TP2: Serviço de Transferência Rápida e Fiável de Dados sobre UDP

Grupo PL59
Alberto Campinho Faria
Bernardo Manuel Ribeiro Marques Soares Silva
Nuno Filipe Maranhão dos Reis

University of Minho, Department of Informatics, 4710-057 Braga, Portugal
e-mail: {a79077,a77230,a77310}@alunos.uminho.pt

Resumo Este relatório detalha o sistema *peer-to-peer* de transferência de ficheiros intitulado FILESHARE, desenvolvido no âmbito do segundo trabalho prático da Unidade Curricular de Comunicações por Computador do curso de Mestrado Integrado em Engenharia Informática, no ano letivo de 2018/2019, da Universidade do Minho.

1 Introdução

Este documento consiste no relatório correspondente ao segundo trabalho prático desenvolvido no âmbito da Unidade Curricular de Comunicações por Computador do curso de Mestrado Integrado em Engenharia Informática, no ano letivo de 2018/2019, da Universidade do Minho.

Com o trabalho em questão, pretende-se que seja construído um sistema *peer-to-peer* transferência fiável de ficheiros, cada *peer* podendo iniciar transferências de *download* e *upload*. O protocolo base de transferência de dados deve garantir a entrega sem erros e ordenada de mensagens, no entanto devendo também utilizar somente conexões UDP.

Neste sentido, foi desenvolvido o sistema FILESHARE, o qual cumpre todos os requisitos impostos pelo enunciado do trabalho. O sistema disponibiliza ainda várias funcionalidades adicionais, destacando-se (1) a possibilidade de realizar transferências em simultâneo entre qualquer par de *peers* (multiplexando várias conecções no mesmo socket UDP) e (2) a paralelização do *download* de ficheiros utilizando vários *peers*.

Material disponibilizado. A implementação do sistema, na linguagem Java e sob a forma de um projeto do IDE *IntelliJ IDEA*[1], é incluída no arquivo `CC-TP2-PL59-Codigo.zip`, disponibilizado juntamente com este relatório. O arquivo contém ainda o programa `fileshare.jar`, resultado da compilação do projeto, através do qual se pode utilizar o sistema. Para executar este programa, recomenda-se a utilização do *script* `fileshare.sh`, o qual adiciona funcionalidades de histórico de comandos ao *prompt* interativo do programa.

Estrutura do documento. Na Secção 2 são descritas todas as funcionalidades disponibilizadas pelo sistema e o modo de utilização da interface de linha de comandos do mesmo. Na Secção 3 é depois apresentada a arquitetura interna do sistema e detalhados vários aspetos relativos à sua implementação, incluindo o protocolo de comunicação fiável sobre UDP. A Secção 4 conclui o relatório.

2 Funcionalidades

Nesta secção enumeram-se as funcionalidades disponibilizadas pelo sistema FILESHARE, detalhando-se depois o modo de utilização da interface de linha de comandos do mesmo.

[1] https://www.jetbrains.com/idea/

2.1 Funcionalidades disponibilizadas

O sistema FILESHARE permite realizar transferências de ficheiros entre *peers*, cada um exportando uma sub-árvore de um sistema de ficheiros local através de um *socket* UDP (cf. Figura 1). As transferências de ficheiros são depois realizadas entre as várias sub-árvores exportadas pelos *peers* envolvidos. Transferências em ambos os sentidos (*i.e.*, *download* e *upload*) podem ser iniciadas por qualquer *peer*. O sistema disponibiliza todas as funcionalidades exigidas pelo enunciado do trabalho, expondo ainda várias funcionalidades adicionais.

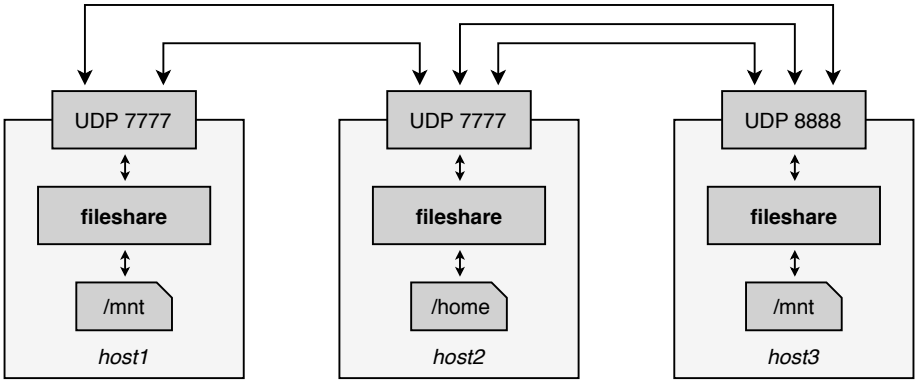

Figura 1. Arquitetura *peer-to-peer* do sistema FILESHARE.

Aceitação de pedidos de transferência. Cada *peer* define, com recurso a uma *whitelist*, quais os *peers* que podem ter acesso à sub-árvore local exportada, tanto para transferências de *download* como de *upload*. Esta *whitelist* consiste num conjunto de intervalos de endereços IPv4 e IPv6 em notação CIDR. Ao se receber um pedido de transferência, este apenas é aceite se o endereço do *peer* que realizou o pedido pertencer a algum dos intervalos de endereços na *whitelist*.

***Transferências de* download.** Um *peer* pode obter um ficheiro a partir de um ou mais outros *peers* realizando uma transferência de *download*. O utilizador deve especificar o caminho para o ficheiro a ser obtido (relativo às raizes das sub-árvores exportadas pelos *peers* remotos) e os endereços dos *peers* a partir dos quais o ficheiro deve ser transferido. Pode-se ainda especificar o caminho do ficheiro local resultante da transferência, caso não se deseje utilizar o mesmo caminho que o do ficheiro remoto.

Caso o utilizador indique que deve ser utilizado mais de um *peer* remoto para se realizar a transferência, o ficheiro é dividido em segmentos de tamanho igual e cada segmento é obtido de um *peer* distinto, em paralelo. Neste caso é também verificado que todos os *peers* reportam o mesmo tamanho de ficheiro.

***Transferências de* upload.** Cada *peer* pode também enviar um ficheiro para um ou mais outros *peers* iniciando uma transferência de *upload*. O utilizador especifica o caminho para o ficheiro a ser enviado (relativo à raiz da sub-árvore exportada pelo *peer* local) e os endereços dos *peers* para os quais o ficheiro deve ser transferido. É ainda possível indicar o caminho do ficheiro remoto resultante da transferência, ao invés de se utilizar o mesmo caminho que o do ficheiro local.

Caso o utilizador indique mais de um *peer* remoto, o ficheiro é transmitido em paralelo e na sua totalidade para cada um dos *peers*.

Transferências concorrentes. O sistema permite realizar qualquer número transferências em simultâneo entre qualquer par de *peers*, incluindo combinações de transferências de *download* e *upload*. Cada *peer* garante que o mesmo ficheiro não é modificado concorrentemente por mais de uma transferência, permitindo no entanto acessos de leitura simultâneos.

Nota-se também que, ao se receber um ficheiro (seja por se ter requerido uma transferência de *download* ou por se servir um pedido de transferência de *upload*), as alterações a esse ficheiro apenas são tornadas visíveis se a transferência for realizada com sucesso na sua totalidade. Em particular, se um ficheiro com o mesmo caminho já existir, este não é alterado caso a transferência falhe ou não termine com sucesso.

2.2 Interface de linha de comandos

O sistema FILESHARE é constituído por um único programa (ficheiro `fileshare.jar` no arquivo `CC-TP2-PL59-Codigo.zip` disponibilizado juntamente com este relatório), o qual permite iniciar um *peer* local e executar transferências utilizando outros *peers*. O programa disponibiliza uma interface de linha de comandos e tem a seguinte forma de invocação:

```
fileshare [-a|--allow-all] [-p|--port <n>] <export_dir>
```

O argumento `<export_dir>` especifica a raiz da sub-árvore de sistema de ficheiros local a ser exportada pelo *peer*, devendo ser um caminho para uma diretoria. As opções têm os seguintes significados:

- `-a` ou `--allow-all` — especifica que qualquer outro *peer* pode realizar transferências utilizando o *peer* local;
- `-p <n>` ou `--port <n>` — especifica a porta UDP local que o *peer* deve utilizar (se esta opção não for dada, a porta 7777 é selecionada).

O programa disponibiliza um *prompt* interativo onde o utilizador pode introduzir comandos com o objetivo de realizar transferências de ficheiros entre *peers*. Para executar o programa, recomenda-se a utilização do *script* `fileshare.sh` (incluído também no arquivo `CC-TP2-PL59-Codigo.zip`), o qual expõe o mesmo modo de utilização mas adiciona funcionalidades de histórico de comandos ao *prompt* interativo.

O *prompt* interativo disponibiliza os seguintes comandos:

`exit`
: Termina o *peer* local e sai do *prompt* interativo. Equivalente a fechar a *stream* de *standard input* através da combinação de teclas Ctrl+D.

`whitelist-add <cidr_peers...>`
: Adiciona os padrões de endereços em notação CIDR especificados (separados por espaços) à *whitelist* de endereços de *peers*.

`whitelist-remove <cidr_peers...>`
: Remove os padrões de endereços em notação CIDR especificados (separados por espaços) da *whitelist* de endereços de *peers*.

`whitelist-all`
: Adiciona os padrões `0.0.0.0/0` e `::/0` à *whitelist* de endereços de *peers*, permitindo que qualquer outro *peer* realize transferências utilizando o *peer* local. Equivalente a se especificar a opção `-a` ou `--allow-all` ao se invocar o programa.

`whitelist-clear`
: Remove todos os padrões de endereços da *whitelist* de endereços de *peers*.

`get <remote_file> [as <local_file>] from <peers...>`
: Realiza o *download* do ficheiro com o caminho `<remote_file>` a partir dos *peers* com endereços `<peers...>` (separados por espaços), armazenando-o com o caminho `<local_file>` (ou, se não especificado, com o caminho `<remote_file>`). Se for dado mais de um *peer*, a transferência é particionada e realizada em simultâneo a partir deles.

put <local_file> [as <remote_file>] to <peers...>
Realiza o *upload* do ficheiro com o caminho `<local_file>` para os *peers* com endereços `<peers...>` (separados por espaços), armazenando-o com o caminho `<remote_file>` (ou, se não especificado, com o caminho `<local_file>`). Se for dado mais de um *peer*, a transferência é realizada em simultâneo para todos eles.

concurrent
Permite realizar qualquer número e tipo de transferências em simultâneo. Execuções de **get** e **put** após a utilização deste comando apenas são realizadas ao se introduzir o comando **run**, após o qual se regressa ao modo de execução imediata de transferências. O comando **cancel** pode ser utlizado para se cancelar a realização de transferências já introduzidas e se voltar ao modo de execução imediata.

2.3 Exemplos de utilização

Apresentam-se de seguida exemplos de utilização do programa `fileshare` e dos comandos do *prompt* interativo disponibilizado pelo mesmo.

- Inicia um *peer* na porta 8888, exportando a diretoria `/mnt`, e adiciona os intervalos de endereços `192.168.1.0/24` e `193.137.0.0/16` à *whitelist*, ficando apto para servir transferências de ficheiros requeridas por *peers* cujos endereços estejam contidos em algum de ambos esses intervalos.

```
$ fileshare -p 8888 /mnt
Exporting directory "/mnt" through UDP port 8888.
> whitelist-add 192.168.1.0/24 193.137.0.0/16
>
```

- Inicia um *peer* na porta 7777, exportando a diretoria `/mnt`, e realiza o *download* do ficheiro `file-1` a partir dos *peers* com endereços `192.168.1.123`, `192.168.1.234` e `172.157.2.167`, e portas 7777, 12345 e 7777, respetivamente. No exemplo, a transferência está em progresso, tendo-se já recebido 63% do ficheiro.

```
$ fileshare /mnt
Exporting directory "/mnt" through UDP port 7777.
> get file-1 from 192.168.1.123 192.168.1.234:12345 172.157.2.167
[ 63%] Getting file-1 from 3 peers... (9673.28 KiB/s)
```

- Inicia um *peer* na porta 7777, exportando a diretoria `/mnt`, e realiza, concorrentemente, (1) o *download* do ficheiro `file-2` a partir do *peer* com *hostname* `host123` e porta 8888, e (2) o *upload* do ficheiro `my-things/file-3` para o *peer* com *hostname* `host123` e para o *peer* com endereço `172.157.2.167`, ambos na porta 7777, e com o nome `another-name` nos *peers* de destino. A primeira transferência falhou devido ao *peer* remoto rejeitar a conecção (o endereço local não pertence à *whitelist* desse *peer*). A segunda transferência terminou com sucesso.

```
$ fileshare /mnt
Exporting directory "/mnt" through UDP port 7777.
> concurrent
  > get file-2 from host123:8888
  > put my-things/file-3 as another-name to host123 172.157.2.167
  > run
  ERROR! The peer refused to establish a connection.
  [100%] Put file-1 to 2 peers. (23.74 MiB/s)
> exit
$
```

3 Implementação

Tendo-se apresentado as funcionalidades disponibilizadas pelo sistema FILESHARE, descreve-se nesta secção a implementação do mesmo, primeiro delineando a sua arquitetura e depois detalhando os seus principais componentes.

3.1 Arquitetura

O sistema foi integralmente implementado na linguagem Java (versão *Java SE 11*), fazendo uso das duas bibliotecas seguintes:

- *argparse4j*[2] versão 0.8.1, utilizada para processar os argumentos da linha de comandos;
- *IPAddress*[3] versão 5.0.2, utilizada para processar *strings* em notação CIDR, representando intervalos contíguos de endereços IPv4 ou IPv6.

A implementação do sistema foi divida em três camadas, a cada uma correspondendo um *package* Java:

1. *Camada de comunicação fiável* (*package* `fileshare.transport`), encapsulando a implementação do protocolo de transferência fiável de dados sobre UDP e expondo uma interface de estabelecimento de conecções para comunicação entre *hosts*;
2. *Camada de transferência de ficheiros* (*package* `fileshare.core`), utilizando a camada anterior para implementar toda a lógica de transferência de ficheiros entre *peers*;
3. *Camada de interface de utilizador* (*package* `fileshare.ui`), implementando a interface de linha de comandos do programa, recorrendo a serviços expostos pela camada anterior.

As interdependências entre estes três *packages* é ilustrada na Figura 2. O código-fonte do sistema está extensivamente documentado através de comentários Javadoc embutidos no próprio código-fonte.

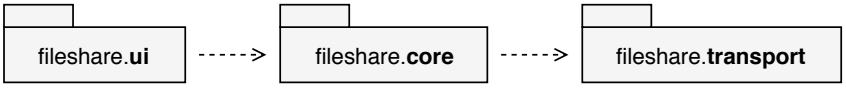

Figura 2. Dependências entre *packages* Java da implementação do sistema.

Em cada uma das subsecções seguintes, detalham-se os aspetos mais relevantes de cada uma das camadas supramencionadas, na ordem apresentada anteriormente.

3.2 Camada de comunicação fiável

A camada de comunicação fiável implementa um protocolo de comunicação sobre UDP com garantia de transmissão sem erros e de ordem. Apresenta-se aqui primeiro a interface exposta por esta camada, detalhando-se depois o funcionamento do protocolo na qual esta se baseia.

Interface disponibilizada. As classes que constituem a interface pública desta camada pertencem ao *package* `fileshare.transport` e são ilustradas na Figura 3, juntamente com as suas interdependências ao nível da interface.

A classe `Endpoint` representa um par endereço-porta, similarmente à classe `java.net.InetSocketAddress`. Ao contrário desta última, no entanto, a classe `Endpoint` não permite

[2] https://argparse4j.github.io/
[3] https://seancfoley.github.io/IPAddress/

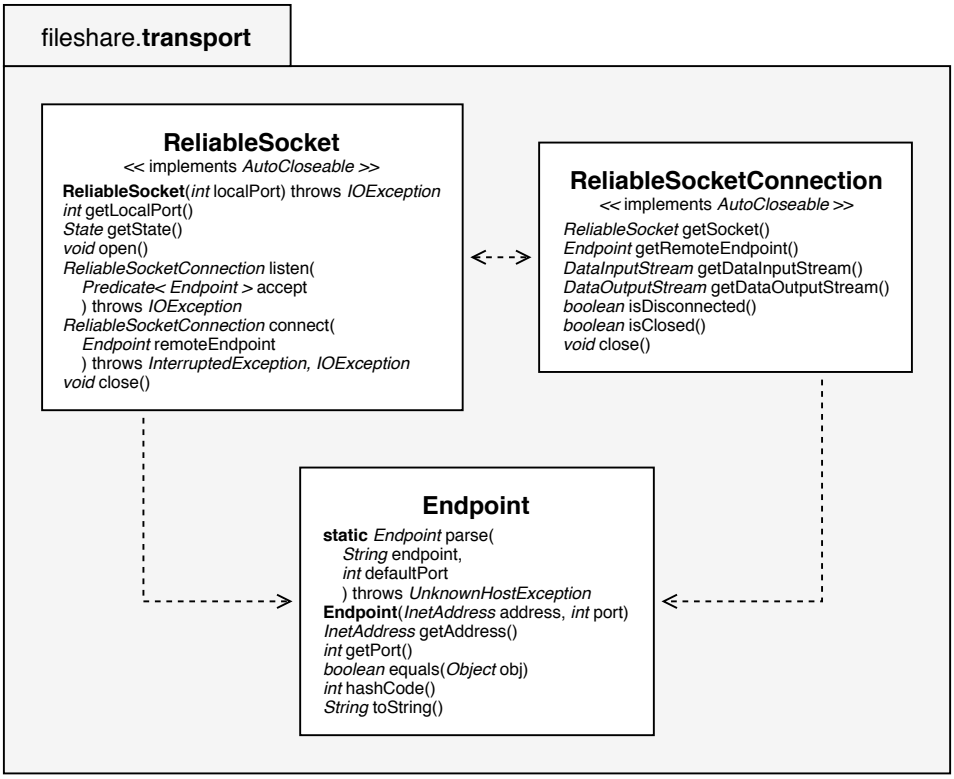

Figura 3. Classes do *package* `fileshare.transport` e dependências ao nível da interface.

hostnames não resolvidos, garantindo que erros relativos à resolução de nomes são tratados pelas camadas superiores.

A classe principal desta camada, `ReliableSocket`, gere um *socket* UDP local, permitindo estabelecer conecções *duplex* com outros *sockets* similares, estas últimas sendo representadas por instâncias da classe `ReliableSocketConnection`.

O método `connect()` da classe `ReliableSocket` permite efetuar pedidos de estabelecimento de conecções com outros *sockets*. O método `listen()` é utilizado para a tratar estes pedidos de conecção, permitindo também definir um predicado que determine se um dado pedido de conecção deve ser aceite ou rejeitado.

Os métodos `getDataInputStream()` e `getDataOutputStream()`, ambos da classe `ReliableSocketConnection`, devolvem *streams* que podem ser utilizadas para se receber e enviar dados através da conecção respetiva. O método `close()` termina a conecção.

Para mais informações sobre a interface exposta por esta camada, remete-se o leitor para a documentação *Javadoc* embutida no código-fonte.

Protocolo de comunicação fiável. A camada de comunicação fiável permite estabelecer canais *duplex* de comunicação orientados à conecção, com garantia de transmissão sem erros, sem dados duplicados e ordenada. Todas as interações entre os *sockets* descritos acima são, internamente, realizadas através de *packets*, os quais podem ser de vários tipos, cada um tendo o seu formato específico. Estes tipos, significados correspondentes e formatos são especificados na Tabela 1.

A descrição que se segue faz referência a vários valores não especificados, tais como *timeouts* e números máximos de retransmissões. Estes valores estão definidos na classe `Config` do *package* `fileshare.transport`.

Os primeiros 5 bytes de todos os *packets* seguem o mesmo formato: 4 bytes para um *checksum* e 1 byte para identificar o tipo de *packet*. O *checksum* é computado por quem envia

Tipo	Descrição	Formato
`CONN`	Pedido de estabelecimento de conecção	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 0) 2 bytes: Identificador local da conecção
`CONN-ACCEPT`	Notificação da aceitação de um pedido de conecção	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 1) 2 bytes: Identificador remoto da conecção 2 bytes: Identificador local da conecção
`CONN-REJECT`	Notificação da rejeição de um pedido de conecção	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 2) 2 bytes: Identificador remoto da conecção
`DATA`	Envio de dados	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 3) 2 bytes: Identificador local da conecção 8 bytes: Posição dos dados na *stream* Entre 1 e 1415 bytes: Dados (*payload*)
`DATA-ACK`	Confirmação da receção de dados	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 4) 2 bytes: Identificador local da conecção 8 bytes: Posição na *stream* tal que todos os dados anteriores tenham sido recebidos
`DISC`	Notificação do término da conecção	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 5) 2 bytes: Identificador local da conecção
`DISC-ACK`	Confirmação da receção da notificação do término da conecção	4 bytes: Checksum 1 byte: Identificador do tipo de pacote (= 6) 2 bytes: Identificador local da conecção

Tabela 1. Tipos de *packets* trocados pelo protocolo de comunicação fiável.

o *packet*, com base no conteúdo do mesmo, e é recomputado e comparado pelo destinatário. Se o *checksum* recebido no *packet* tiver um valor distinto do computado pelo recipiente, o *packet* é simplesmente descartado.

Por forma a se estabelecer uma conecção, é enviado um *packet* do tipo `CONN`. Ao receber um destes *packets*, o recipiente responde com um outro *packet* do tipo `CONN-ACCEPT` ou `CONN-REJECT`, aceitando ou rejeitando a conecção, respetivamente. Se o *packet* `CONN` não obtiver resposta, este é reenviado (após um determinado intervalo de tempo). Se após um certo número de retransmissões não existir resposta, é reportado um erro à camada superior.

Após se estabelecer uma conecção, ambos os lados da conecção podem enviar dados através de `packets` do tipo `DATA`. O recipiente deve depois enviar um *packet* `DATA-ACK` por forma a reconhecer a receção dos dados. Se não for recebido um *packet* `DATA-ACK`, os dados são retransmitidos. Resumidamente, é seguida uma abordagem *Go-Back-N*, não existem *acknowledgments* negativos e os *acknowledgments* são cumulativos e expressos como *offsets* em bytes correspondentes aos dados recebidos desde que a conecção foi estabelecida.

Ao terminar uma conecção é enviado um *packet* `DISC`, aguardando-se depois uma resposta na forma de um *packet* `DISC-ACK`. O *packet* `DISC` é retransmitido se não houver

resposta, até um determinado número máximo de retransmissões, após o qual a conecção é considerada, de qualquer forma, terminada.

Se um dos lados da conecção morrer, isto é detetado pela ausência de respostas `DATA-ACK` a dados enviados, e a conecção é dada como terminada.

3.3 Camada de transferência de ficheiros

A camada de transferência de ficheiros implementa, com recurso à camada de comunicação fiável previamente descrita, o protocolo *peer-to-peer* de transferência de ficheiros disponibilizado pelo sistema FILESHARE. De seguida apresenta-se a interface exposta por esta camada, descrevendo-se também os protocolos de comunicação entre *peers* para a execução de transferências de ficheiros.

Interface disponibilizada. As classes que constituem a interface pública da camada de transferência de ficheiros, todas pertencentes ao *package* `fileshare.core`, são ilustradas na Figura 4 juntamente com as suas interdependências ao nível da interface.

A classe `Peer` gere um *peer* local no sistema FILESHARE, sendo então a classe central desta camada. A sua interface permite gerir a *whitelist* de endereços de *peers*, descrita previamente, e, através do método `runJobs()`, iniciar a execução de transferências de ficheiros — aqui denominadas de *jobs* — com outros *peers*. Esta classe serve também, em segundo plano, pedidos de transferência de ficheiros realizados por outros *peers*.

Brevemente, as restantes classes e enumerações têm as seguintes responsabilidades:

- `AddressRange` – representa um intervalo contíguo de endereços IPv4 ou IPv6;
- `AddressWhitelist` – utilizada para representar a *whitelist* de endereos de *peers*;
- `ExportedDirectory` – gere a sub-árvore do sistema de ficheiros exportada pelo *peer*;
- `JobType` – define os dois tipos de transferência: *download* (`GET`) e *upload* (`PUT`);
- `Job` – define uma transferência de um ficheiro, em qualquer direção e envolvendo qualquer número de *peers* remotos;
- `JobState` – representa o estado de um *job* que está a decorrer.

Para mais informações sobre a interface exposta por esta camada, remete-se o leitor para a documentação *Javadoc* embutida no código-fonte.

Protocolo de transferência de ficheiros. Para executar um *job*, o *peer* estabelece conecções com todos os outros *peers* envolvidos na transferência e segue, depois, um protocolo de comunicação específico à direção da transferência.

Os protocolos utilizados em transferências dos tipos *get* e *put* são delineados nas Tabelas 2 e 3, respetivamente, onde o *cliente* corresponde ao *peer* que requeriu a transferência e o *servidor* corresponde ao *peer* restante (caso exista mais de um *peer* remoto, o protocolo é seguido para cada um deles). Explicita-se que estas tabelas não são uma especificação completa do protocolo, em particular não apresentando todos os casos de erro, destinando-se apenas a ilustrar a implementação do mesmo.

3.4 Camada de interface de utilizador

Por fim, a camada de interface de utilizador implementa a interface de linha de comandos disponibilizada pelo comando `fileshare`. Para efeitos de referência, as classes que implementam esta camada são ilustradas na Figura 5. No entanto, considera-se que a descrição detalhada desta camada está fora do âmbito do presente relatório.

Como tal, para mais informações sobre a implementação desta camada, remete-se o leitor para a documentação *Javadoc* embutida no código-fonte.

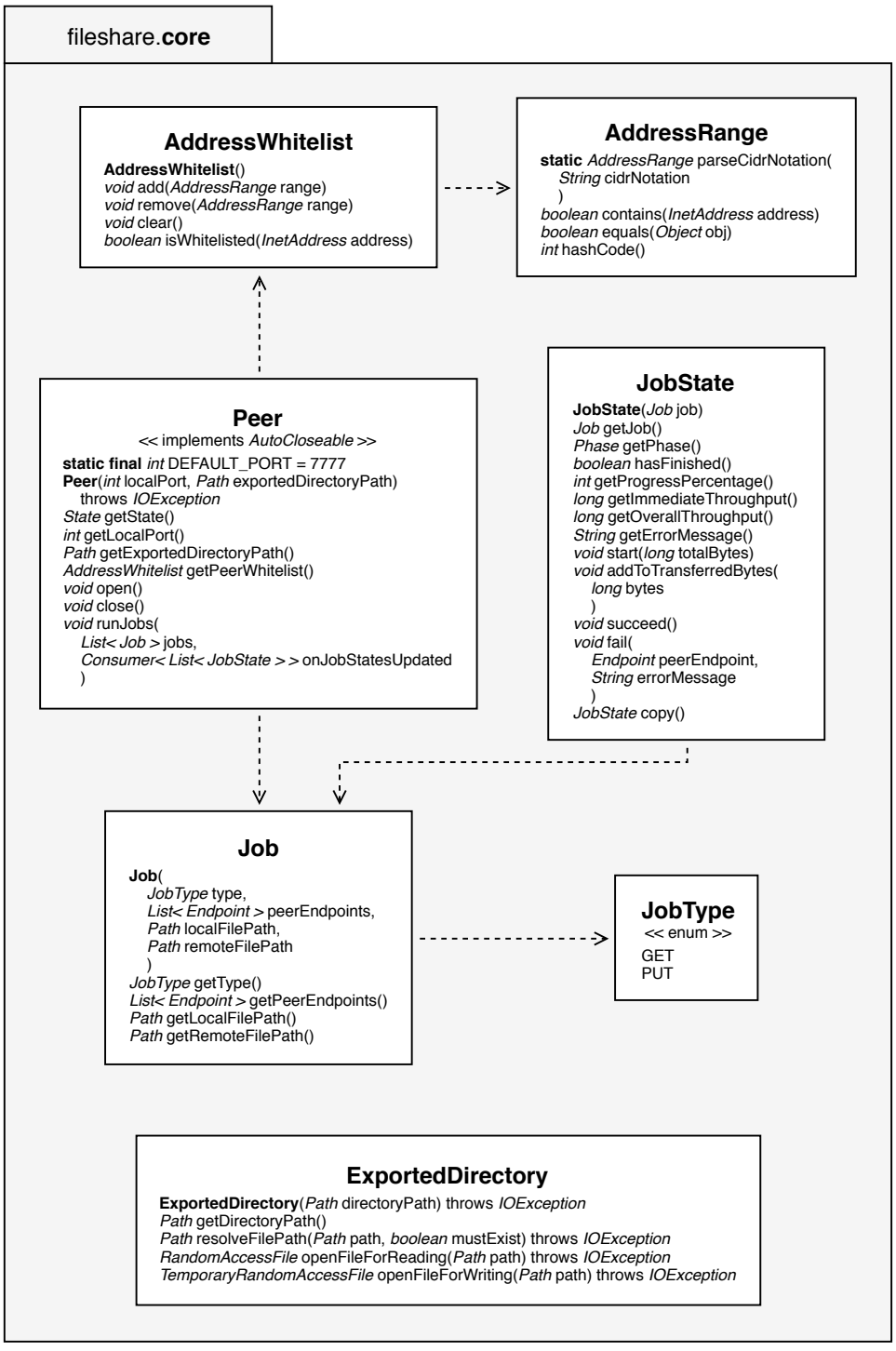

Figura 4. Classes do *package* `fileshare.core` e dependências ao nível da interface.

	Cliente	Servidor

Comportamento normal	`writeByte(0)` (tipo de *job*)
	`writeUTF(path_file_servidor)`
	`flush()`	

		`readByte()` (= 0)
		`path_file_servidor = readUTF()`
		[se erro, ir para *caso de erro 1*]
		`writeLong(tamanho_file)`
		`flush()`

	`tamanho_file = readLong()`
	[se erro, ir para *caso de erro 2*]
	`writeLong(offset_segmento)`
	`writeLong(tamanho_segmento)`
	`flush()`	

		`offset_segmento = readLong()`
		`tamanho_segmento = readLong()`
		escreve conteúdo do segmento
		`flush()`
		`close()`

	lê e armazena conteúdo do segmento `close()`	

Caso de erro 1 (*e.g.*, ficheiro não existe no servidor)		`writeLong(-1)`
		`writeUTF(mensagem_erro)`
		`flush()`
		`close()`

	`readLong()` (= -1)
	`mensagem_erro = readUTF()`
	reporta erro
	`close()`	

Caso de erro 2 (*e.g.*, tamanhos distintos entre servidores)	reporta erro `close()`	
		deteta *end-of-file* `close()`

Tabela 2. Protocolo de comunicação entre dois *peers*, ao nível da camada de transferência de ficheiros, para execução de um *job* do tipo *get*.

	Cliente	**Servidor**

Comportamento normal	`writeByte(1)` (tipo de *job*)
	`writeUTF(path_file_servidor)`
	`writeLong(tamanho_file)`
	`flush()`
	
	`readUTF()` (= `""`)
	escreve conteúdo do ficheiro
	`flush()`
	
	`readUTF()` (= `""`)
	`close()`
		
		`readByte()` (= 1)
		`path_file_servidor = readUTF()`
		`tamanho_file = readLong()`
		[se erro, ir para *caso de erro 1*]
		`writeUTF("")`
		`flush()`
		
		lê e armazena conteúdo do ficheiro
		[se erro, ir para *caso de erro 2*]
		`writeUTF("")`
		`flush()`
		`close()`

Caso de erro 1 (*e.g.*, ficheiro está em utilização)	
	
	`mensagem_erro = readUTF()`
	`close()`
	reporta erro
		`writeUTF(mensagem_erro)`
		`flush()`
		`close()`

Caso de erro 2 (*e.g.*, erro ao escrever ficheiro)	
	
	`mensagem_erro = readUTF()`
	`close()`
	reporta erro
		`writeUTF(mensagem_erro)`
		`flush()`
		`close()`

Tabela 3. Protocolo de comunicação entre dois *peers*, ao nível da camada de transferência de ficheiros, para execução de um *job* do tipo *put*.

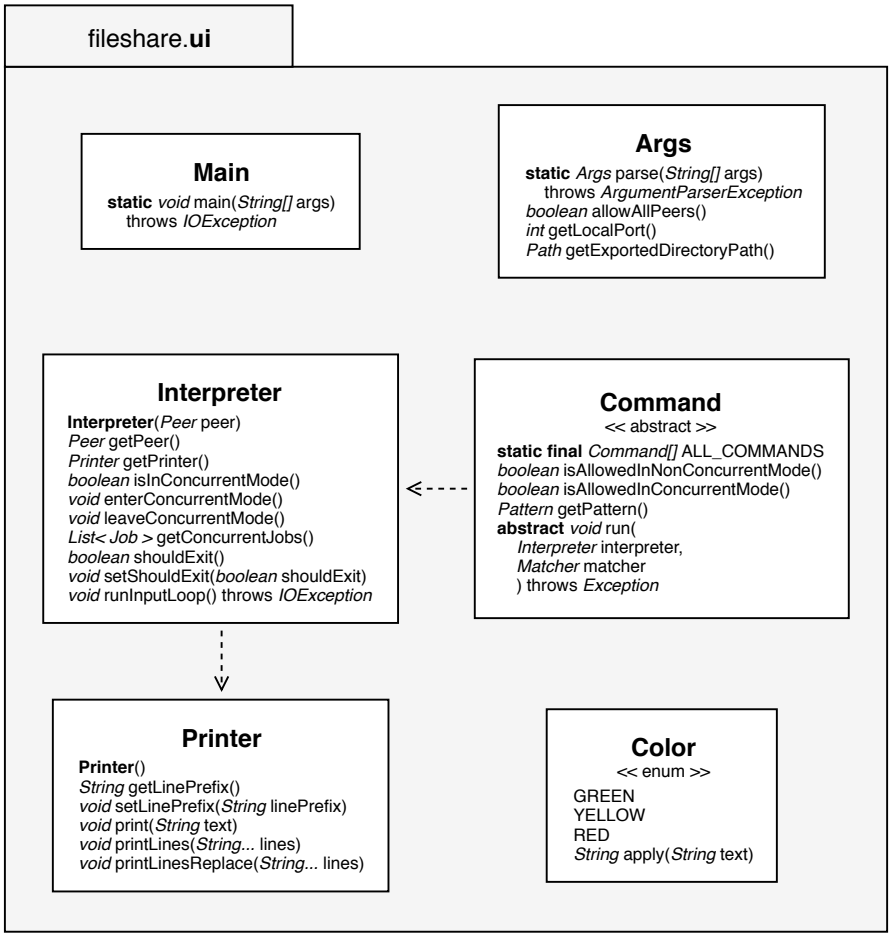

Figura 5. Classes do *package* `fileshare.ui` e dependências ao nível da interface.

4 Conclusão

Neste relatório detalhou-se o sistema FILESHARE, desenvolvido no âmbito do segundo trabalho prático da Unidade Curricular de Comunicações por Computador do curso de Mestrado Integrado em Engenharia Informática, no ano letivo de 2018/2019, da Universidade do Minho.

Começou-se por descrever as funcionalidades do sistema e o modo de utilização da interface de linha de comandos disponibilizado pelo mesmo, detalhando-se depois a sua arquitetura, implementação e funcionamento do protocolo de comunicação fiável utilizado pelo sistema.

O sistema FILESHARE cumpre todos os requisitos impostos pelo enunciado do trabalho, permitindo efetuar transferências de ficheiros entre *peers* em modo de *download* e *upload* de forma fiável. O sistema disponibiliza ainda várias funcionalidades adicionais, destacando-se (1) o suporte para transferências simultâneas de ficheiros entre qualquer par de *peers* em ambas as direções e (2) o suporte para transferências *multi-peer*, permitindo paralelizar o *download* de ficheiros utilizando vários *peers* de origem.