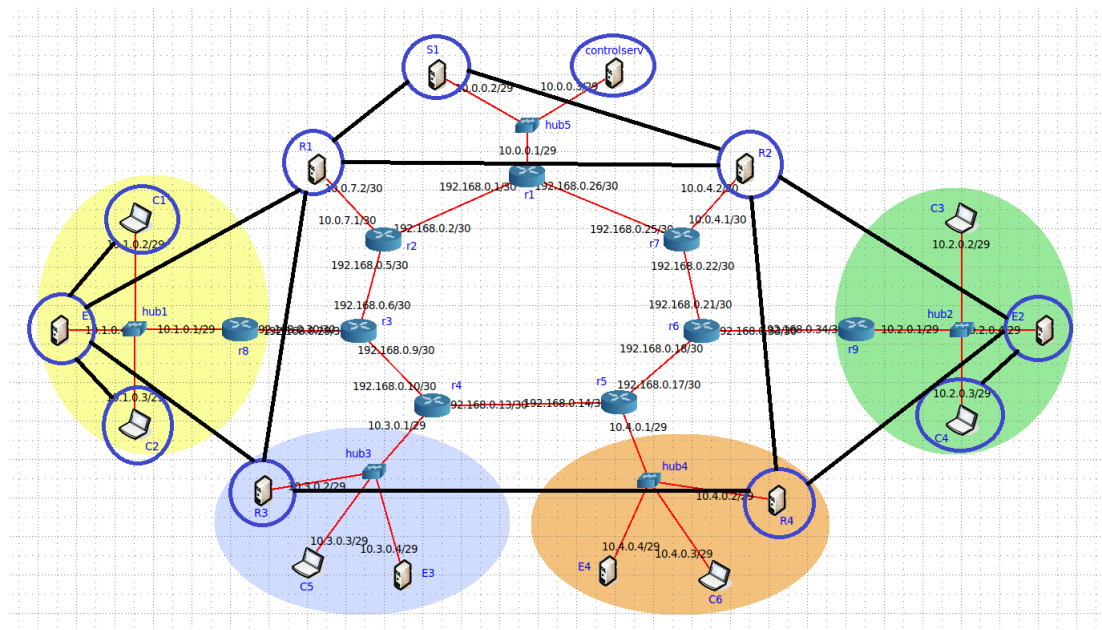
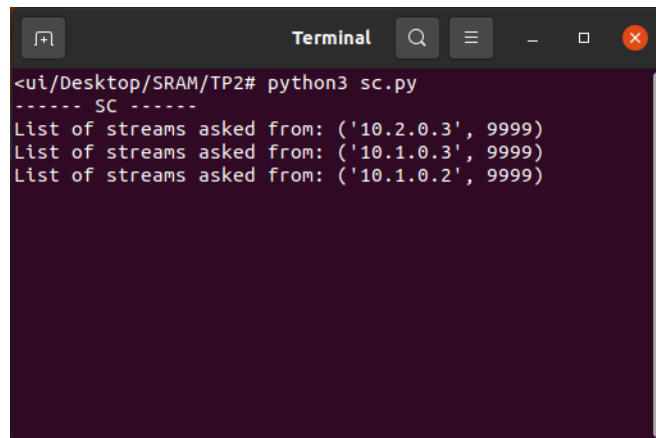


Figura 13: Topologia de teste.

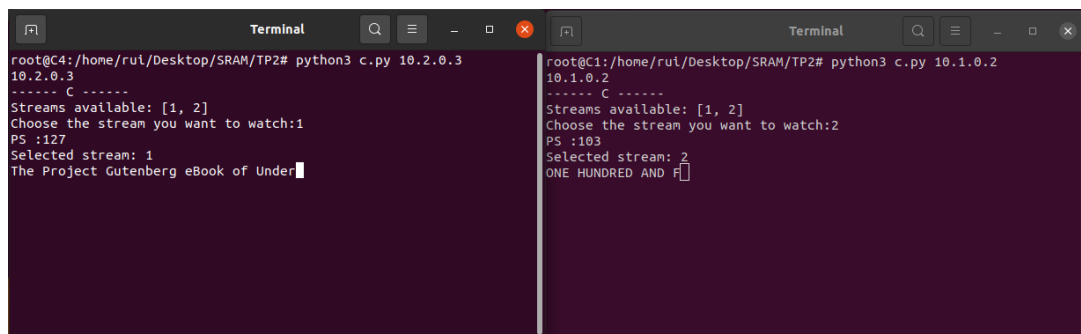
Após ter a topologia física a correr foram abertos os terminais dos vários serviços sendo que após iniciar os 3 clientes foi obtida a seguinte informação no controlador de serviços.



```
<ui/Desktop/SRAM/TP2# python3 sc.py
----- SC -----
List of streams asked from: ('10.2.0.3', 9999)
List of streams asked from: ('10.1.0.3', 9999)
List of streams asked from: ('10.1.0.2', 9999)
```

Figura 14: Terminal do controlador de serviços após conexão de 3 clientes.

Depois de iniciados 2 clientes foi acedido ao terminal de ambos em que ao correr o programa do cliente este fala com o controlador de serviços (SC) para pedir as *streams* disponíveis. Após isso estas são apresentadas aos clientes em que este tem a possibilidade de escolher qual quer receber. Na figura seguinte é possível verificar 2 clientes a requisitar *streams* diferentes e a receber os seus conteúdos, armazenados em ficheiros disponíveis ao SC.



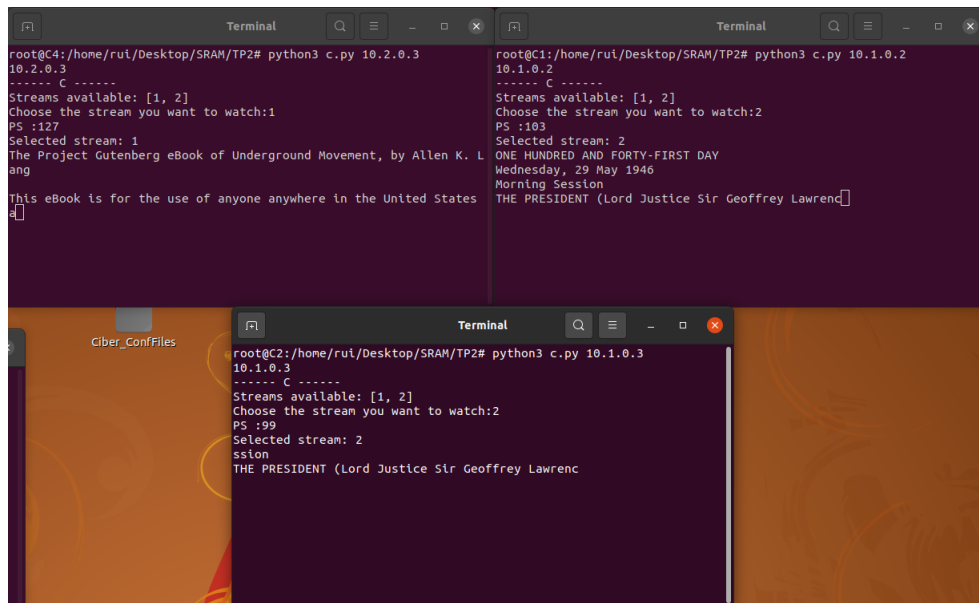
```
root@C4:/home/ru1/Desktop/SRAM/TP2# python3 c.py 10.2.0.3
10.2.0.3
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:1
PS :127
Selected stream: 1
The Project Gutenberg eBook of Under

root@C1:/home/ru1/Desktop/SRAM/TP2# python3 c.py 10.1.0.2
10.1.0.2
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:2
PS :103
Selected stream: 2
ONE HUNDRED AND F
```

Figura 15: Terminal dos clientes C1 e C2.

Com os 2 clientes a receber as *stream* foi iniciado um terceiro cliente no outro lado da topologia que pediu a transmissão da *stream* 2. É possível reparar que, como era pedido no enunciado, o cliente começa a receber conteúdo em direto, ou seja, o que este recebe é o que todos os clientes que pedissem essa *stream* recebiam (Como

comprovado pelos dados recebidos tanto no cliente 2 como no novo cliente).



```
root@C4:/home/ru/Desktop/SRAM/TP2# python3 c.py 10.2.0.3
10.2.0.3
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:1
PS :127
Selected stream: 1
The Project Gutenberg eBook of Underground Movement, by Allen K. L
ang
This eBook is for the use of anyone anywhere in the United States
a]

root@C1:/home/ru/Desktop/SRAM/TP2# python3 c.py 10.1.0.2
10.1.0.2
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:2
PS :103
Selected stream: 2
ONE HUNDRED AND FORTY-FIRST DAY
Wednesday, 29 May 1946
Morning Session
THE PRESIDENT (Lord Justice Sir Geoffrey Lawrenc]

root@C2:/home/ru/Desktop/SRAM/TP2# python3 c.py 10.1.0.3
10.1.0.3
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:2
PS :99
Selected stream: 2
ssion
THE PRESIDENT (Lord Justice Sir Geoffrey Lawrenc
```

Figura 16: Terminal dos clientes C1, C2 e C3.

De modo a testar se após um cliente se desconectar havia a possibilidade de este voltar a receber conteúdo foi realizada a desativação de um cliente. O controlador de serviços apresentou o conteúdo da seguinte figura em que este obteve a informação que um cliente ficou offline devido aos *beacons* enviados entre a rede.


```
Terminal
<ui/Desktop/SRAM/TP2# python3 sc.py
----- SC -----
List of streams asked from: ('10.2.0.3', 9999)
List of streams asked from: ('10.1.0.3', 9999)
List of streams asked from: ('10.1.0.2', 9999)
OFFLINE: 10.1.0.2
```

Figura 17: Terminal do controlador de serviços após desconexão de um cliente.

Voltando a ativar o cliente desconectado (cliente da direita na figura) é possível observar que este, após requisitar a *stream* número 1, começa a receber na mesma posição e ao mesmo tempo que o cliente da esquerda na figura seguinte.

```
Terminal
root@C4:/home/ru/Desktop/SRAM/TP2# python3 c.py 10.2.0.3
10.2.0.3
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:1
PS :127
Selected stream: 1
The Project Gutenberg eBook of Underground Movement, by Allen K. Lang
ang
This eBook is for the use of anyone anywhere in the United States
and most other parts of the world at no cost and with almost no re
strictions whatsoever. You may copy it, give it away or re-use it
t under the terms of the Project Gutenberg License included with t
his eBook or online at www.gutenberg.org. If you a

Terminal
root@C1:/home/ru/Desktop/SRAM/TP2# python3 c.py 10.1.0.2
10.1.0.2
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:1
PS :250
Selected stream: 1
enberg License included with this eBook or online at www.gutenberg.
org. If you a

Terminal
root@C2:/home/ru/Desktop/SRAM/TP2# python3 c.py 10.1.0.3
10.1.0.3
----- C -----
Streams available: [1, 2]
Choose the stream you want to watch:2
PS :99
Selected stream: 2
ssion
THE PRESIDENT (Lord Justice Sir Geoffrey Lawrence): The Tribunal will
adjourn this afternoon at 4 o'clock in order to sit in closed sess
ion.
MR. THOMAS J. DODD (Executive Trial Counsell for the United States):
Mr. President, the day before yesterday the Tribunal asked if we wo
uld ascertain
```

Figura 18: Terminal dos clientes após desconexão.

4 Conclusão

Este trabalho teve como objetivo principal a implementação de uma rede aplicacional de Streaming live, bem como a familiarização com as redes Overlay. Durante a realização do código o obstáculo mais difícil foi o controlo de pedidos de streaming, nomeadamente saber quando enviar e para que cliente enviar, já a questão de controlar a topologia foi realizada com uma maior facilidade.

Apesar do grupo não ter conseguido implementar todas as funcionalidades que estavam presentes no enunciado do trabalho, este considera que foi realizado um trabalho bastante satisfatório tendo em conta que foi realizado por apenas 2 pessoas.