Comunicações por Computador

Trabalho Prático Nº. 2 - Gateway Aplicacional e Balanceador de Carga sofisticado para HTTP

Eduardo Coelho, Henrique Neto, Júlio Alves
e-mail: {a89616,a89618,a89468}@alunos.uminho.pt

25 de maio de 2021

1 Introdução

Um gateway aplicacional pode ser utilizado por diversos motivos, entre os quais, uma maior segurança, um melhor desempenho e a flexibilidade de poder gerir um serviço em função de parâmetros que só estão disponíveis na camada de aplicação. O objetivo deste trabalho prático é, então, implementar um gateway de aplicação que opere exclusivamente com o protocolo HTTP/1.1 que seja capaz de responder a múltiplos pedidos em simultâneo, através de uma pool dinâmica de N servidores de alto desempenho, utilizando um protocolo a especificar para este efeito. Este gateway receberá então pedidos HTTP, dirigidos à porta 80, passando depois por solicitar um ou mais servidores disponíveis na pool de servidores. A comunicação com estes terá de ser feita através de um protocolo especificado para o efeito, que terá de funcionar sobre UDP e não poderá ser orientado à conexão e não poderá ter informação de estado. Os servidores servirão todos os mesmos ficheiros e poderão ser parados ou iniciados a qualquer instante, sem que o funcionamento do gateway seja afetado. O gateway poderá também, se assim entender, recusar pedidos dos clientes por questões de segurança.

2 Arquitetura da solução

2.1 FileServerGateway

FSGateway Esta classe constitui a superfície da aplicação do *FileServerGateway*, e desta forma é responsável por abrir os sockets e inicializar o servidor *HTTP* (ou *HTTPS*). Desta forma, esta classe é uma *Runnable*, que começa por abrir um socket *UDP* e inicializar um servidor *HTTP*. De seguida, instancia o runnable *ServerManager*, que é responsável pela gestão do socket *UDP* e toda a comunicação que o envolve.

FSGProperties Nesta classe controlamos as propriedades relativas ao FastServerGateway, tais como o MTU, a quantidade máxima de servidores, o tempo de tolerância do servidor, a quantidade máxima de pedidos, o tempo entre pings e também definimos as chaves públicas e privadas do protocolo *HTTPS*. Adicionalmente implementa uma *BlackList*, que conforme o ficheiro parametrizado, indica quais os servidores proibidos de interagir com o *gateway*. Todas estas propriedades podem ser alteradas pelo utilizador no ficheiro *fsg.cfg*.

HTTPExchanger Esta classe contém apenas algum métodos estáticos e independentes de contexto, usados para definir *Headers* e algumas formas de conteúdo genérico (*body)* para alguns pacotes *HTTP*.

Main Na classe *Main*, como seria de esperar, apenas instanciamos os objetos necessários para que o programa funcione, sendo neste caso criada a instância do *FSGateway* consoante os argumentos dados pelo utilizador ao inicializar o programa.

ServerManager Nesta classe temos os *maps* que contém a cache dos ficheiros, da cache dos servidores e dos servidores que estão atualmente disponíveis e uma fila com um buffer de cache. Através desta classe, como o nome indica, controlamos os servidores, servidores esses que são controlados através de diferentes diferentes workers, sendo eles o *Dispatcher*, *CacheUpdater*, *CacheCleaner* e *CyclicClock*. O funcionamento e a função destes workers será explicada posteriormente.

2.1.1 Workers

CacheCleaner Este worker é o responsável por limpar a cache e esse processo desenrola-se da seguinte forma. O CacheCleaner possui duas filas de pedidos para limpar a cache e estas duas filas têm cada uma funções diferentes, sendo que uma das filas contém todos os servidores que irão deixar de fornecer ficheiros e a outra fila é um par que contém informação de um ficheiro e um servidor e irá remover a entrada do ficheiro no servidor, ou seja, informará que o ficheiro não estará mais disponível no servidor indicado e irá limpar o registo desse ficheiro no mesmo. Este worker é um runnable que é instanciado no *ServerManager*.

CacheUpdater O worker *CacheUpdater* possui uma fila de chunks *FSCache* e, para cada chunk, irá inserir as entradas para esse ficheiro no servidor respetivo. O *CacheUpdater* também é um runnable instanciado no *ServerManager* e está sempre a correr.

CyclicClock Este worker não é nada mais do que um relógio lógico cíclico, cujo ciclo lógico é entre os 0 e os 255 e executará uma tarefa de cada vez que o relógio avança. Este relógio é instanciado num timer do *ServerManager*, timer esse que é uma thread.

Dispatcher O worker seguinte é o worker responsável por receber todas as tramas UDP. O Dispatcher possui três consumidores e um predicado. O predicado indica se o servidor de onde a trama veio está autenticado. Relativamente aos consumidores, existe um consumidor global, um consumidor do servidor e um consumidor para o caso de o predicado não ser verdadeiro. O consumidor global de tramas irá receber todas as tramas que são destinadas ao gateway em si e não para uma instância do servidor. O consumidor do servidor irá receber todas as tramas que são destinadas a uma instância do servidor que está dentro do gateway. O consumidor *Unlogged* irá receber as tramas dos servidores que não estão autenticados. Este worker irá estar constantemente a ler e a fazer parsing das tramas que recebe. Caso o parsing falhe, a trama é discartada, caso contrário, as condições dos consumers serão testadas. Após serem testadas, teremos de ver se a trama é acknowledged. Este worker é instanciado numa thread do *ServerManager* e é um worker crítico no programa, na medida em que o seu funcionamento é completamente essencial para o bom funcionamento do mesmo.

ServerHandler O *ServerHandler* é um runnable que é instanciado de cada vez que um servidor se autentica com sucesso. Este worker instancia a sua própria thread e possui um sistema de tickets Tem também informação sobre o instante do último ping que foi recebido, o endereço do servidor a que corresponde, o socket de saída e uma queue dos pedidos de ficheiros. Quando um servidor se autentica, é criada uma instância do *ServerHandler* e para essa instância também é criada uma thread que fica instanciada no mesmo. Esta thread está sempre à escuta e a ler a queue de pedidos e responderá sempre aos mesmos por ordem de chegada. Esses pedidos chegam sempre que o servidor recebe um pedido *GET*, que invocará o método *GET* do *ServerHandler* e esse método vai fazer o pedido de *FileRequest* e inseri-lo na fila de pedidos do servidor. Após isso, retira um ticket e fica à espera da sua vez. Quando chega a sua vez, o seu pedido é processado e verifica se este teve êxito. Se não tiver êxito, é lançada uma exceção *FILENOTFOUND*. Quando o servidor recebe uma trama local pelo *ServerManager*, o *ServerHandler* vai invocar ou o método *receiveNotFound(FSChunk chunk)* ou o método *receiveOK(FSChunk chunk*, que irá fornecer ao pedido atual o pacote recebido. Quando o servidor morre, a thread instanciada é terminada e o *ServerManager* termina a instância do *ServerHandler*.

2.2 FastFileServer

FastFileServer Esta classe representa a aplicação do *FastFileServer*, sendo que aqui são inicializadas as sockets dos datagramas, o endereço do gateway, a pasta que pretendemos que o *FastFileServer* controle e a porta do servidor. Inicializamos também o *FFSReceiver* e o *Pinger*, que serão explicadas seguidamente.

FFSReceiver Esta classe é bastante simples, sendo que se limita a receber os datagramas e de os enviar com recurso ao *FastFileServer*. Por sua vez, ele separa os datagramas entre acknowledgements e não acknowledgements. As tramas de não acknowledgements são colocadas numa fila de espera. As acknowledgements são identificadas como sendo pings ou não. Caso sejam pings, são enviadas para o *Pinger*. Se não forem pings, são enviadas para o *FastFileServer* para ele poder fazer a gestão dos datagramas.

FFSForwarder A classe *FFSForwarder* é a responsável por encaminhar os pacotes para o gateway. Para esse encaminhamento é efetuado um controlo de janela semelhante ao TCP. Só há dois tipos de tramas que possuem acknowledgements, sendo elas as tramas de dados: data e cache. O controlo de congestão através da janela deslizante é efetuado através de temporizadores associados a um RTT esperado. Também é responsável por enviar as tramas de gestão e controlo, mas uma vez que estas não enviam acknowledgements, não são geridas pela janela.

Pinger A classe *Pinger* é um runnable responsável que faz parte de um mecanismo de gestão de conexão implementada entre o gateway e o servidor. Para tal, este envia periodicamente datagramas do tipo *Ping_Request* através de um *timer*, que permitem ao gateway confirmar a presença do servidor e em contrapartida, o servidor responde com um datagrama do tipo *Ping_Response* que o servidor fazer a mesma confirmação.

FFSProperties Assim como o *FSGProperties*, o *FFSProperties* é uma classe responsável por ler o ficheiro de configuração, neste caso o *ffs.cfg*, que contém propriedades importantíssimas para o bom funcionamento do programa, tais como o tamanho máximo da queue, o tempo de timeout dos Acknowledgements e o número máximo de tentativas de conexão permitidas. Todas estas propriedades podem ser alteradas pelo utilizador no ficheiro de configuração.

Main A *Main*, novamente, é classe que instancia os objetos necessários para o programa funcionar, sendo neste caso esse objeto o *FastFileServer*. Nesta classe são também instanciados os endereços do gateway e o endereço da máquina que está a correr o servidor, o path da diretoria da qual o servidor vai ter controlo e as portas do servidor e local, sendo que estas são predefinidas para a porta 80.

2.3 FileServerProtocol

De forma a criar o protocolo necessário para a realização deste trabalho prático, foi criado o package FileServerProtocol, que será constituído pelos packages Exceptions, FileServerChunk e Structs. O protocolo em si está definido no package FileServerChunk que, entre várias classes, tem como classe abstrata a classe FSChunk que será responsável pela criação dos datagramas. Através do método *DatagramPacket build()* receberá uma instância de um FSChunk, e com o auxílio do método *void write(ByteArrayOutputStream in)* que indica qual o tipo de FSChunk que o datagrama será, transformará essa instância num datagrama.

2.4 Utils

Neste package definimos duas estruturas utilitárias. O *BoundedQueue* que representa uma fila como um *BoundedBuffer*, ou seja uma fila onde existem várias *threads* produtoras e consumidores de conteúdo concorrentemente. O *Pair* representa um simples par de dados.

3 Especificação do protocolo

3.1 Formato das mensagens protocolares

Todos os tipos de datagramas seguirão a estrutura seguinte:

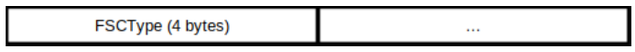


Figura 1: Estrutura geral de um FSChunk

Esta classe representa um datagrama geral. Assim, todos os datagramas que apresentaremos de seguida extendem esta classe

O datagrama poderá ser de diferentes tipos, sendo eles:

3.1.1 FSAnnouncement

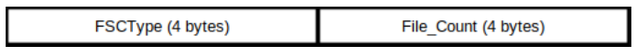


Figura 2: Estrutura de um datagrama FSAnnouncement

Para um FastFileServer se conectar ao gateway da nossa aplicação, tem primeiro que anunciar essa mesma intenção. Assim, criámos o datagrama FSAnnouncement de modo a qualquer FastFileServer poder-se conectar ao gateway. Para além do tipo no cabeçalho, este datagrama contém o nº de ficheiros existentes nesse servidor

3.1.2 FSAccept

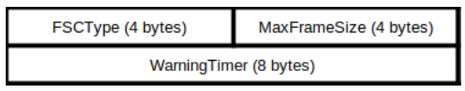


Figura 3: Estrutura de um datagrama FSAccept

Caso o gateway decida aceitar a tentativa de conexão de um dado FileFastServer que lhe tenha enviado um FSAnnouncement (se o gateway ainda não tiver o número máximo de servidores conectados), então pode responder-lhe com um datagrama FSAccept. Este datagrama, para além do tipo, contém a informação do tamanho máximo dos datagramas que o servidor lhe pode enviar (MaxFrameSize) e também o intervalo de tempo em que o servidor tem de comunicar o valor do seu relógio lógico (este relógio tem como objetivo controlar a inatividade dos servidores e será explicado em mais detalhe posteriormente).

3.1.3 FSCache

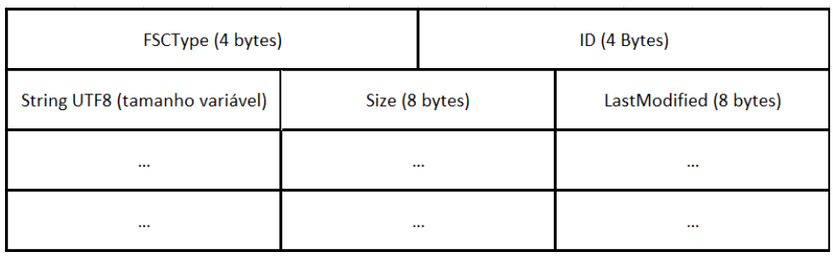


Figura 4: Estrutura de um datagrama FSCache

De modo a manter o mapeamento de ficheiros para servidores atualizado no gateway, os servidores podem enviar a informação dos seus ficheiros, enviando datagramas FSCache. Assim, este datagrama contém, para além do tipo e para cada ficheiro, o seu nome, a data da última modificação e o seu tamanho. Posteriormente será explicado em mais detalhe o funcionamento da cache na nossa aplicação.

3.1.4 FSGet

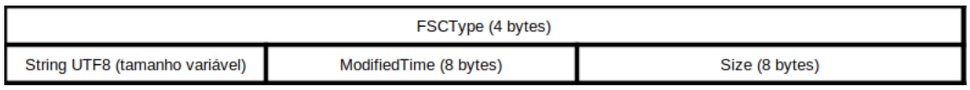


Figura 5: Estrutura de um datagrama FSGet

Ora, quando o gateway recebe um pedido HTTP GET de um host, tem de transferir esse ficheiro dos FastFileServers. Deste modo, pode utilizar o datagrama FSGet que, para além do tipo, contém toda a informação necessária para poder identificar unicamente o ficheiro em causa: nome, data de modificação e tamanho.

3.1.5 FSData

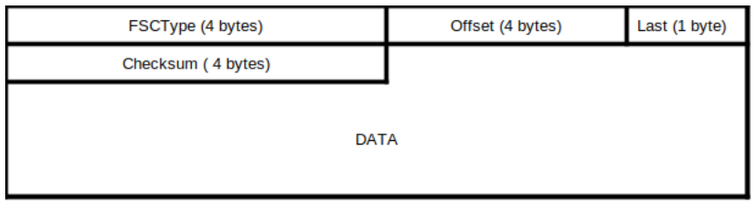


Figura 6: Estrutura de um datagrama FSData

Uma vez que o protocolo FastFileServer permite a transferência de ficheiros em blocos/chunks, precisamos de um datagrama que os represente. Com este propósito criámos o datagrama FSData que, para além do tipo, contém: um byte que indica se é o último chunk desse ficheiro, um campo checksum para deteção de erros, o offset e, por último, os dados relativos a este bloco do ficheiro. O modo como o servidor divide um ficheiro em chunks e envia para o gateway está explicado em mais detalhe posteriormente.

3.1.6 FSFail

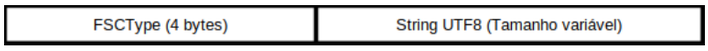


Figura 7: Estrutura de um datagrama FSFail

Quando um servidor se tenta conectar ao gateway, este pode recusar esta tentativa de conexão em casos especiais, como por exemplo, quando o gateway já está conectado ao número máximo de servidores permitido. Assim, quando o gateway decide que a conexão não pode ser estabelecida envia um datagrama FSFail para o servidor correspondente. Este datagrama contém uma descrição do motivo da conexão ter sido rejeitada, para além do tipo.

3.1.7 FSPing

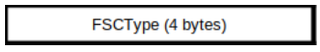


Figura 8: Estrutura de um datagrama FSPing

Para o gateway conseguir controlar os servidores que estão inativos, estes precisam de enviar pings num intervalo de tempo pré-determinado pelo FSAccept. Assim, este datagrama não necessita de nenhum campo para além do seu tipo para servir o seu propósito.

3.1.8 FSDataAck

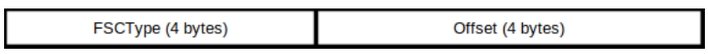


Figura 9: Estrutura de um datagrama FSDataAck

https://www.overleaf.com/project/6059b93e3cda2f48f97a7490 Sempre que o gateway recebe um datagrama FSData de um servidor, precisa de lhe responder com um Acknowledge, para o servidor saber que não será preciso retransmissão. Deste modo, criámos o datagrama FSDataAck onde, para além do tipo, contém o offset do datagrama FSData correspondente.

3.1.9 FSCacheAck

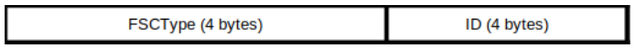


Figura 10: Estrutura de um datagrama FSCacheAck

Para além de confirmar a receção de chunks de ficheiros é importante garantir que o gateway tenha sempre a cache atualizada e completa. Com este propósito em mente, criámos o datagrama FSCacheAck que funciona como um Acknowledge próprio para a receção de datagramas FSCache. Para além do tipo, contém o ID do datagrama FSCache correspondente.

3.1.10 FSNotFound

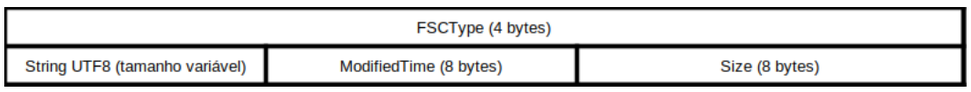


Figura 11: Estrutura de um datagrama FSNotFound

Se, após um pedido de transferência de um ficheiro a um servidor, este verificar que o ficheiro não existe, então é importante ele avisar o gateway, para podermos pedir o mesmo ficheiro a outro servidor que o tenha. Assim, criámos o datagrama FSNotFound que, para além do tipo, contém a informação do ficheiro em falta: nome, data de modificação e tamanho.

Todas as classes que representam cada um destes datagramas possuem um método estático *read(ByteArrayInputStream input)* que criará um datagrama do tipo dessa classe.

Há também uma classe denominada *FSFail* que não representa um datagrama mas é a responsável por indicar o tipo de erros que fazem com que um datagrama seja *Not_Acceptable*.

3.2 Interações

3.2.1 Handshaking / conexão servidor-gateway

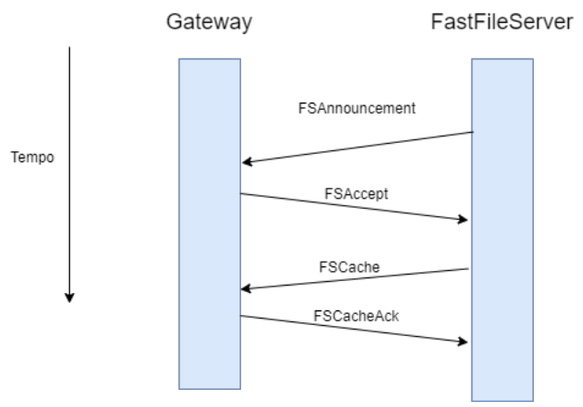


Figura 12: Diagrama Temporal do estabelecimento da conexão entre o FastFileServer e o gateway

Para anunciar a sua presença, o *FastFileServer* envia um *chunk FSAnnouncement* contendo o número de ficheiro que pretende disponibilizar ao *Gateway*. Com isto, ao receber a trama o *Gateway* procede em averiguar o pedido e envia uma resposta *FSAccept* (contendo os parâmetros para a sua execução) ou *FSFail* (de tipo *Not_Acceptable*) conforme este seja aceite ou não. Caso este seja aceite, o servidor prosseguirá a enviar uma série de *chunks* de tipo *Cache* que são posteriormente confirmadas pelo gateway através de *Acknoledgements* do tipo *Cache_Acknowledgement*.

3.2.2 Pedido de um ficheiro

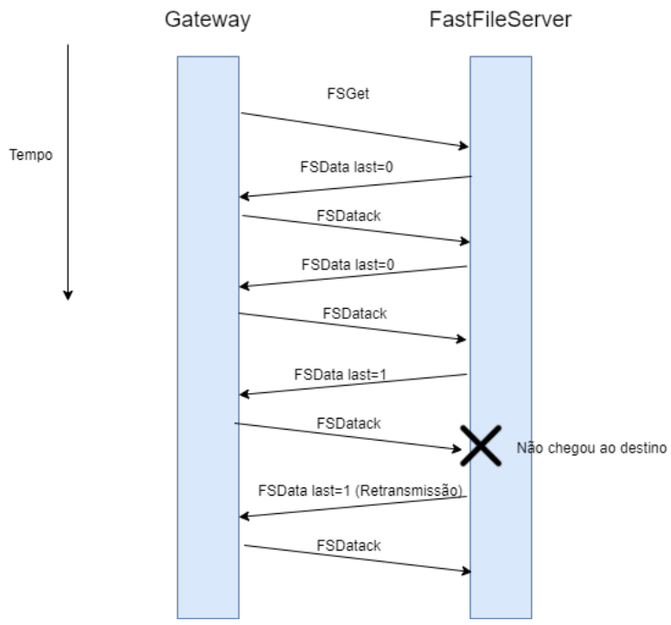


Figura 13: Diagrama Temporal do processo de transferência de um ficheiro

Para transferir um ficheiro, após selecionar um FastFileServer, o gateway envia-lhe um datagrama FSGet com a informação de qual ficheiro enviar. De seguida, o FastFileServer vai lendo o ficheiro selecionado e criando tramas de dados (FSData) e enviando de volta para o gateway. Por sua vez, o gateway responde, para cada trama recebida, com um acknowledgement. Como é explicado em mais detalhe posteriormente, caso o servidor não receba o acknowledgement dentro do tempo de timeout definido, reenvia a mesma trama.

3.2.3 Inatividade de Servidores

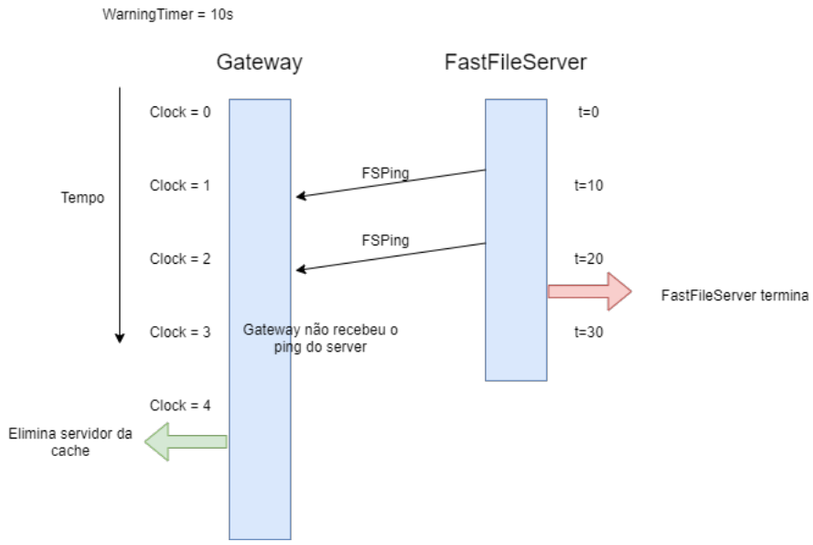


Figura 14: Diagrama Temporal do processo de Ping

Quando o gateway responde à tentativa de conexão de um servidor com um FSAccept, um dos campos dessa trama é o warningTimer. Este campo corresponde ao intervalo de tempo em que o servidor deve enviar um ping para o gateway para este saber que ainda está ativo. Quando deixa de receber pings, o gateway assume que o servidor ficou inativo e remove-o da sua cache.

4 Implementação

4.1 Servidor HTTP e HTTPS

O servidor *HTTP* foi implementado com recurso a *package* $com.sun.net.httpserver$ presente nas bibliotecas nativas ás instalações padrão do Java. Para utilizar esta classe, é necessário definir *handlers* para os pedidos *HTTP* e adicionalmente estabelecer as regras de gestão das respetivas *threads* que este possa a vir precisar. No

nosso caso, estes métodos encontram-se espalhados pelas classes *FSGateway*,*ServerManager*,*ServerHandler* e *FileRequest*. Adicionalmente, a *package* possui uma extensão á classe de *HttpServer* correspondente á implementação do serviço *HTTPs*, cuja forma de utilização é idêntica ao servidor inseguro, com a única diferença sendo a necessidade de configurar o certificado necessário para o seu funcionamento.

4.2 Cache no gateway

A cache de ficheiros do *gateway* encontra-se distribuída por dois mapeamentos complementares. O primeiro é essencialmente relevante aos pedidos *HTTP*, permite ao *gateway* reconhecer a *metadata* a partir do nome fornecido nos pedidos de como por exemplo *GET* e *HEAD*. O segundo é relevante para o escalonamento de tarefas, visto que este mapeia a *metadata* referida anteriormente aos endereços dos servidores que a disponibilizam. Desta forma é possível verificar rapidamente quais as opções existentes para se obter um ficheiro. Estes mapeamentos são geridos por dois trabalhadores diferentes, o *CacheUpdater* e o *CacheCleaner*.

O *CacheUpdater* interpreta *chunks* do tipo *Cache* colocados pelo *Dispatcher* numa fila presente no *ServerManager* atualizando ambos os mapeamentos à medida que lê os datagramas.

Por outro lado, o *CacheCleaner* recebe pedidos diretamente do *gateway*, correspondentes a dois tipos, remover entradas de servidores em ficheiros específicos ou remover todas as entradas de um servidor na cache. O primeiro caso é efetuado quando uma mensagem de ficheiro indisponível é enviada por um servidor durante um *Get*, e o segundo caso ocorre quando o servidor é considerado como indisponível e se encontra a ser removido do estado do servidor.

4.3 Gestão de Servidores

Existem vários contextos em que o *gateway* gere os servidores. De seguida iremos detalhar cada um deles.

4.3.1 Gestão de autenticação

Para os servidores se poderem autenticar, enviam um *announcement* e ficam à espera que o *gateway* envie uma resposta. Se o servidor não receber uma resposta ao fim de um tempo (instanciado nas propriedades do servidor), este tentará outra vez, até eventualmente desistir e terminando a sua execução. Por outro lado, o gateway, ao receber um *announcement* verificará se o servidor em questão não se encontra na blacklist e se tem disponibilidade para o gerir. Caso tenha, este instanciará um *handler* para a gestão do servidor que lhe responderá com uma trama do tipo *Accept* onde indicará varias informações vitais para a transmissão de dados (como, por exemplo, o tamanho máximo das tramas). Caso não seja aceite este envia uma mensagem de erro do tipo *Not_Acceptable*.

Adicionalmente, se um servidor se encontrava recentemente autenticado, se a instância deste ainda não foi caducada do gateway, este prosseguirá e reunirá o servidor com a sua instância perdida.

4.3.2 Gestão de conexões

Para garantir que todos os servidores registados no *gateway* estavam alcançáveis e vice-versa, foi implementada um sistema de "pings"que na verdade aproximam-se mais de *beacons* para garantir que cada máquina eram alcançável pela outra.

Tendo em conta a autenticação previamente falada, existe uma fase de autenticação em que o servidor é aceite ou rejeitado. Caso for aceite, receberá um FSAccept do gateway que contém uma parâmetro chamado *warningtimer* que indica o período em que o servidor deve reforçar a sua presença ao *gateway* (ou seja o periodo no qual este deve envia um *ping request*. De cada vez que o servidor envia um *ping*, fica à espera de uma resposta *ping response*. Caso esta trama ultrapasse o tempo de timeout (definido na propriedade *acktimeout*), reenvia o ping. Este processo é repetido até atingir o valor máximo de tentativas de conexão definido nas propriedades. Quando este valor é ultrapassado, assume-se que o servidor perdeu a conexão e este termina a sua execução.

No lado do gateway existe o *CyclicClock* que contabiliza os tempos definidos no *warningtimer* através de um relógio lógico ciclico. Cada vez que o gateway recebe o ping, irá ao *handler* do servidor correspondente e atualizará o valor correspondente ao relógio deste. Como quando o servidor fica inalcançável, deixa de enviar pings pelo que este valor deixa de ser atualizado. Se o gateway perceber que um servidor está atrasado um determinado tempo (também definido nas propriedades) em comparação ao relógio do gateway, assume que o servidor já não está disponível e procede a eliminar todos os registos seu estado interno, parando assim a thread do *ServerHandler* respetivo e enviando um pedido ao *CacheCleaner* para limpar todas as entradas desse servidor na cache.

4.3.3 Alocamento de Servidores para pedidos

Na implementação atual, cada servidor responde a um pedido de cada vez. Consequentemente, o *gateway* aloca os servidores conforme o número de pedidos existente na fila, dando preferência aos servidores menos ocupados para efectuar os downloads. Adicionalmente, caso um servidor alocado apresente um erro e não consiga transmitir, este reconheceria esse problema (se ele ocorresse antes da transmissão) e alocaria o próximo servidor na lista caso este existisse.

Desta forma o *gateway* apresenta paralelismo em escala o que permite um melhor serviço quanto maior for o número de servidores. Em contrapartida, o contra-reciproco também é verdade, e desta forma quando menos servidores existirem, menor será a escalabilidade do *gateway* e sucessivamente o tempo de resposta do pedido poderá piorar. Em sistemas reais, é possível que este problema não seja tão significativo, porém caso esta aplicação seja alvo de desenvolvimento futuro, este seria um dos primeiros pontos a serem revistos.

4.4 Reconstrução de ficheiros

Para reconstruir os ficheiros, cada *file request* está programado de forma a usar o mínimo possível de memória. Desta forma, os pacotes correspondentes a cada ficheiro, assim que disponíveis, são enviados para o cliente *HTTP* respeitando a ordem respetiva e desta forma, evitando o risco de ficarmos sem memória em momentos alto tráfego de dados.

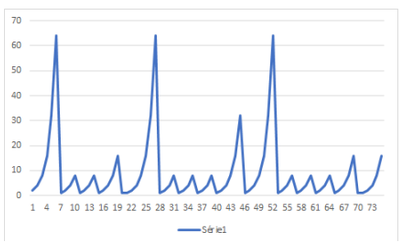
4.5 Controlo de erros UDP

Para controlo de erros nas tramas UDP as tramas de dados de ficheiros apresentam um valor de *checksum* que permite assim verificar se ocorreu algum erro durante a sua transmissão. Inicialmente, tinhamos implementado um novo tipo de datagrama (FSReGet) para estes casos, no entanto, com o sistema de acknowledgments que irá ser explicado a seguir, chegámos à conclusão que basta fazer com que o gateway não envie acknowledgement para o servidor quando o checksum falha. Para além disto, implementámos um sistema de acknowledgements que permite aos servidores terem a certeza que o gateway está a receber as tramas de dados e cache com sucesso. Para isto, sempre que o gateway recebe uma trama de dados ou cache, responde ao servidor com o Acknowledge correspondente. Do lado do servidor, caso este acknowledgment chegue antes do timeout configurado, então sabe que não irá ser necessário reenviar essa trama. Caso contrário, quando há o timeout, volta a enviar a trama e espera, novamente, pelo acknowledge correspondente. Deste modo, temos a certeza que os ficheiros chegam intactos ao gateway.

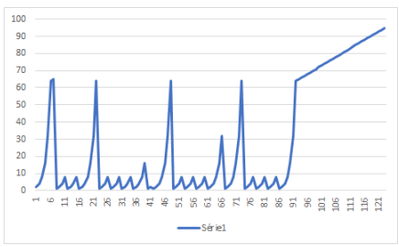
4.6 Gestão de congestionamento

Para cada datagrama de dados (seja ele um *FSData* ou um *FSCache*), é necessário efetuar uma confirmação da chegada de forma a gerir as tramas enviadas. Desta forma , para gerir o número de datagramas enviadas e consequentemente o numero de acks esperados, foi desenvolvida uma janela deslizante (presente no *FFSFowarder)*, parecia à implementada no protocolo TCP, de forma a garantir um fluxo consistente de informação sem sobrecarregar o *gateway*.

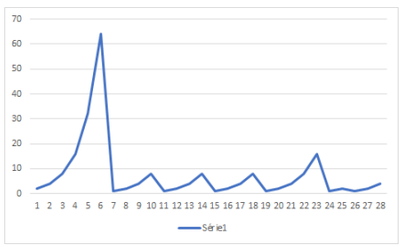
Desta forma, existem dois modos de evolução da janela deslizante. A primeira consiste num crescimento exponencial da janela enquanto que a segunda consiste num crescimento constante na janela. Este dois modos de evolução são adotados dependendo das seguintes condições. Quando a janela abre, encontra-se inicialmente num evolução exponencial e irá aumentando até um parâmetro instanciado nas propriedades. Quando esse parâmetro for alcançado a evolução da janela passa a ser constante ate a um valor máximo de abertura, caso o utilizador o tenha definido. Durante estes momentos caso aconteça um número pré-especificado de timeouts a janela é reposta ao estado inicial, e volta a crescer exponencialmente a partir do valor 1. Adicionalmente, caso isto não se verifica, mas forem detetados uma quantidade predefinida de acks duplicados, o tamanho da janela será reduzido para metade. Desta forma é possível garantir um bom funcionamento do gateway, visto que foi verificado em testes que o congestionamento deste resultava num tempo de serviço muito pior.



Evolução da janela com switch a 64 e timeout a 1000 nanosegundos



Evolução da janela com switch a 16 e timeout a 1000 nanosegundos



Evolução da janela com switch a 16 e timeout a 3000 nanosegundos

5 Testes e resultados

5.1 Browser

5.1.1 Ficheiros pequenos

Começamos por tentar transferir ficheiros de texto em vários formatos diferentes, .txt, .html, todos com sucesso e sem perda conteúdo. De seguida experimentámos transferir imagens e vídeos. Nas imagens não houve qualquer corrupção ou diminuição de qualidade. Nos vídeos, tanto em termos de imagem como de áudio também não houve qualquer problema.

5.1.2 Ficheiros grandes

Como, em vez de reconstruir os ficheiros no gateway e só depois enviar para o host, estamos a enviar os chunks à medida que o recebemos, quisemos testar com ficheiros de maiores dimensões para verificar se ainda iríamos ter problemas com o uso de memória por parte da aplicação. Assim, tentamos transferir um ficheiro .zip com 4187992 KB = 4,187 GB. Após a transferência ser concluída, experimentamos extrair o ficheiro e verificar se o filme contido estava corrompido. Ambas as operações foram concluídas com sucesso. Outro aspeto que este teste nos permitiu analisar foi a flutuação da velocidade de download que a nossa aplicação proporciona. Todas estas transferências foram efetuadas com ligação à rede Eduroam da Universidade do Minho durante o período

de aulas, pelo que, é correto assumir que existe congestionamento da rede. Ora, apesar de os 4,187 GB terem sido transferidos muito rapidamente, tivemos velocidades desde 11 KB/s a 108 MB/s. Acreditámos que esta flutuação se deva ao algoritmo de prevenção de congestionamento da rede implementado por nós. Ora, a descida da velocidade de transferência para a ordem dos KB/s deve ser causada pela deteção de congestionamento da rede devido a timeouts no lado do FastFileServer, o que provoca uma descida no tamanho da janela de envio nos servidores. Estas descidas eram seguidas por um aumento de velocidade novamente à ordem dos 100 MB/s o que também confirma a hipótese de esta flutuação se dever ao algoritmo de controlo de congestionamento da rede.

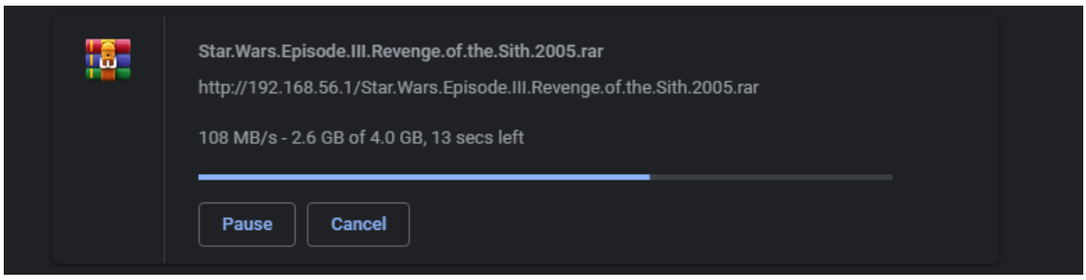


Figura 15: Instante em que a velocidade do ficheiro atingiu 108 MB/s

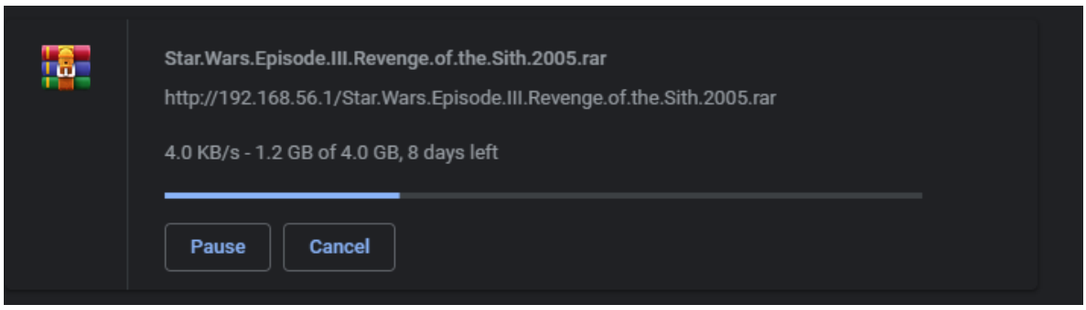


Figura 16: Instante em que a velocidade do ficheiro resetou para 11 KB/s

5.2 Vários pedidos em simultâneo

Um dos objetivos da aplicação é o gateway suportar múltiplos pedidos em simultâneo e distribuir a "carga"de pedidos de ficheiros pelos servidores. Assim, nos próximos testes iremos analisar o comportamento da aplicação face a múltiplos pedidos simultâneamente.

5.2.1 1 FastFileServer

Como já foi explicado, os chunks para um dado ficheiro vêm todos do mesmo FastFileServer, logo todos os FastFileServers têm uma lista de pedidos que estão a ser processados, o que nos permite calcular a "carga"com que cada servidor está. Esta decisão torna a distribuição de "carga"por múltiplos servidores bastante eficiente porém, no caso em que só o gateway só tem 1 FastFileServer conectado, este só pode responder sequencialmente aos vários pedidos que lhe são feitos.

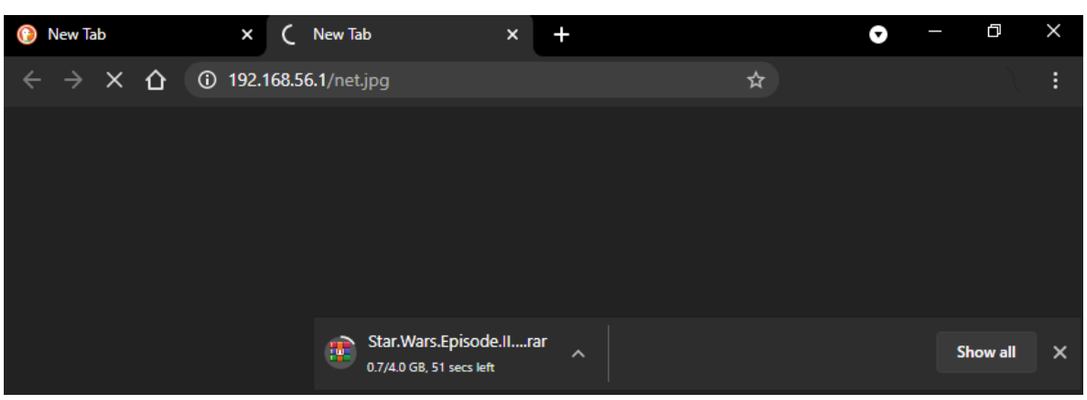


Figura 17: Múltiplos pedidos de ficheiros com 1 servidor conectado

Como esperado, caso façamos vários pedidos com 1 só FastFileServer conectado, as transferências são feitas uma após a outra, sequencialmente. Neste caso, o primeiro separador foi o primeiro a fazer um pedido de um ficheiro à aplicação e iniciou a transferência imediatamente mas, o segundo separador, após fazer um pedido de uma imagem ao servidor ainda está à espera do seu ficheiro.

5.2.2 Vários FastFileServers

Com vários FastFileServers, no caso em que os ficheiros pedidos estão presentes em todos os servidores o nosso algoritmo de distribuição de "carga"funciona como esperado, atribuindo cada ficheiro ao servidor que tem menos "carga", permitindo várias transferências ao mesmo tempo. Logo, este resultado também coincide com o esperado.

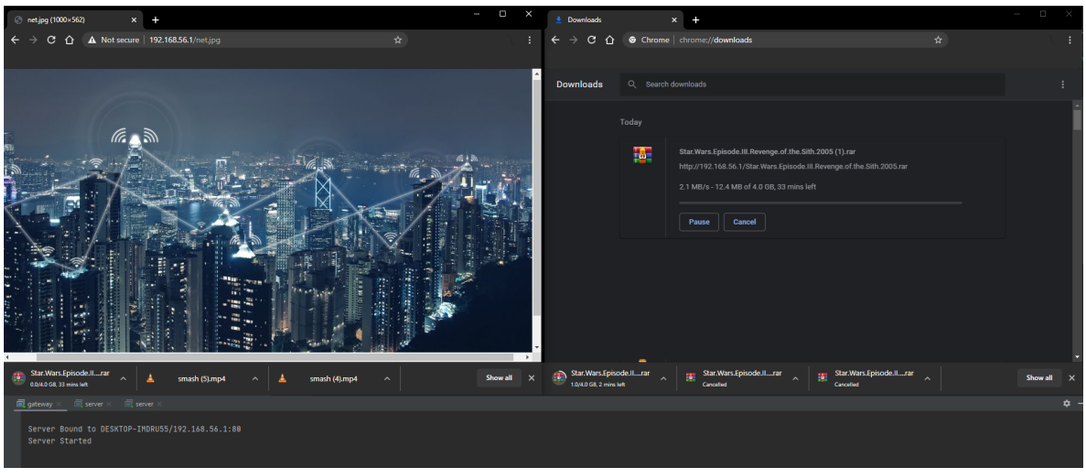


Figura 18: Múltiplos pedidos de ficheiros com 2 servidores conectados

5.3 Wget

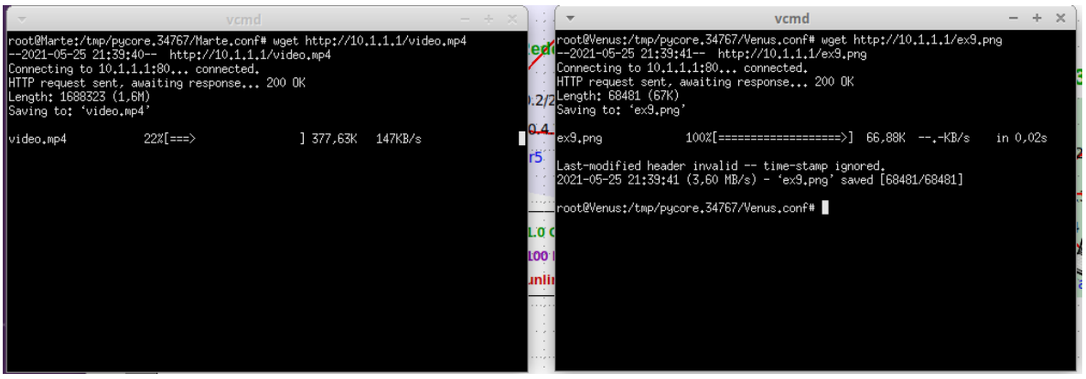


Figura 19: Múltiplos pedidos de ficheiros com 2 servidores conectados

6 Conclusões e trabalho futuro

Para a realização deste trabalho foi necessário, acima de tudo, implementar um gateway aplicacional e desenvolver um protocolo que teria de funcionar sobre UDP e não poderia nem ser orientado à conexão nem possuir qualquer informação de estado. Estes requisitos fizeram com que tivessemos de aprofundar o nosso conhecimento sobre as matérias lecionadas nas aulas, principalmente as matérias de Transporte e HTTP. Como decidimos fazer a nossa aplicação utilizando a linguagem de programação *JAVA*, tivemos também de pesquisar e estudar as interfaces que nos poderiam ter sido, e foram, muito úteis para uma melhor implementação da mesma. Fomos capazes de cumprir com os objetivos propostos com sucesso e acreditamos que a realização deste trabalho foi muito importante para a consolidação dos conhecimentos adquiridos na UC.

A Ficheiros de Configurações

A.1 FileServerGateway fsg.cfg

MaxRequests = 50 MTU = 4096 MaxServersCount = 40 ServerTolerance = 2 PingTimer = 300 GetTimeout = -1 Backlog = 50 seeIncomingFrames = false seeOutgoingFrames = false seeOutgoingAcknowledgements = false extendFrameOutput = false Alias = Alias67 Password = password KeyAlgorithm = RSA KeystoreFile = keys.keystore SSLProtocol = TLS KeyStoreAlgorithm = JKS KeyManagerAlgorithm = SunX509 TrustManagerAlgorithm = SunX509

A.2 FastFileServer ffg.cfg

stdTimeOut = 5000 ackTimeOut = 1000 maxQueueSize = 100 timeoutThreshold = 5 congestionAvoidanceSwitch = 64 maxConnectionAttempts = 3 maxPingAttempts = 10 seeThresholdResolve = false seeIncomingFrames = false seeOutgoingFrames = false extendFrameOutput = false