Serviço Over the Top para entrega de multimédia

José Manuel Antunes de Carvalho - PG53975, José Miguel Ferreira Barbosa - PG52689, and Miguel Filipe Cidade da Silva - PG54097

Universidade do Minho
Campus Gualtar, Braga, Portugal, 4710-057
https://www.uminho.pt/PT

Resumo No âmbito da unidade curricular de Engenharia de Serviços em Rede do primeiro ano do Mestrado em Engenharia Informática foi proposta a construção de um serviço *Over the Top* para a entrega de multimédia, ou seja uma aplicação de *Streaming* que utiliza a infraestrutura de rede existente simulada no emulador *CORE*.
Neste documento iremos descrever as nossas decisões e implementação deste serviço que desenvolvemos em Java, tendo em atenção a arquitetura do nosso sistema, os protocolos criados, envio de dados e das redes *Underlay* e *Overlay* que construímos.

Keywords: Streaming · Redes · Protocolos

1 Introdução

Nesta fase do trabalho prático referente à UC de Engenharia de Serviços em Rede, fomos desafiados a conceber e prototipar um serviço *Over The Top*, que promova a eficiência e a otimização de recursos para melhor qualidade de experiência do utilizador. Usamos os nossos conhecimentos obtidos nesta cadeira e também na licenciatura para desenhar e construir um protótipo de entrega de áudio/vídeo, a partir de um servidor de conteúdos para um conjunto de clientes.

Para tal nós fomos confrontados com vários desafios e objetivos a cumprir e a atingir. Um exemplo importante de um dos nossos objetivos foi a otimização da banda larga com pretexto de entregar os conteúdos pretendidos com a maior eficiência possível. Algo importante também de frisar é que optamos por utilizar ambos os protocolos de transporte (*UDP* e *TCP*), em contextos específicos, que iremos explicar mais adiante.

Este relatório está dividido em várias secções, onde iremos explicar detalhadamente as decisões que tomamos durante a conceção do nosso serviço. Começaremos por mostrar e explicar a nossa topologia e a arquitetura da solução, mostrando e explicando as classes que criamos e definimos. A seguir explicaremos cada um dos protocolos e a troca de mensagens. Após isso, iremos detalhar a nossa implementação, o código e as bibliotecas que utilizamos para realização do trabalho prático. Depois explicamos as limitações possuídas pela nossa solução e o porquê

destas e demonstrar os testes realizados e os resultados obtidos. Para terminar temos as conclusões que retiramos e perspetivas para trabalho futuro.

É importante também salientar que este serviço segue uma metodologia *opt-in*, ou seja, os utilizadores devem expressar claramente a sua intenção de aceder ao conteúdo para o adquirirem. Esta metodologia é o contrário da *opt-out* na qual os utilizadores automaticamente tem acesso aos conteúdos. Portanto, os utilizadores apenas visualizam os recursos que desejarem, estando livres para os aceder conforme desejarem.

2 Especificação da rede

O primeiro passo no processo de desenvolvimento foi construir as redes que suportariam o nosso serviço. Para tal, através do emulador CORE montamos a topologia presente na figura 1. Esta topologia tem a nomenclatura dos clientes e servidores baseada em freguesias da cidade de Braga, enquanto que os *switches* tem nomes alusivos a consolas da *Nintendo*.

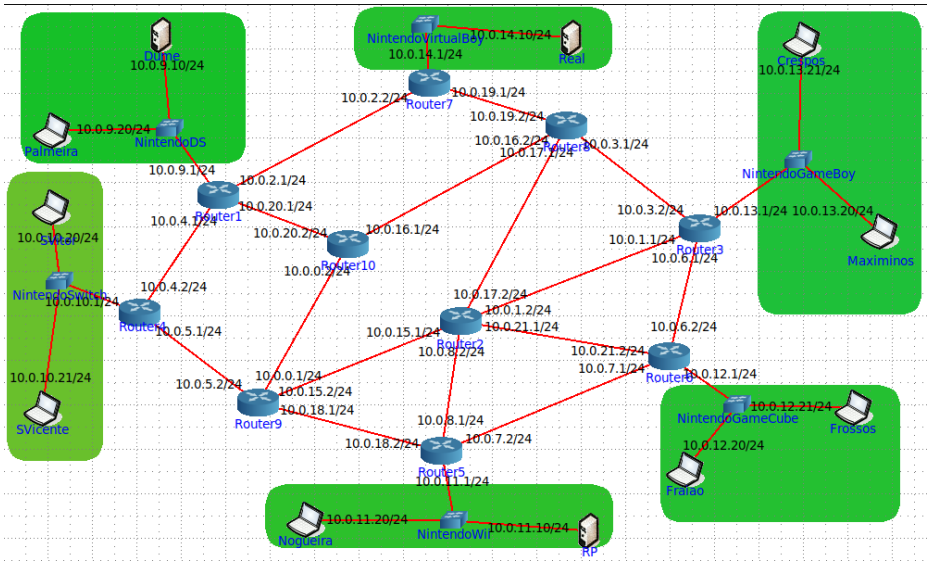

Figura 1. Topologia da nossa rede *Underlay*

Através da rede *Underlay* montada surge após a remoção de dois nodos a nossa rede *Overlay* que consta na figura 2.

Como se consegue entender na figura, as ligações são bidirecionais. Os clientes conectam-se nos mesmos routeadores da rede *Underlay*.

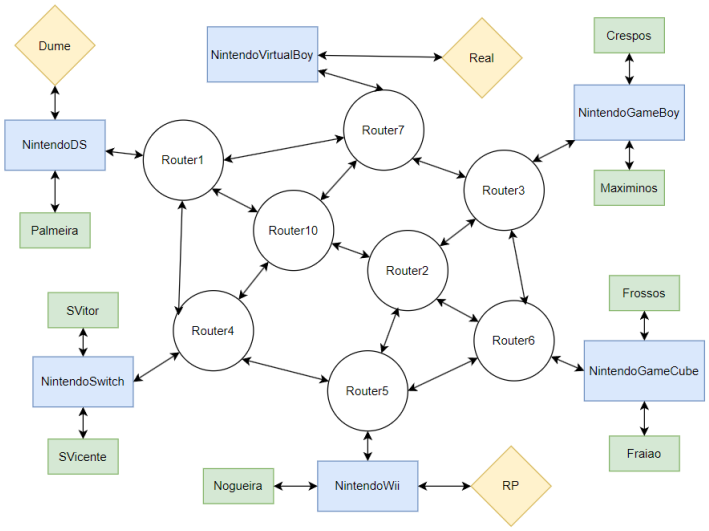

Figura 2. Topologia da nossa rede *Overlay*

3 Arquitetura da solução

Nesta secção vamos mostrar a arquitetura da nossa solução, relativamente às classes que implementamos. Não vamos entrar em muito detalhe nesta fase, pois à frente do relatório explicaremos as coisas em maior detalhe. Na figura 3, temos a representação da nossa arquitetura. Para facilitar a visualização da mesma, dividimos os diferentes tipos por cores. As quatro classes mais importantes, relativamente ao funcionamento da solução, estão pintadas por azul, sendo que estas estão ligadas entre si, e aos restantes tipos. As duas classes desenhadas a pensar na visualização do vídeo, de cor de rosa, estão ligadas entre si e à classe **Client**. Os protocolos, que estão a verde, estão ligadas às classes nas quais são aplicadas. E por último temos os executáveis, que estão representados a roxo e apenas estão conectados à própria classe. No que toca aos protocolos, na figura 4 que está na secção seguinte, exploramos com mais detalhe cada um destes. Mas fazemos um pequeno link nesta secção para facilitar a visualização das ligações entre estes e as classes:

- ProtocolEndStreaming - representa o protocolo para terminar o Streaming que faz ligação com as classes **ServerNode** e **ServerDB**
- ProtocolLoadContent - está relacionado com os pacotes de *streaming*, que faz ligação com as classes **ServerRP** e **ServerDB**.
- ProtocolTransferContent - está relacionado com os pacotes de *streaming*, que faz ligação com as classes **Client** e **ServerNode**.
- ProtocolStartStreaming - representa o protocolo para iniciar o *streaming*, que faz ligação com as classes **Client** e **ServerNode**.

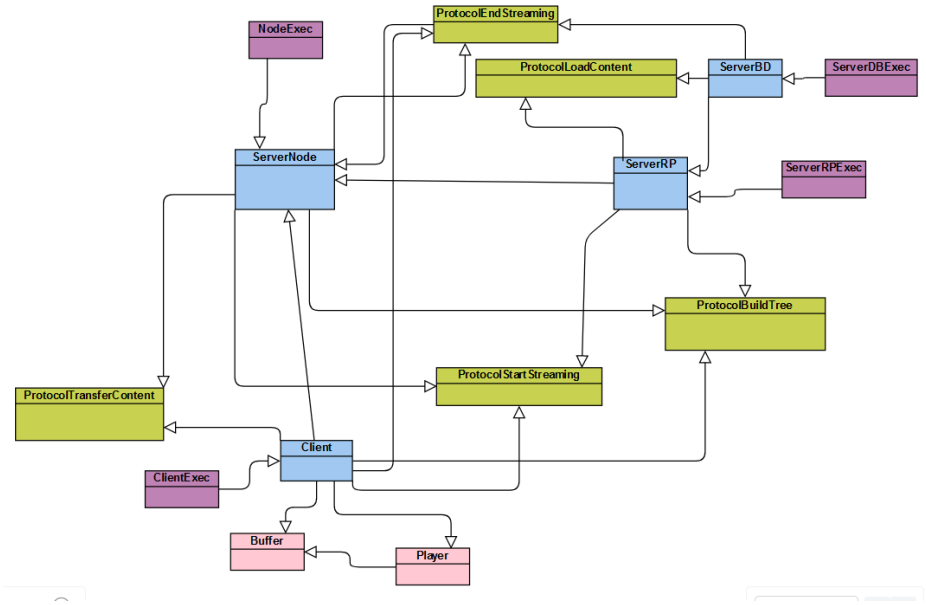

Figura 3. Arquitetura da nossa solução

– ProtocolBuildTree - representa os protocolos referentes à construção da árvore, que faz ligação com as classes **Client**, o **ServerNode** e o **ServerRP**.

4 Especificação do(s) protocolo(s)

A figura 4, ilustra os protocolos que definimos para o nosso trabalho e vai ser o nosso guia visual nesta secção. Como podemos observar na imagem, construimos cinco protocolos diferentes.

Os dois primeiros que vamos falar são aqueles que referem à construção da árvore de rede, sendo que temos um para a pergunta e um para a resposta.

Relativamente ao protocolo de construção da árvore, este protocolo tem a fase da pergunta e fase da resposta, com dois objetivos principais, saber quais são os vizinhos que conseguem transmitir um certo conteúdo e saber quais destes vizinhos os mais rápidos.

O **Protocolo de Pergunta** desempenha um papel fundamental na fase inicial da construção da árvore. A sua principal responsabilidade é que cada nodo pergunta aos seus vizinhos se eles estão a transmitir um determinado recurso.

O **Protocolo de Resposta** desempenha o papel complementar da pergunta, que basicamente contém apenas um código.

Os outros três protocolos foram desenvolvidos para gerir a transmissão de dados, estando os três relacionados ao *streaming*.

Um dos protocolos refere ao pacote de início do *streaming* outro é o protocolo dos pacotes de *streaming* e o último é o protocolo para terminar o *streaming*.

O **Protocolo do Pacote de Início do *Streaming*** é o protocolo responsável por iniciar o processo de *streaming*. A sua função principal é estabelecer as ligações entre nodos para o *streaming*, sendo esta feita com base no protocolo de construção de árvore. Este protocolo apenas contém o recurso que será feita a inicialização do *streaming*.

O **Protocolo dos Pacotes de *Streaming*** simboliza o conteúdo da transmissão em si.

Por último, o **Protocolo para terminar Streaming** serve única e exclusivamente para quando um nodo quer parar de receber a transmissão de um certo recurso. No pacote vai encapsulado o *IP* do nodo que quer parar a transmissão e o recurso associado à transmissão.

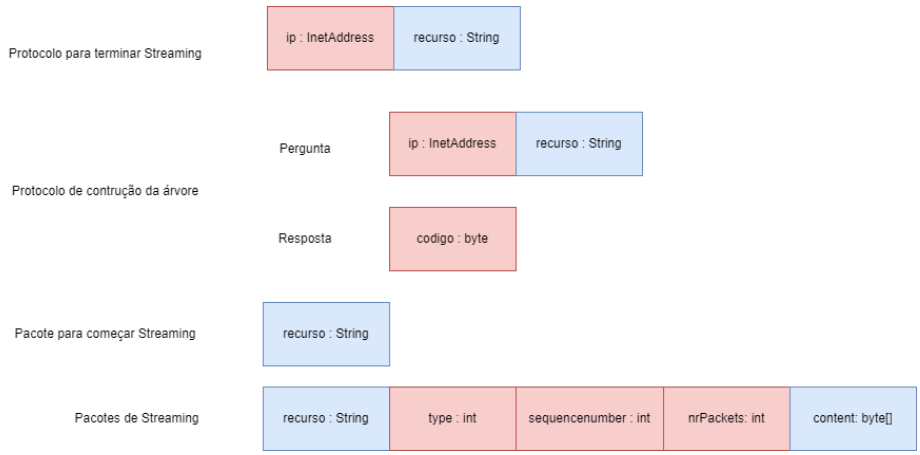

Figura 4. Protocolos referentes a 'Streaming' e à conexão

4.1 Formato das mensagens protocolares

Cada um dos pacotes que implementamos tem maneiras diferentes de comunicar e enviar o seu conteúdo a outras partes do sistema.

No **Protocolo de construção da árvore**, os dois protocolos têm um formato diferente nas mensagens protocolares.

Este protocolo contém um endereço IP, que é o endereço IP do cliente que está a requisitar o recurso e o recurso que o cliente pretende receber. Criamos esta associação entre IP cliente e recurso para evitar situações cíclicas entre nodos da rede.

O de pergunta possui dois componentes, um sob forma *InetAddress*, que representa o endereço IP do cliente que está a requisitar o recurso e outro em formato *String*, que representa o recurso que procuramos e tencionamos transmitir.

O de resposta possui apenas um componente sob formato byte com o código enviado pelo nodo, que contém a sua resposta consoante o pedido que recebeu. Os códigos e seus significados encontram-se na tabela 1.

Tabela 1. Tabela com os significados dos códigos do protocolo de construção de árvore

Código	Significado
0	Loop
1	Found
2	NoFound

No **Pacote para começar o *Streaming***, a mensagem também possui apenas um componente, que se encontra no formato *String*, que representa o recurso que iremos transmitir. O **Protocolo de Streaming**, é aquele que é mais complexo e possui mais componentes. Os campos deste protocolo e os seus significados estão explicados na tabela 2. Os valores e significados do campo *type*, encontram-se na tabela 3.

Tabela 2. Tabela com os significados dos campos do protocolo streaming

Campo	Significado
recurso	Recurso associado à transmissão do pacote
type	Tipo do pacote
sequenceNumber	Número de sequência do pacote
nrPackets	Número do pacote associado ao pacote *sequenceNumber*
content	Conteúdo do pacote

Tabela 3. Tabela com os significados dos códigos do campo *type* do protocolo de *streaming*

Código	Significado
0	codAudio
1	codVideo
2	codEndStream
3	codNoExist

4.2 Interações

Neste projeto, nós temos várias interações, sendo que decidimos usar *TCP* e *UDP* para diferentes interações entre nodos. O *UDP*, foi usado apenas na transmissão de um recurso, ou seja, na parte do *multicasting* em si. Utilizamos o *UDP* para este caso, uma vez que é mais rápido apesar de ser menos confiável em comparação com o *TCP*. Para as restantes situações, nós utilizamos o *TCP*, por ser muito mais seguro de utilizar, uma vez que assegura, que os pacotes são sempre entregues no destino.

No que toca, à comunicação entre protocolos, começamos por falar da árvore. Se um determinado nodo recebe uma pergunta do protocolo de construção de árvore, a primeira coisa que este irá fazer é verificar se está a transmitir o conteúdo. Caso esteja, então responde automaticamente com uma resposta com código 1. Se não estiver, vai verificar se no passado recente já recebeu aquele pedido de recurso. Caso tenha recebido, então responde automaticamente com uma reposta com o código 0. Caso nenhuma das situações anteriores se verifique, então o nodo irá fazer um pedido de pergunta, para os nodos vizinhos. Após receber as respostas de todos os vizinhos, o nodo responde ao nodo que lhe questionou uma resposta com código 1 se algum vizinho encontrou e 2 caso nenhum vizinho tenha encontrado.

Na parte de começar o *streaming*, simplesmente o nodo tenta estabelecer uma ligação *TCP* com o melhor vizinho e manda uma mensagem utilizando o protocolo *Start Streaming* e o *socket TCP* é fechado. Se o melhor vizinho estiver indisponível, o nodo tenta contactar o segundo, e também estiver indisponível, o terceiro e assim sucessivamente até chegar ao pior vizinho. Se todos os vizinhos estiverem indisponíveis, então o *streaming* não começa. Relativamente ao *streaming* em si, um socket *UDP* é aberto nos nodos, que ficaram à espera de mensagens de streaming. Quando recebem uma, eles desencapsulam a mensagem, retiram o recurso, verificam quais os nodos vizinhos que querem aquele recurso e manda para todos esses vizinhos o pacote encapsulado novamente. Relativamente ao *end streaming*, quando um nodo recebe essa mensagem ele irá remover o vizinho associado que já não pretende a transmissão de um determinado recurso. Se ele verificar que já não existe nenhum vizinho que pretenda aquele recurso, então ele vai enviar uma mensagem o nodo que lhe está a transmitir o recurso a informar que ele próprio já não quer receber transmissão daquele recurso.

A figura 5 ilustra as etapas deste processo.

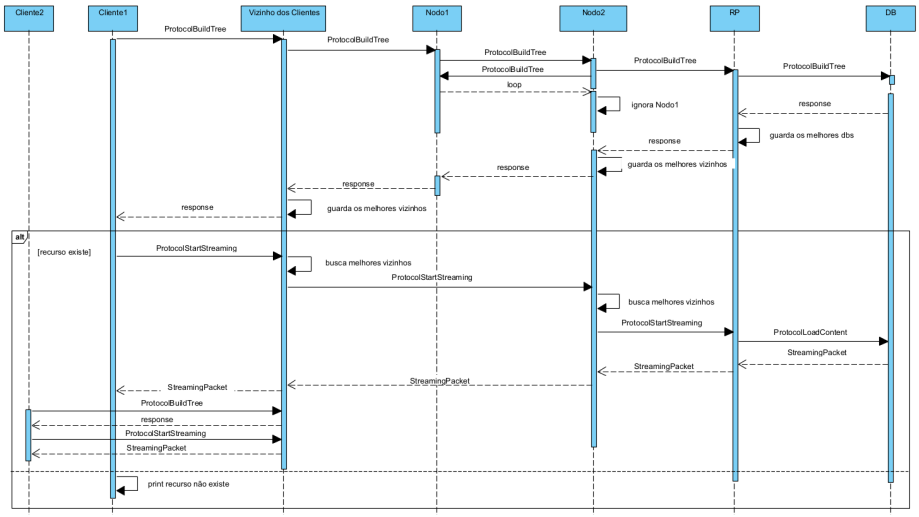

Figura 5. Diagrama de Sequência que demonstra a troca de mensagens entre as componentes do sistema.

5 Implementação

Nesta secção iremos abordar pormenores relativos à implementação do nosso sistema. Iremos começar por explicar em detalhe o que cada classe criada faz, enunciando os métodos mais importantes de cada uma. Apresentaremos também os parâmetros necessários para correr o código também listaremos algumas bibliotecas que usamos.

5.1 Detalhes

Primeiro, vale a pena explicar que na raiz da pasta do trabalho temos duas diretorias. Uma contem os nossos ficheiros de configuração e outra com o código do projeto.

Começando com os ficheiros de configuração, estes são ficheiros de texto bastante simples e temos um ficheiro de configuração para cada nodo. O ficheiro de configuração dos servidores com o conteúdo, cada linha corresponde ao caminho até um ficheiro correspondente a um recurso. Os ficheiros dos nodos contam com os endereços IP dos vizinhos.

Já na diretoria *src/* podemos encontrar o código fonte do projeto, sendo que na raiz da diretoria tem o ficheiro com a nossa topologia.

Explicando primeiro a diretoria onde temos as classes relativas aos nossos protocolos. Aqui temos duas subdiretoras, *Helper/* e *Utils/* e quatro módulos nos quais declaramos o código relativo aos nossos protocolos, nomeadamente métodos que permitem encapsular e desencapsular informação transmitida por estes pacotes. A diretoria *Utils/* apenas apresenta a classe Manager que nada

mais é do que uma classe que implementa métodos que imprimem mensagens de sucesso alusivas à comunicação. Por sua vez a diretoria *Helper/* conta com:

- HelperConnection - um *record* que ajuda a representar mensagens de certos protocolos no sistema, agrupando o nome de um recurso com IP.
- HelperContentReader - classe cuja função é hospedar métodos que auxiliem a ler o conteúdo encapsulado num pacote.
- HelperContentWriter - semelhante à HelperContentReader mas com a função contrária.
- HelperProtocols - classe que conta com métodos que auxiliam a escrita e leitura de conteúdo para pacotes TCP e UDP.

Focando agora na diretoria *Nodes/*, aqui contamos com três subdiretoras. A primeira que vamos explicar é a *Utils/* que conta com classes utilitárias tal como na diretoria anterior, no entanto aqui temos mais classes, sendo estas:

- Cods - Classe que apenas conta com códigos que utilizamos ao longo do projeto como o número das portas, ou códigos que referenciam o tipo de dados ou para gerir a comunicação ou o tamanho dos pacotes.
- Debug - Classe com métodos auxiliares para o processo de Debug, apresentando informação no ecrã ou mensagens de erro.
- OrganizeIps - Classe auxiliar que ajuda no processo de estabelecimento de comunicação com vizinhos com métodos que organizam os IPs dos vizinhos com base no tempo de resposta. Esta classe apenas é usado num teste.
- TaskExecutor - Classe auxiliar com métodos úteis para correr e interromper tarefas em paralelo.
- VideoExtractor - Classe que extrai tanto as imagens (frames) como o áudio que compõe um vídeo.

Na pasta *Execs/* constam os nossos executáveis, iremos falar deles em detalhe na subsecção seguinte devido à importância dos parâmetros destas classes.

Por fim a diretoria *Classes* conta com as classes mais importantes do código, estas são:

- Buffer - Classe que recebe os pacotes de *streaming* e os organiza de modo a apresentar de forma correta o conteúdo no lado do cliente.
- Client - Classe que implementa os métodos que permitem o funcionamento do clientes, pedindo e acedendo a recursos.
- Player - Classe que implementa a interface do cliente através de Swing. Utiliza um JFrame, JButtons, um JPanel e uma JLabel de modo a montar a janela na qual o cliente visualiza o recurso obtido.
- ServerDB - Classe representativa dos servidores que hospedam a base de dados, conta com métodos para validação de recursos e caso o servidor tenha o recurso enviar áudio e vídeo do mesmo. Também conta com métodos para processar a comunicação com o RP.
- ServerNode - Classe que representa os nodos da topologia, contem informação relativa a quais são os vizinhos do nodo, e quais são os recursos que o mesmo está a transmitir. Também conta com informação relativa aos vizinhos cuja

resposta é mais rápida. Para além disso, os locks também são utilizados para assegurar o sucesso da concorrência. A nível de métodos conta com métodos para adicionar e remover recursos, clientes e vizinhos, tal como métodos para aceder ao melhor vizinho, para enviar e receber pedidos, encapsulando dados e para iniciar *multicast*, havendo também métodos para iniciar o nodo e para inicio de *streaming*, estabelecendo a comunicação.
- ServerRP - Subclasse de ServerNode que implementa o RP, conta com os mesmos métodos de ServerNode tendo também métodos para contactar o servidor que hospeda os recursos e para adicionar o recursos no RP. Aqui também temos uma versão diferente do método *startStreaming(Socket socket)*.
- StreamingPacket - Classe auxiliar para os protocolos de transmissão.

No que toca a paralelismo, quando iniciamos um *ServerNode*, ou seja, o RP ou um nodo intermédio da rede, são inicializadas 5 *threads*, sendo que cada uma é responsável por um determinado serviço. Estes serviços são:

- Serviço de *multicast*
- Serviço de começar *Streaming*
- Serviço de terminar *Streaming*
- Serviço de construir a árvore
- Serviço de gestão de recursos

Para além destas *threads*, como nos 4 primeiros serviços o nodo atua como um servidor, é criada uma nova *thread* sempre que é recebido um novo pedido, sendo que esta será por processar e responder esse pedido.

Estas *threads* possibilitam que o nodo esteja aberto para diferentes serviços paralelamente e também fazem com que o serviço esteja otimizado.

5.2 Parâmetros

O nosso sistema apresenta vários parâmetros a ter em consideração, seja para executar o código como em algumas funções.

Relativamente aos executáveis, temos quatro cada um tem de ser invocado com argumentos diferentes:

- Client, que é o executável a correr na consola do cliente, tem como argumentos o IP do cliente, o IP do vizinho ao qual se conecta e o recurso a aceder.
- NodeExec, executável a correr nos nodos da rede, tem como argumento um ficheiro de configuração onde constam os endereços de IP dos nodos vizinhos.
- ServerDBExec, que corre no servidor com a base de dados, tem como argumento o ficheiro de configuração da base de dados.
- ServerRPExec, que executa no RP, assim como outros, recebe o ficheiro de configuração do RP.

Os executáveis NodeExec, ServerDBExec e ServerRPExec podem ter um segundo argumento que é um valor numérico que serve para realização de testes.

Já a nível de classes do código não iremos explicitar os parâmetros necessários para evocar todas, mas gostaríamos de destacar que para criar os nossos *PDUs*, eles necessitam dos parâmetros especificados na secção 4.1 para serem criados.

5.3 Bibliotecas de funções

No que diz respeito a bibliotecas e a classes que importamos para o projeto, dado que recorremos a Java optamos pelas seguintes:

- InetAddress e InetSocketAddress: Implementam endereços IP, a primeira sem a porta definida e a segunda com a porta definida
- Socket.
- Java Swing: utilizada para a interface.
- DatagramPacket: Implementa datagramas para ligações não orientadas à conexão (UDP).
- DataInputStream e DataOutputStream: encapsulamento e desencapsulamento de dados de forma segura para várias threads, neste trabalho usamos apenas para escrever e ler conteudo em pacotes TCP.
- ReadWriteLock e ReentrantReadWriteLock: para definir as secções críticas.
- Classes básicas do Java para gestão de ficheiros, de conjuntos de elementos e de erros.

6 Limitações da Solução

As principais limitações da nossa solução são:

- *Multicast* dinâmico
- Pacotes perdidos
- Vídeos pesados

Nas próximas secções iremos abordar cada uma das limitações, indicando eventuais problemas que elas possam trazer para o nosso serviço, bem como possíveis soluções.

6.1 *Multicast* dinâmico

Um dos extras que está contido no enunciado é para fazer o *multicast* dinâmico.

A nossa implementação tem por base o *multicast* estático, ou seja, depois de definida a árvore de *multicast*, as ligações entre os nodos da rede *Overlay* não mudam ao longo do tempo. Esta solução, embora seja a mais simples, trás diversos problemas e limitações.

Um dos principais problemas está relacionado com a latência dos nodos. Se um determinado nodo começa a ter demasiados vizinhos com os quais tem que

fazer transmissão de um certo recurso, poderá eventualmente começar a atrasar o envio dos pacotes.

Outro aspeto, e talvez o mais critico, está relacionado quando um nodo da árvore fica inativo, seja qual for motivo. Esta situação é trágica na nossa implementação e afeta por completo o serviço de *streaming* para todos os nodos que dependiam desse mesmo nodo. Isto resulta numa árvore fica desconectada.

Uma das soluções para resolver este problema de *multicast* dinâmico era implementamos um novo protocolo que verificasse o estado dos nodos na rede. Este protocolo seria algo semelhante ao protocolo *RTCP*.

6.2 Pacotes perdidos

Implementar um mecanismo de deteção e recuperação de pacotes é outro extra do projeto, ao que acabamos por não implementar. Desenvolver o sistema adaptado para detetar e recuperar pacotes perdidos é algo extremamente necessário no mundo real. Pacotes perdidos na rede pode fazer com que certos *frames* ou pedaços de áudio sejam recebidos de forma incompleta no lado do cliente.

Uma das formas de resolver este problema, era utilizarmos o protocolo *TCP* em vez do *UDP*, pois o protocolo *TCP* garante a receção dos pacotes. Se fosse para permanecer o *UDP*, poderíamos criar um mecanismo que um certo nodo ou o cliente detetasse a perda de um pacote, e quando fosse detetado essa perda, teríamos que criar um novo protocolo para que algum nodo superior retransmitisse o pacote pedido.

6.3 Videos pesados

Graças aos testes realizados, conseguimos verificar que a nossa solução funciona direito para vídeos curtos e leves, mas para vídeos de grande dimensão a nossa solução apresenta certos erros.

Quando referimos vídeos de grande dimensão, não estamos a referir que o problema seja com o tempo do vídeo, mas sim por exemplo, cada *frame* tiver um grande número de bytes e quando temos um áudio à mistura.

Estes erros ocorrem no cliente e nos nodos intermédios da rede e acontecem sobretudo pelo facto de os pacotes serem recebidos fora de ordem.

Como já foi mencionado, nós temos um pacote de *streaming* com o tipo *EndStream*, que serve para limpar a árvore *multicast*, aquando a transmissão de um certo conteúdo termina. Por vezes os nodos intermédios e o cliente recebem esse pacote primeiro antes de receber todos os pacotes do conteúdo.

Uma das formas de resolver este problema, era utilizarmos o protocolo *TCP* em vez do *UDP*, pois o protocolo *TCP* garante a ordem de chegada dos pacotes.

Outro aspeto que poderá causar este erro, tem a ver com o próprio emulador core, visto que a nossa solução utiliza muitas *threads* para o serviço como um todo, acaba por torna-se pesado simular um ambiente de *streaming* numa maquina virtual paraa transmissão de conteúdo com muitos bytes.

7 Testes e resultados

Os teste realizados foram:

1. RP - DB pergunta de recurso inexistente
2. RP - DB pergunta de recurso existente
3. RP - DB pedido de transmissão de recurso inexistente
4. RP - DB pedido de transmissão de recurso existente
5. Testes RP - DB quando parte dos servidores DB estão desligados
6. Cliente pede transmissão de um recurso inexistente
7. Cliente pede transmissão de um recurso existente
8. Transmissão do mesmo recurso 2 Clientes da mesma sub-rede
9. Transmissão do mesmo recurso 2 Cliente de sub-redes diferentes
10. Transmissão do mesmo recurso 3 Cliente, 2 da mesma sub-rede e 1 de uma sub-rede diferente
11. Transmissão quando temos todos os nodos da rede ligados
12. Transmissão de recursos diferentes para 2 Clientes
13. Transmissão termina quando todos os Clientes da mesma saem
14. Transmissão continua enquanto houver Clientes

Os nodos que cada teste utiliza está representado na tabela 4. O objetivo de cada teste está indicado na tabela 5. Em todos os testes mencionados, obtivemos o resultado esperado, sendo que tal como indicado na secção 6.3, a nossa implementação não funciona para certos tipos de vídeos.

Tabela 4. Nodos utilizados para cada teste

Teste	*RP*	*Dume*	*Real*	*R1*	*R2*	*R3*	*R4*	*R5*	*R6*	*R7*	*R10*	*S Vitor*	*S Vicente*	*Maximinos*
1	✓	✓	✓											
2	✓	✓	✓											
3	✓	✓	✓											
4	✓	✓	✓											
5	✓	✓												
6	✓	✓					✓	✓				✓		
7	✓	✓					✓	✓				✓		
8	✓	✓					✓	✓				✓	✓	
9	✓	✓				✓	✓	✓	✓			✓	✓	✓
10	✓	✓				✓	✓	✓	✓			✓	✓	✓
11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
12	✓	✓					✓	✓				✓	✓	
13	✓	✓					✓	✓				✓	✓	
14	✓	✓					✓	✓				✓	✓	

Tabela 5. Objetivos de cada teste

Teste	Objetivo
1	Um servidor *DB* devolve código 2 para o *RP*
2	Um servidor *DB* devolve código 1 para o *RP*
3	Um servidor *DB* devolver código 2 para o *RP*
4	Um servidor *DB* devolver código 1 para o *RP*. Transmissão de pacotes entre o *DB* e o *RP*
5	Objetivos testes 1-4
6	Observar a troca de mensagens entre os nodos da rede *Overlay*. Cliente *S Vitor* é informado que o recurso não existe
7	Observar a troca de mensagens entre os nodos da rede. Cliente *S Vitor* é informado que o recurso existe e começa a receber o *streaming*.
8	Clientes *S Vitor* e *S Vicente* são informados que o recurso existe e ambos começam a receber o *streaming*.
9	Clientes *S Vitor* e *Maximinos* são informados que o recurso existe e ambos começam a receber o *streaming*.
10	Objetivos testes 8 e 9.
11	Objetivos testes 6,7,8,9 e 10
12	Observar o *streaming* dos diferentes recursos
13	Troca de pacotes de *streaming* param de navegar na rede.
14	Transmissão continua normalmente para o Cliente que não para o *streaming*

8 Conclusões e trabalho futuro

Com o término deste projeto conseguimos afirmar que foi um desafio interessante, pois conseguimos consolidar conhecimentos adquiridos ao longo da UC de forma intuitiva. Aprofundamos e aplicamos *Multicasting* e *Unicasting* em momentos distintos, o que nos permitiu compreender melhor a diferença entre estas duas formas de comunicação e montamos pela primeira vez uma rede *Overlay*.

Em retrospetiva, o uso do protocolo *TCP* acabava por resolver partes das nossas limitações, mas optamos por manter o uso do *UDP*, pois um serviço de *streaming* tem que ter em conta a número de pacotes trocados entre nodos de modo a otimizar a utilização largura de banda, garantindo o melhor serviço, com *TCP* isso seria um obstáculo pois teríamos mais pacotes na rede.

A nível de desafios, acreditamos na parte de planeamento e implementação do sistema não enfrentamos nada que fosse contra as nossas expectativas, acreditamos que a escolha de Java como linguagem e ambiente de desenvolvimento foi bastante acertada, pois facilitou a divisão de trabalho entre os elementos do grupo com a sua estrutura modulada em classes separadas. No entanto, a parte de testes já apresentou mais obstáculos, devido à complexidade dos testes realizados.

Estamos satisfeitos com o que desenvolvemos, pois conseguimos alcançar aquilo que eram os nossos objetivos neste projeto. No entanto, não implementamos os extras do enunciado por motivos de gestão do tempo.