# Engenharia de Serviços em Rede
# Trabalho Prático 2: Serviço *Over the Top* para entrega de multimédia

Ana Murta (**pg50184**)
Ana Henriques (**pg50196**)
Francisco Peixoto (**pg47194**)

**Grupo 5 - PL7**

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mails: {pg50184, pg50196, pg47194}@alunos.uminho.pt

**Resumo** Proposto no âmbito da Unidade Curricular de Engenharia de Serviços em Rede, o presente trabalho prático consiste no desenvolvimento de um protótipo de entrega de vídeo com requisitos de tempo real, satisfazendo os pedidos de um conjunto de clientes através de um servidor de conteúdos. Para validar a solução, recorrer-se-á ao emulador CORE e utilizar-se-á alguns cenários de teste.

**Keywords:** *Streaming* · UDP · Pacotes RTP · *Flooding* · RTSP · *oNode*

## 1 Introdução

Para cumprir com o objetivo resumidamente apresentado, é preciso construir uma rede *overlay* aplicacional, em que alguns nodos atuam como intermediários no reenvio eficiente dos dados, evitando atrasos significativos e o uso desnecessário de recursos, como a largura de banda. A solução final deve, por isso, possuir uma arquitetura que garanta a qualidade do serviço de entrega a fim de contribuir para uma melhor experiência do utilizador.

Ambicionando, então, a eficiência e otimização da entrega dos dados, a estratégia aplicada pelo grupo de trabalho foi descobrir os caminhos mais curtos entre os nodos, desde o servidor até aos respetivos clientes, definindo como métrica o tempo de envio e o número de saltos entre os nodos da rede *overlay*.

O desenvolvimento deste trabalho prático foi conseguido com recurso à linguagem de programação *Python* visto possuir uma vasta biblioteca sobre programação com *sockets* e *threads*, o que facilitou a concretização da solução final. No decorrer deste relatório, serão explicadas todas as decisões tomadas ao longo do desenvolvimento do trabalho prático, começando por modelar a arquitetura da solução (**Secção 2**); apontar os protocolos usados e a respetiva interação com os mesmos (**Secção 3**); especificar a implementação elaborada (**Secção 4**); apresentar os testes e os resultados obtidos (**Secção 5**); e efetuar uma breve introspeção acerca do produto final e ideias para trabalho futuro (**Secção 6**).

## 2 Arquitetura da Solução

Na Figura 1, temos um diagrama simples e intuitivo, no qual estão representadas as classes que, em conjunto, permitem a implementação deste protótipo.

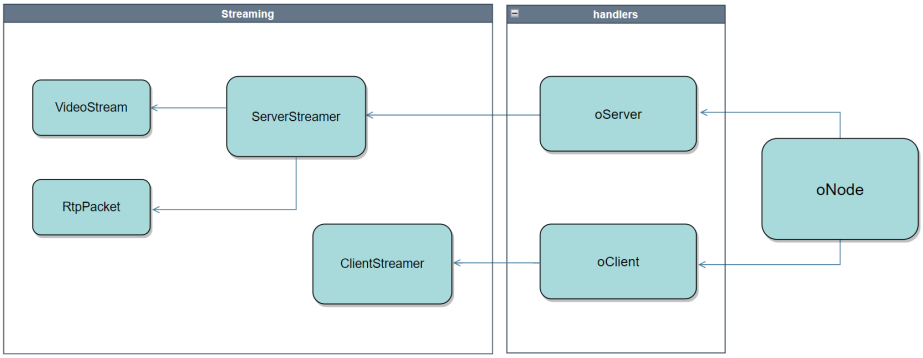


**Figura 1.** Arquitetura da Solução Final

Para construir a topologia, optou-se pela abordagem baseada num controlador, que sugere a indicação de um nodo como contacto para arranque da rede: `$ oNode <bootstrapper>`. Como tal, e de modo a que cada nó conheça os seus vizinhos, foram criados ficheiros *json* para cada um dos nós, nomeados por `node_info{node_name}.json`, cuja informação será, posteriormente, explicada com mais cuidado na Secção 4. Na topologia construída, são identificados três tipos de nodos: o **Servidor**, responsável por escutar pedidos e enviar ficheiros; ***Big Nodes***, nodos intermediários que, além das funcionalidades do Servidor, também fazem pedidos e, na situação de não possuirem os ficheiros de multimédia, pedem-nos ao Servidor; e ***Nodes***, que apenas realizam pedidos. Apesar das suas diferenças, estes 3 tipos têm em comum a sua participação no *flooding*.

Relativamente ao *flooding*, que acontece periodicamente de 30 em 30 segundos, cada nodo envia uma mensagem aos seus nodos vizinhos e, se os seus vizinhos lhe responderem, ele considera que os mesmos estão ativos. É essencial denotar que são descartadas as mensagens com um elevado número de saltos.

Sempre que é feito um pedido de *streaming*, o nodo que recebe o pedido tem de verificar se possui esse ficheiro de multimédia. Assim que encontra esse recurso, o mesmo envia uma mensagem ao nodo que fez o pedido, informando-o que encontrou o ficheiro, e estabelece uma conexão P2P (*peer-to-peer*). Na situação do recurso não ser encontrado, o cliente é também informado do sucedido, assim como no caso da conexão ser perdida.

Uma arquitetura P2P é caracterizada por *peers* que requisitam serviços a outros *peers*, e, de igual forma, prestam serviços a outros *peers*. Deste modo,

cada *peer*, representado por um *Big Node* na nossa topologia, atua simultaneamente como cliente e servidor da rede, não exigindo ter um servidor central a coordenar o roteamento das mensagens. Além da descentralização, esta arquitetura faculta um aumento da autoescabilidade já que novos *peers* proporcionam melhor capacidade de serviço.

Outro ponto importante a referir é o cálculo do *Big Node* (ou do próprio Servidor, se for o caso) mais próximo de cada *Node*: se a diferença de tempo de um nodo simples a um *Big Node* for menor do que a qualquer outro, ou alternativamente a diferença de tempo for igual, mas o número de saltos menor, esse *Big Node* é então considerado o mais perto, sendo ele responsabilizado por realizar o serviço de *streaming* ao nodo simples se contiver o vídeo requerido.

### 2.1 Classes Constituintes

**RtpPacket** A classe `RtpPacket` é responsável por codificar e descodificar pacotes RTP (*Real-time Transport Protocol*), contendo métodos que acessam e configuram os vários campos do cabeçalho, bem como recuperam essas informações, nomeadamente o número de sequência, o *timestamp* e também o *payload* do pacote. O protocolo RTP disponibiliza funções de transporte de rede *end-to-end* que transmitem dados multimédia em tempo real numa rede. Deste modo, esta classe é utilizada para criar pacotes RTP com os campos de cabeçalho pretendidos e, ainda, analisá-los ao receber pacotes RTP.

**VideoStream** A classe `VideoStream` lê o vídeo do ficheiro e retorna a próxima *frame* do mesmo depois de ler os primeiros 5 *bits* do ficheiro indicado, que contém o comprimento da *frame* atual, incrementando o seu valor. Esta classe também se encarrega de rastrear o número da *frame* atual. Inicialmente, o número da *frame* é definido como zero, o que indica que ainda nenhuma foi lida.

**ClientStreamer** A classe `ClientStreamer` é um *script* do lado do cliente que cria uma GUI para um *player* de *streaming* de vídeo, recorrendo à biblioteca `Tkinter`. De modo a cumprir com esse fim, esta classe inclui funções para estabelecer uma conexão com o Servidor através de um *socket* UDP; enviar mensagens RTSP a partir desse *socket* UDP gerado, indicando o tipo de pedido e o número de sequência; e ouvir os pacotes RTP recebidos para depois ser feita a extração dos dados da imagem de cada pacote RTP e a sua exibição. Os tipos de pedidos feitos ao Servidor são: `SETUP`, para configurar a conexão para transmitir o vídeo; `PLAY`, para iniciar o *streaming* após a respetiva conexão ter sido feita; `PAUSE`, para parar o *streaming* até que o mesmo seja (ou não) retomado; e `TEARDOWN`, para interromper a conexão e terminar o *streaming*.

**ServerStreamer** A classe `ServerStreamer` é um *script* do lado do servidor que transmite um vídeo por RTSP, contendo funções que recebem, processam e respondem aos pedidos RTSP de um cliente. Dentro das várias funções constituintes

desta classe, a função `sendRtp()` lê a próxima `frame` do vídeo, codifica-a num pacote RTP e envia-a por UDP para o cliente, enquanto que a `recvRtspRequest()` recebe pedidos RTSP feitos pelo cliente e chama a `processRtspRequest()` para os processar depois. Dependendo do tipo de pedido, o processamento do mesmo segue diferentes procedimentos. Por exemplo, se for um pedido `SETUP`, serão enviadas respostas RTSP para o cliente e será criado um objeto `VideoStream` responsável por ler ficheiro de multimédia. No entanto, no caso de ser um pedido `PLAY`, será gerado um novo `socket` para RTP/UDP e iniciar-se-á uma nova `thread` para enviar pacotes RTP ao cliente.

**oNode** A classe `oNode` representa a aplicação que implementa o protótipo de entrega de vídeo, começando por inicializar o nó passado como argumento e, seguidamente, enviar uma mensagem a todos os seus nós vizinhos na rede *overlay* dentro de um determinado número de saltos – ***flooding***. A mensagem contém informações sobre o nó, desde o endereço IP da sua interface, à porta pela qual acontecerá o flooding, à porta pela qual será feito o *streaming* do vídeo, até à indicação se é, ou não, um *Big Node* ou o Servidor. Nesta classe, está definida ainda uma string `PACKET_FORMAT` de maneira a ser possível a codificação da mensagem num formato binário. Numa fase posterior, o *script* recorre aos módulos `handlers.oClient` e `handlers.oServer` para criar os objetos cliente e servidor, que serão usados para lidar com as mensagens recebidas e enviadas, respetivamente. Para além do *flooding*, a aplicação `oNode` também se encarrega de utilizar o objeto servidor para realizar o *streaming* do vídeo requisitado e, seguidamente, exibido ao objeto cliente – ***streaming***. Todas as estratégias e decisões tomadas, que possibilitaram o desenvolvimento desta aplicação, serão detalhadamente descritas mais à frente, na Secção 4.

## 3 Especificação dos Protocolos

### 3.1 Formato das Mensagens Protocolares

O protocolo de *flooding*, que ajuda os nodos a entender qual o melhor caminho para cumprir os pedidos de *streaming*, é *peer-to-peer*, descentralizado e dinâmico. Este protocolo tem por base o envio de mensagens e, para isso, definiu-se um conjunto de variáveis locais na aplicação `oNode`, cuja função é precisamente auxiliar no processo de *flooding*.

O dicionário "message", que representa a mensagem enviada no processo do *flooding*, contém as seguintes informações:

- **node_id**: o nó atual;
- **flood_port**: a porta utilizada para o *flooding*;
- **stream_port**: a porta utilizada para o *streaming*;

- **tempo**: uma lista de dois elementos `datetime`, representando o tempo em que a mensagem foi enviada e a diferença entre o tempo em que a mensagem foi enviada e quando foi recebida;
- **saltos**: o número de saltos feitos desde que a mensagem foi enviada, inicializado com zero;
- **last_refresh**: o último *refresh* da mensagem que, na versão final da aplicação, poderia ajudar a determinar que nós foram perdidos e excluí-los das informações locais;
- **is_server**: um booleano indicando se o nó é um servidor;
- **is_BigNode**: um booleano indicando se o nó é um *big node*;
- **nearest_server**: uma lista dos servidores mais próximos, ordenada por proximidade, com seu IP, porta de *streaming* e delta.

### 3.2 Interações

Objetivando implementar a rede *peer-to-peer* de forma dinâmica, cada nodo irá correr aquando da sua inicialização um ficheiro de *bootstrapping* que consiste na inicialização das suas variáveis locais.

Após a inicialização do nodo, o mesmo irá correr três *threads*: `refresh_table`, que executa o processo de flooding; `ClientUI`, isto se for um nodo cliente que pretende fazer pedidos; e `server.stream`, executada adicionalmente quando é um nodo servidor, permitindo receber pedidos dos clientes e efetuar *streaming*.

***Flooding*** Dentro do processo de *flooding*, a função `message_handler` irá criar um *socket* UDP através do endereço IP da interface do nodo e da sua porta de *flooding*, sendo esse *socket* então passado à *thread* relativa ao envio contínuo de mensagens de *flooding*, e à *thread* que trata da receção de mensagens de *flooding*. Aquando da receção de uma mensagem de *flooding*, as informações da mensagem são atualizadas com a hora atual e o número de saltos antes de ela ser novamente enviada, bem como a lista de servidores mais próximos.

***Streaming*** No processo de *streaming*, se um nodo fizer um pedido de *stream*, o mesmo será respondido pelo *Big Node* mais próximo, guardado numa lista nomeada por `nearest_server` que está guardada na mensagem de *flooding*. Todavia, se esse *Big Node* não contiver o vídeo solicitado, então o pedido passará a ser respondido pelo segundo *Big Node* presente na tal lista, e assim sucessivamente.

## 4 Implementação

### 4.1 Construção da Topologia *Overlay*

O grupo de trabalho acabou por implementar a abordagem baseada num controlador – *bootstrapper*, sugerida pela estratégia 2 da segunda etapa, para

construir a topologia *overlay*, na qual existem *Nodes*, *Big Nodes* e o Servidor, tal como foi mencionado na Secção 2. Cada nó está associado a um ficheiro *json* − `node_info{node_name}`, que contém informações acerca do tal nó, sendo o modelo desse ficheiro apresentado abaixo:

```
{
  "node_id": "10.0.0.20",
  "port_flooding": 3005,
  "port_streaming": 4005,
  "ports": [
              {"ip": "10.0.0.10", "port": 3000},
              {"ip": "10.0.1.21", "port": 3015},
              {"ip": "10.0.1.22", "port": 3010}
            ],
  "is_bigNode": false,
  "is_server": false,
  "connections": ["10.0.0.10","10.0.1.21","10.0.1.22"]
}
```

As únicas chaves deste modelo *json* que ainda não foram interpretadas no presente relatório são "ports"e "connections", que correspondem à lista de portas dos vizinhos e à lista de vizinhos do nodo descrito pelo ficheiro.

### 4.2 Serviço de *streaming*

No que toca ao serviço de *streaming*, o grupo implementou um cliente e um servidor simples com base no código do livro de apoio facultado pela equipa docente, onde o servidor é capaz de ler o vídeo de um ficheiro, enviando-o por pacotes numerados ao cliente que, por sua vez, reproduzi-lo-á numa nova janela. Nessa janela, existem 4 botões que, ao serem selecionados, são traduzidos como pedidos: `SETUP`, `PLAY`, `PAUSE`, `TEARDOWN`.

Para que o pedido possa ser enviado ao servidor, recorrendo a uma estrutura de mensagem que indica o nome do vídeo, o número de sequência, o *hostname* e a porta RTP, é preciso que cada tipo de pedido esteja associado a um estado anterior: `SETUP -> INIT`; `PLAY -> READY`; `PAUSE -> PLAYING`; `TEARDOWN -> NOT INIT`. Sendo isto verificado com sucesso, o servidor recebe, então, um pedido ao qual responderá, seguindo o que foi explicitado na Secção 2 quando as classes `RtpPacket`, `VideoStream` e `ServerStreamer` foram apresentadas.

Assim que o pacote RTP chega ao cliente, a imagem do vídeo é transmitida numa ordem específica, consoante o número de sequência de cada frame, e não pela ordem de chegada. Se o número de sequência atual for maior que o número da frame, o pacote é descartado.

### 4.3 Monitorização da rede *overlay*

A terceira etapa deste trabalho prático consistiu no desenvolvimento de um processo de monitorização da rede *overlay*, na qual todos os nodos difundem

periodicamente uma mensagem a fim de estar atualizado acerca das condições de entrega na rede e de eventuais mudanças na mesma. Para isso, foram escritas várias funções que, em conjunto, formam um algoritmo de *flooding*.

Primeiramente, temos a `send_message()` que, recebendo a mensagem difundida e o nodo para o qual a mesma será direcionada, cria um novo *socket* e o vincula ao endereço IP e à porta extraídos do nodo passado. Em seguida, serializa a mensagem e envia-a através do *socket* antes de o fechar. Esta função é chamada na função `send_message()`, que itera sobre a lista de portas dos vizinhos do *bootstrapper*, aplicando-a a todos os elementos dessa lista. Já a função `refresh_message()` atualiza a variável "tempo"e a variável "last_refresh"da mensagem difundida com a hora atual.

Entrentanto, a `refresh()` chama a função `flood()` para enviar as mensagens iniciais de *flooding* e depois entra num *loop* infinito onde executa o processo de flooding periodicamente de 30 em 30 segundos, utilizando a `refresh_message()` para atualizar os dados da mensagem que será reenviada.

Finalmente, a função `receive_message()`, executada quando um nó recebe uma mensagem a partir de outro nó, atualiza a mensagem com o delta de tempo entre o envio e a receção da mensagem e, logo depois, verifica se a mensagem atingiu o número máximo de saltos. Caso contrário, é modificada a lista de servidores mais próximos, transmitindo a mensagem para todos os vizinhos do nó. A função `listen_to()` escuta as conexões de entrada numa determinada porta e passa todos os dados recebidos à `receive_message()` para que esta os processe. A função `listening()` inicia uma nova *thread* para cada nó na rede com a finalidade de escutar as conexões de entrada.

### 4.4 Construção de rotas para a entrega de dados

Com recurso ao processo de *flooding* explicado acima, foi possível construir as rotas para a entrega de dados que seria, posteriormente, necessário determinar a rota através da qual a *stream* solicitada será entregue. Como tal, cada nó da *overlay* deve considerar como métrica mais favorável o menor atraso, e para atrasos iguais, o menor número de saltos. Tendo em conta esta métrica, é descoberto, para cada nodo, o *Big Node* ou Servidor mais perto de si; isto porque, após ser um pedido de *streaming*, quem estará encarregado de escutar e responder a esse pedido será precisamente o Big Node ou Servidor mais próximo. No entanto, em primeiro lugar, é verificado se esse nó tem o vídeo solicitado porque, na situação de não ter, este terá de efetuar o pedido em questão ao servidor.

## 5 Testes e Resultados

Como cenário de teste, construímos a rede overlay ilustrada na Figura 2, onde temos um Servidor, 2 nodos intermediários (C2 e C5) e 7 nodos simples (C1, C3, C4, C6, C7, C8, C9).

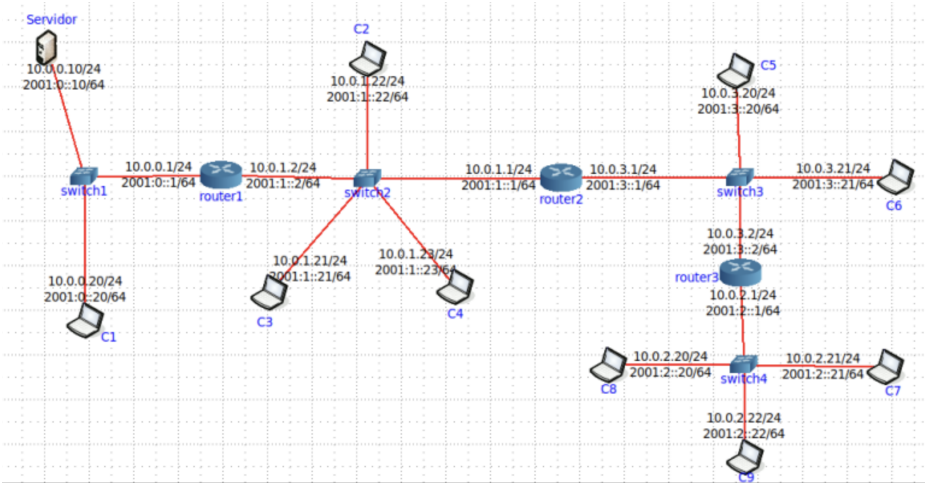


**Figura 2.** Cenário de teste: Rede overlay constituída por 9 nós

Na Figura 3, podemos confirmar que alguns nodos requerem a conexão à rede *overlay* ao indicar o seu *hostname*. Isto provoca a iniciação do processo de *flooding*, no qual são trocadas mensagens com os seus vizinhos de modo a ter conhecimento do estado da rede *overlay*, ou seja, que nodos vizinhos é que se encontram ativos, e também do Big Node (ou Servidor) mais próximo de si. A partir do momento que este é conhecido, começar-se-á uma conexão com ele para que o serviço de *streaming* possa acontecer.

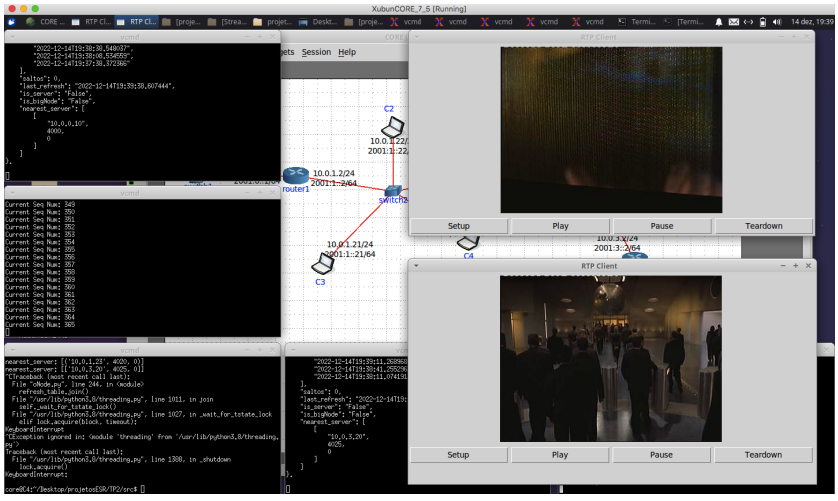


**Figura 3.** Exemplo de um resultado bem sucedido

Analisando, por outro lado, a Figura 4, podemos verificar uma situação de erro, em que o *Big Node* mais próximo do nodo em questão não possui o vídeo cujo *streaming* foi requerido.

De maneira a contornar este problema, definimos que, na situação do *Big Node* mais perto não conter o tal ficheiro de multimédia, a solução seria passar a responsabilidade ao segundo *Big Node* mais próximo, e assim sucessivamente, até que o serviço de *streaming* seja cumprido.

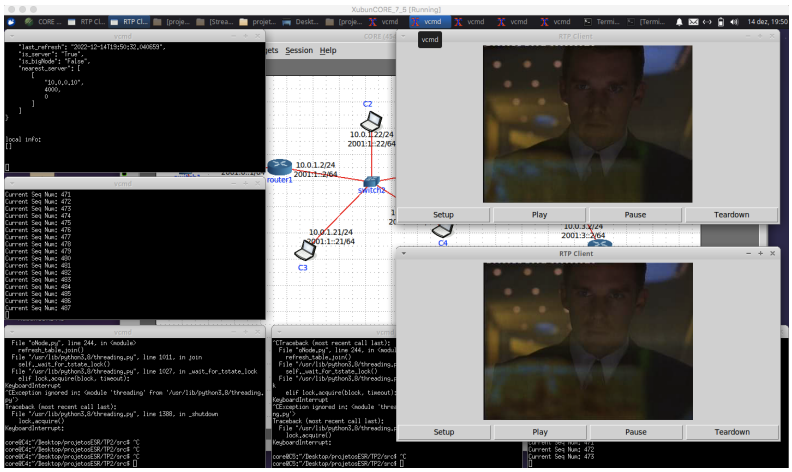


**Figura 4.** Outro exemplo de um resultado bem sucedido

## 6 Conclusões e Trabalho Futuro

Com a realização deste projeto, foi possível consolidar a matéria lecionada nas aulas teóricas relativamente aos protocolos e serviços de *streaming* de vídeo.

A equipa de trabalho sentiu algumas dificuldades no decorrer do projeto, nomeadamente na construção das rotas para o envio dos dados. Tendo sido solicitado uma rede *overlay* dinâmica, em que os nodos se podem conectar e desconectar da mesma a qualquer momento, a solução final tinha de garantir que uma próxima rota seria encontrada de modo a que a entrega pudesse ser feita com sucesso independentemente das adversidades. No entanto, apesar de ter enfrentado algumas dificuldades, o grupo acredita que concluiu um bom trabalho, respeitando os requisitos mínimos propostos.

Adicionalmente, como trabalho futuro, e visto que o protocolo de transporte utilizado é UDP, caracterizado por ser não fiável, o grupo desejaria implementar um mecanismo de controlo de perdas, bem como um ou outro requisito extra apontado no enunciado.