Trabalho Prático 3 - Serviço *Over the Top* para entrega de multimédia

Ariana Lousada[1,2 PG47034], Carlos Gomes[2,3 PG47083], and Tiago Sousa[2,3 PG47684]

[1] Universidade do Minho, Braga, Portugal `https://www.uminho.pt/`
[2] Departamento de Informática, Universidade do Minho, Braga
`https://www.di.uminho.pt/`

1 Introdução

O presente documento foi desenvolvido no âmbito da disciplina de Engenharia de Serviços e Rede com vista na criação de um programa de transmissão de vídeo e *streaming* utilizando o protocolo de transporte TCP. Um dos grandes exemplos deste tipo de serviço será a Netflix, Hulu ou até mesmo a Twitch, que tem subido em popularidade ao longo dos últimos meses.

Neste relatório são apresentadas todas as etapas de desenvolvimento desde a conceção da solução até à construção da mesma, detalhando as várias decisões tomadas pela equipa de trabalho ao longo das várias etapas.

Este trabalho prático foi desenvolvido na linguagem de programação `Python`.

2 Arquitetura da Solução

O principal objetivo do presente trabalho prático rege-se na construção de uma rede *overlay peer-to-peer* a partir da qual seja possível a um *streamer* fazer *stream* de um vídeo para vários clientes que assim o pretendam.

A solução desenvolvida para o problema é, muito resumidamente, uma máquina de eventos. Todos os nós pertencentes à topologia lidam com eventos de alguma forma:

- Streamer - Imprime eventos;
- Relay(nodo intermédio) - Recebe mensagens e reencaminha-as;
- Cliente - Recebe mensagens e faz *requests* de *stream*.

Assim, a solução construída consiste em três tipos de nós: *streamers*, clientes e nós intermédios(*relays*). Os nós intermédios são as entidades responsáveis pelo reencaminhamento de pacotes para os clientes(assim como para outros nós intermédios) e contêm uma lista de nós da rede que naquele instante pertencem ao fluxo de dados (e respetivas distâncias em relação ao nodo atual). Os nós pertencentes a esta lista correspondem às entidades que estão de momento a transmitir pacotes, isto é, nós que têm acesso a dados de *stream*.

Um nó intermédio, caso seja o primeiro a ligar-se na rede pode também assumir o papel de *bootstrapper*. O *bootstrapper* aguarda que os seus vizinhos estabeleçam conexões com ele. A potencial falha deste *bootstrapper* não irá influenciar o funcionamento da rede, visto que a rede é construída utilizando uma estratégia descentralizada.

3 Especificação dos protocolos

3.1 Formato das mensagens protocolares

As mensagens protocolares enviadas entre os diferentes nós da rede podem ser de um dos seguintes quatro tipos:

- `HELLO` ("H") : Mensagem de *"Hello world"*; Utilizada maioritariamente em testes de conexões entre nós.
- `HEARTBEAT` ("D") : Mensagem de *heartbeat*; Utilizada para atualizar as informações na rede, maioritariamente informações relacionadas com a localização dos dados de *stream*.
- `REQUEST` ("R") : Mensagem de *request* de *stream*; Utilizada pelos *relays* (nós intermédios) e pelos clientes para pedir dados de uma *stream*.
- `STREAM` ("S") : Mensagem de *stream*; Este tipo de mensagens contém pacotes RTP de *stream*, que são reencaminhados pelos nós intermédios até chegarem aos clientes.

3.2 Interações

Como já anteriormente mencionado, os diferentes nós da rede vão comunicando entre si através dos vários tipos de mensagens trocadas entre eles.

As interações da rede *overlay* são totalmente dependentes de mensagens do tipo *heartbeat*.

Este *heartbeat* consiste numa mensagem que é enviada pelo *streamer* para o seu *peer*(o nodo intermédio vizinho) a cada dois segundos. Por sua vez, o nodo intermédio vai reencaminhar essa mensagem para os seus vizinhos e assim sucessivamente, até chegar a todos os nós da topologia.

Inicialmente, o *streamer* envia por multicast dados da sua *stream* para o nodo intermédio ao qual está conectado. Por sua vez, este nó intermédio passa a fazer parte do fluxo de dados. Esta informação é depois transmitida a toda a rede através de um *heartbeat*, que vai possibilitar a atualização das distâncias aos dados de *stream* em cada nodo da rede.

Quando um cliente faz *request* de uma *stream* este consulta a sua lista de nós pertencentes ao fluxo de dados. Avaliando as distâncias, é escolhido o nó ao qual o cliente esteja mais próximo e envia uma mensagem do tipo `REQUEST`

para o seu nó intermédio vizinho, que vai ser reencaminhada até chegar ao nodo pretendido.

Após receção do *request*, o nodo pretendido irá então enviar mensagens sucessivas do tipo `STREAM`, que contêm pacotes RTP. A stream para o cliente é feita em *unicast*.

4 Implementação

Para a construção da solução, foram estabelecidas quatro etapas distintas. Contudo, visto que a etapa 3 consiste em apenas testes relacionados com a rede *overlay*, apenas vão ser expostos o raciocínio e decisões tomadas na primeira, segunda e quarta etapas.

4.1 Etapa 1 - Construção da rede *Overlay*

Para a construção da rede *overlay* é inicialmente ligado um nó intermédio da topologia(*bootstrapper*). Este nó vai então aguardar por conexões dos seus vizinhos. Não existe a possibilidade deste ficar à escuta por tempo indeterminado, visto que logo que o *streamer* entre em execução manda dados automaticamente ao seu vizinho. Com os eventuais *requests* por parte dos clientes e *heartbeats* do *streamer*, vão ser estabelecidas ligações por toda a extensão da rede.

4.2 Etapa 2 - Construção das rotas para os fluxos

Para construção das rodas de fluxo de dados, foi desenvolvida uma estratégia que fosse aplicável também em contexto real, com auxílio a mensagens do tipo *heartbeat*. Como já anteriormente mencionado, estas mensagens permitem que cada nó da topologia possua informações sobre os nós pertencentes ao fluxo de dados e respetivas distâncias. De acordo com esta medida de distância (que neste caso corresponde ao número de saltos), o cliente irá fazer um *request* ao nodo pertencente ao fluxo de dados mais próximo.

4.3 Etapa 4 - *Streaming*

Para a *stream* foram implementados um cliente e um *streamer* simples. Cada *frame* do vídeo é então eviada encapsulada num pacote RTP do *streamer* ao cliente. Cada frame recebida é armazenada numa pasta "cache".

Etapa 4.1 - Definição de RTP Packets Para utilizar RTP Packets, é necessário analisar o funcionamento destes e o papel que irão ter ao longo da comunicação entre nós - os dados de um vídeo ou de um áudio vão ser encapsulados neste tipo de pacotes, o que leva a que sejam indispensáveis num serviço de transmissão de *stream*.

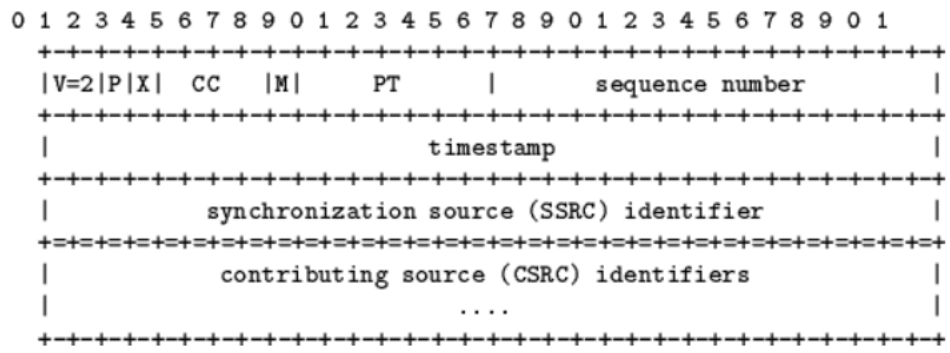

Fig. 1. Formato de um RTP *packet*

Cada pacote deve tem um *header* no mínimo de 12 bytes. Os campos utilizados neste tipo de pacotes são:

- Version (2 bits);
- Padding (1 bit);
- Extension (1 bit);
- CSRC Count (4 bits);
- Marker (1 bit);
- Payload Type (7 bits) (26 - Motion JPEG);
- Sequence Number (16 bits);
- Timestamp (32 bits);
- SSRC (32 bits);
- CSRC (32 bits);

Estes pacotes podem também ter uma extensão de *header* opcional.

```
def encode(self, V, P, X, CC, seqNum, M, PT, SSRC, payload):
    timestamp = int(time())
    header = bytearray(HEADER_SIZE)

    header[0] = header[0] | V << 6;
    header[0] = header[0] | P << 5;
    header[0] = header[0] | X << 4;
    header[0] = header[0] | CC;
    header[1] = header[1] | M << 7;
    header[1] = header[1] | PT;
    header[2] = (seqNum >> 8) & 0xFF;
    header[3] = seqNum & 0xFF;
    header[4] = (timestamp >> 24) & 0xFF;
    header[5] = (timestamp >> 16) & 0xFF;
    header[6] = (timestamp >> 8) & 0xFF;
    header[7] = timestamp & 0xFF;
    header[8] = (SSRC >> 24) & 0xFF;
    header[9] = (SSRC >> 16) & 0xFF;
    header[10] = (SSRC >> 8) & 0xFF;
    header[11] = SSRC & 0xF
    self.header = header
    self.payload = payload
```

Fig. 2. Código do cabeçalho dos pacotes RTP.

Etapa 4.2 - Cliente à escuta de RTP Packets; Play da stream Para dar a possibilidade ao cliente de consumir a stream, este tem de obrigatoriamente receber pacotes que contenham a informação necessária. Para esse efeito, cria uma conexão através de um *socket* e fica sempre à espera de receber os pacotes.

No momento de receção, irá ser feito um `decode` de forma a obter o pacote RTP inserido na mensagem do tipo `STREAM`. Também irá utilizar o *payload* do packet para atualizar o frame no Tk[3].

```
def listenRTP(self):
    while True:
        time.sleep(0.05)
        data = self.sock.recv(20480)
        if data:
            rtp_packet = RtpPacket()
            rtp_packet.decode(data)
            self.updateMovie(self.writeFrame(rtp_packet.getPayload()))
            self.master.update()
```

Fig. 3. Excerto de código: Escuta

[3] Biblioteca utilizada para a reprodução da *stream*.

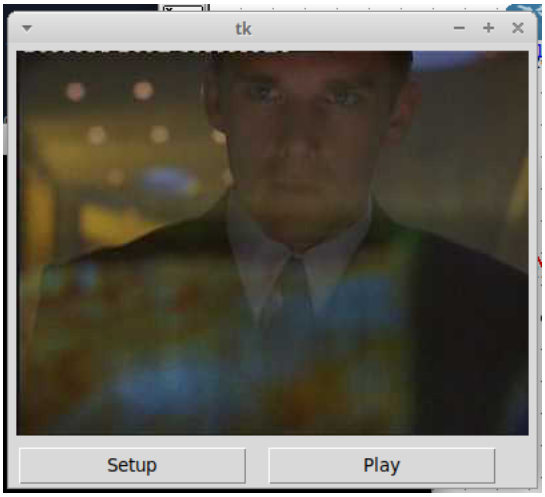

Fig. 4. Vizualização da stream

4.4 Problemas de implementação

Apesar de ser possível enviar os vários pacotes de *stream* entre os vários nós da rede, não foi possível à equipa de trabalho enviar um vídeo por completo na rede *overlay*, apenas as frames do vídeo. Isto poderá ser algum erro relacionado com a construção ou com o `unpack` dos pacotes RTP das mensagens protocolares. Infelizmente, foi um problema que a equipa de trabalho não foi capaz de solucionar.

5 Testes e resultados

5.1 Cenários construídos

De modo a ser possível testar a solução desenvolvida, foi essencial ter em conta vários cenários de forma a verificar possíveis falhas nos protocolos e estratégias aplicadas. Assim, com base em algumas propostas por parte da equipa docente, foram construídos três cenários distintos.

Cenário 1 O primeiro cenário consiste em apenas um *streamer*, dois clientes e um nó intermédio.

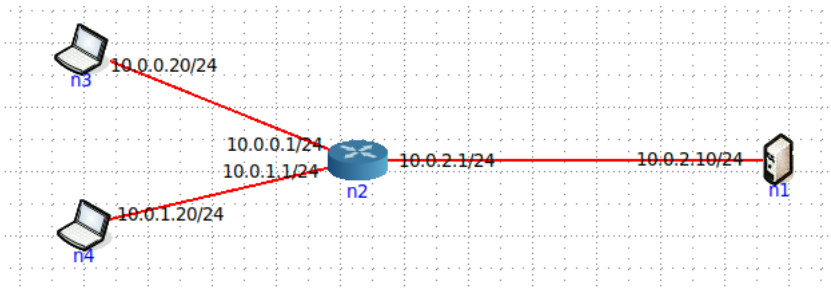

Fig. 5. Cenário 1

Neste caso, o primeiro nó a ser ligado vai ser o nó intermédio. Como este nó coincide com o vizinho do *streamer*, este recebe imediatamente dados de *stream*. Entretanto, um *heartbeat* é enviado para o resto da topologia(neste caso, para os dois clientes).

Quando um dos clientes faz um *request*, este é enviado ao nó intermédio, que já faz parte do fluxo de dados. Por sua vez, o nó intermédio envia pacotes ao cliente para consumo.

Cenário 2 O segundo cenário consiste em apenas um *streamer*, dois nós intermédios e dois clientes.

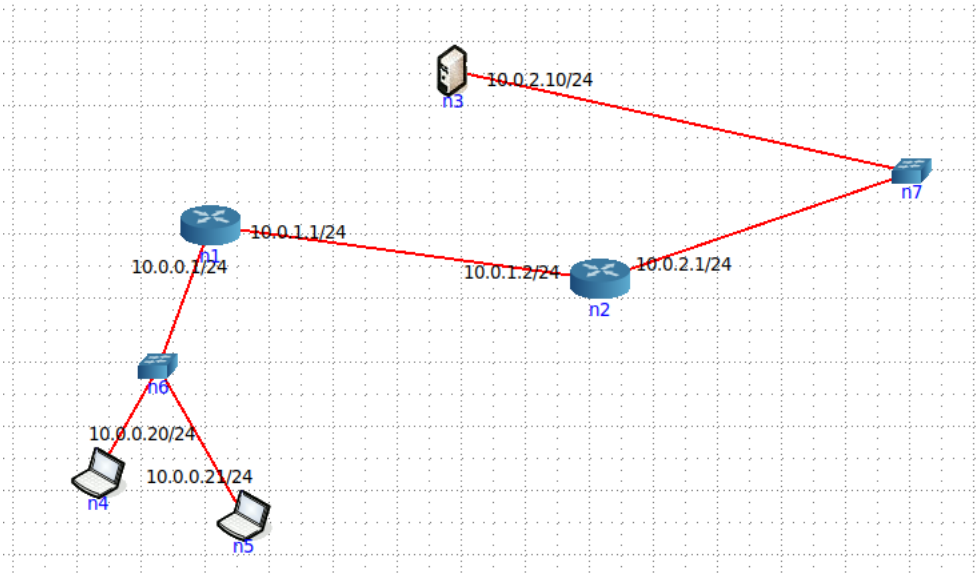

Fig. 6. Cenário 2

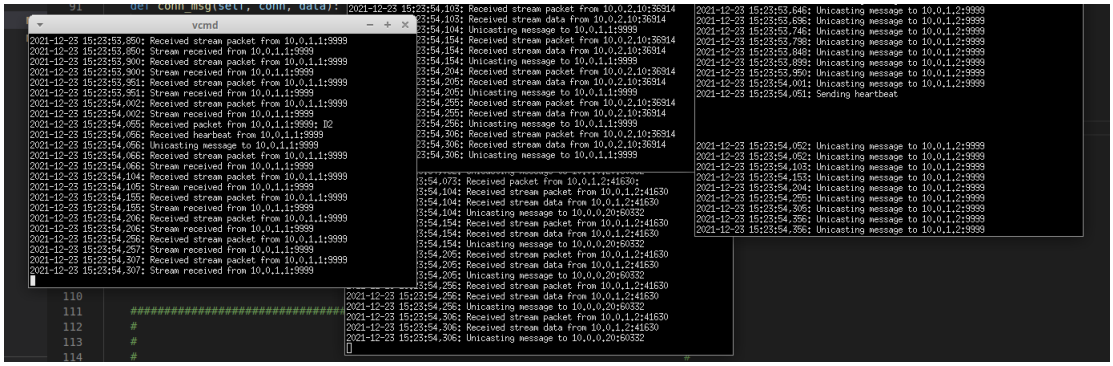

Fig. 7. Teste - Cenário 2(da esquerda para a direita: Cliente, Nós intermédios e Streamer.)

Tal como é possível observar na figura 7, os nós intermédios reencaminham as mensagens recebidas do *streamer* para o cliente.

Cenário 3 O terceiro cenário consiste em dois clientes, quatro nós intermédios e dois *streamers*.

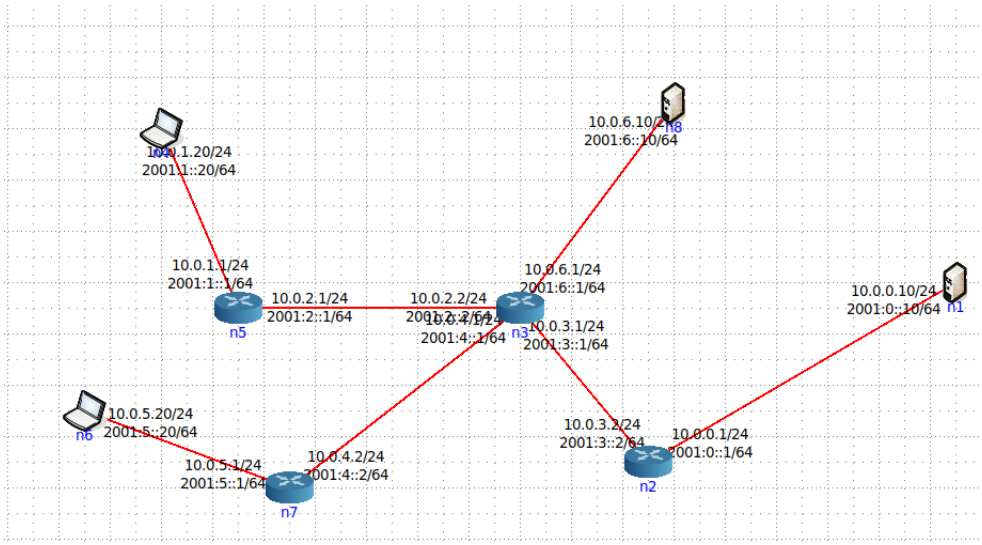

Fig. 8. Cenário 3

Neste cenário, independentemente do cliente que fizer *request* primeiro e do streamer, o nó `n3` irá sempre pertencer ao fluxo de dados. Consequentemente, a

mensagem de *request* irá ser sempre reencaminhada até este nodo, que por sua vez irá enviar mensagens com pacotes RTP.

Cenário 4 Para o quarto e último cenário, construiu-se uma topologia semelhante à disponibilizada por parte da equipa docente no enunciado do trabalho prático.

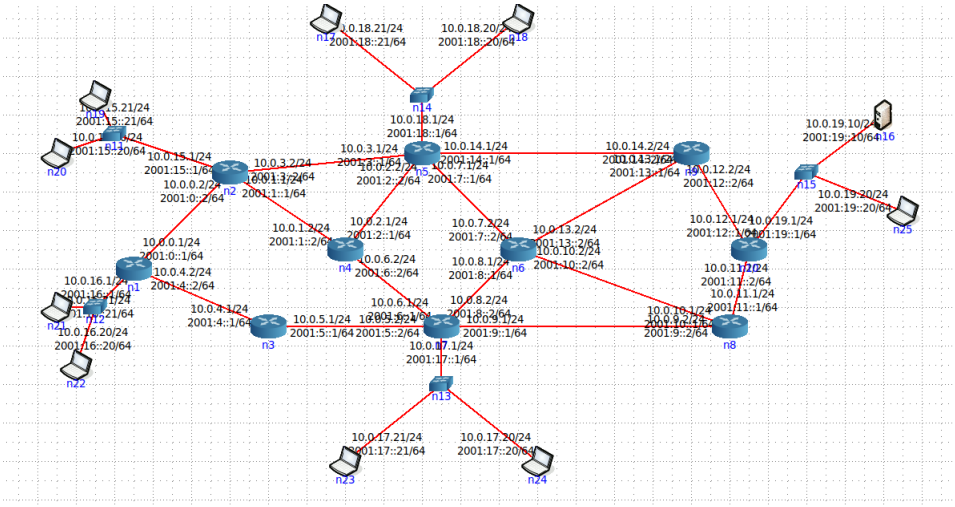

Fig. 9. Cenário 4

Para este cenário, tendo em conta a seguinte situação: o cliente `n23` mandou um *request* e está de momento a receber dados de *stream*.

Na eventualidade de, por exemplo, o cliente `n22` mandar um *request*, o nodo mais próximo pertencente ao fluxo de dados vai ser o `n7`. Com isto, o *request* vai ser reencaminhado até este nodo, que por sua vez irá enviar mensagens com RTP *packets* para o cliente `n22`.

6 Conclusões e trabalho futuro

A realização deste trabalho ofereceu-nos a oportunidade de aprofundarmos os nossos conhecimentos relativamente ao funcionamento de um protocolo de streaming, assim como a comunicação entre diferentes clientes e servidores e a transmissão de dados através de RTP Packets.

Graças ao desenvolvimento deste trabalho prático, conseguimos aplicar os conceitos lecionados nas aulas teóricas da unidade curricular, o que resultou na consolidação de assuntos relativos a estes tópicos.

Trabalho futuro passaria pela implementação de TLS na comunicação entre packets assim como um mecanismo de autenticação para permitir que a rede descentralizada seja segura.