*25 de maio de 2019*

Sistemas Distribuídos

***Serviço de Backup Distribuído para a Internet***

*(Mestrado Integrado em Engenharia Informática e Computação)*

***Grupo 7 da turma 4:***

* Helena Montenegro [up201604184@fe.up.pt](mailto:up201604184@fe.up.pt)
* João Álvaro Ferreira [up201605592@fe.up.pt](mailto:up201605592@fe.up.pt)
* João Fidalgo up201303098@fe.up.pt
* Juliana Marques up201605568@fe.up.pt

Índice

[***1.*** ***Introdução*** 2](#_Toc9705871)

[***2.*** ***Arquitetura do Sistema*** 2](#_Toc9705872)

[***3.*** ***Implementação do Sistema*** 3](#_Toc9705873)

[***3.1.*** ***Mensagens trocadas*** 3](#_Toc9705874)

[***3.1.2.*** ***Protocolo de Backup*** 4](#_Toc9705875)

[***3.1.3.*** ***Protocolo de Restore*** 5](#_Toc9705876)

[***3.1.4.*** ***Protocolo de Delete*** 6](#_Toc9705877)

[***3.1.5.*** ***Protocolo de Reclaim*** 6](#_Toc9705878)

[***3.1.6.*** ***Verificação de Peers Ativos*** 7](#_Toc9705879)

[***3.1.7.*** ***Mensagens entre PeerManagers*** 7](#_Toc9705880)

[***3.2.*** ***Concorrência*** 8](#_Toc9705881)

[***3.3.*** ***Protocolos*** 8](#_Toc9705882)

[***3.4.*** ***Conexões seguras usando JSSE*** 9](#_Toc9705883)

[***4.*** ***Aspetos relevantes*** 9](#_Toc9705884)

[***4.1.*** ***Segurança*** 9](#_Toc9705885)

[***4.2.*** ***Escalabilidade*** 9](#_Toc9705886)

[***4.3.*** ***Tolerância a falhas*** 9](#_Toc9705887)

[***5.*** ***Instruções de compilação*** 10](#_Toc9705888)

[***6.*** ***Conclusão*** 11](#_Toc9705889)

## ***Introdução***

No âmbito da unidade curricular de Sistemas Distribuídos, foi desenvolvido um sistema de backup de ficheiros para Internet. Para tal optamos por uma arquitetura que consiste num sistema centralizado com vários **PeerManager**, que têm como objetivo gerir as réplicas dos ficheiros, fornecendo informação aos **Peers**. O serviço suporta backup, restore e eliminação de ficheiros.

Neste relatório iremos primeiro apresentar a arquitetura do sistema, seguida da explicação da implementação dos protocolos e outros aspetos que consideramos importantes.

## ***Arquitetura do Sistema***

No que toca à arquitetura do sistema, existem três classes essenciais: **Peer**, **PeerManager** e **TestApp**.

O **Peer** representa um computador que se disponibiliza a fazer backup de ficheiros. É responsável por agir como intermediário entre o sistema e o utilizador, podendo fazer pedidos de backup, restauro e eliminação de ficheiros. Este é também responsável por responder a pedidos destes protocolos, guardando ficheiros em memória, acedendo a estes e eliminando-os quando pedido.

O **PeerManager** é o gestor da informação sobre a réplica dos ficheiros do sistema, contendo informação sobre os ficheiros existentes, quem os guarda, os *peers* existentes e os outros gestores existentes. É responsável por garantir que todos os gestores existentes no sistema se encontram atualizados, contendo toda a informação sobre o sistema.

A **TestApp** é a interface de comunicação com o utilizador. A partir desta é possível chamar os quatro protocolos existentes no sistema: **BACKUP**, **RESTORE**, **DELETE e RECLAIM**.

O tipo de comunicação utilizado no sistema é o RMI entre **Peer** e **TestApp** e **TCP**, com **JSSE** para conexões seguras, entre as restantes classes. Para tal, verificamos a necessidade de existir uma classe constantemente aberta ao longo do período de vida do **Peer** ou do **PeerManager**, cujo objetivo é aceitar conexões, o que foi desenvolvido na classe **ConnectionThread**.

A classe **ConnectionThread,** sempre que aceita um *socket*, cria uma nova classe: **AcceptConnectionThread**, encarregue de ler a mensagem recebida e de reencaminhar para uma nova *thread* que irá lidar com as mensagens. No caso do **PeerManager**, esta nova *thread* tem o nome de **ManagerMessageHandler**. No caso do **Peer**, é a *thread* **PeerMessageHandler**. Estas duas últimas classes são, então, responsáveis por tratar das ações subsequentes à receção de uma mensagem.

As mensagens em si estão organizadas na classe **Message**, que tem como objetivo tanto a construção das mensagens de acordo com os argumentos recebidos, como a destruição destas de acordo com os bytes recebidos por um *socket*.

As classes responsáveis pelo envio de mensagens inicial, essencial aos protocolos existentes, encontram-se nas classes **BackupThread**, **RestoreThread**, **DeleteThread** e **ReclaimThread**.

Existem ainda as classes **CheckActiveThread**, chamada apenas pelo **PeerManager**, e a classe **SendActiveThread**, chamada pelo **Peer**, que têm como objetivo a verificação de que um *peer* ainda se encontra ativo, como será explicado melhor na parte da implementação.

## ***Implementação do Sistema***

## ***Mensagens trocadas***

Nesta secção serão descritas as mensagens trocadas nos vários protocolos.

#### **Entrada no sistema**

Quando um **Peer** se junta ao sistema, envia ao **PeerManager** a seguinte mensagem:

JOIN <peer\_id> <address> <port> <CRLN><CRLN>

Em que *peer\_id* diz respeito ao identificador do *peer*, e *address* e *port* são o ponto de acesso ao *peer*, necessários para iniciar uma conexão com o mesmo.

Se o *peer* já tiver no seu sistema uma pasta chamada *peer<peer\_id>/backup*, que contenha ficheiros dentro, serão enviadas as mensagens **STORED** (descritas mais à frente) para cada um destes ficheiros, para o *peer* comunicar ao manager que tem estes ficheiros.

Como resposta ao pedido de entrada no sistema, o **PeerManager** envia uma mensagem de **MANAGER\_INFO**, descrita mais à frente nesta secção, a informar o **Peer** sobre os outros gestores existentes no sistema, o que lhe permite conectar-se com outro gestor quando o gestor ao qual se ligou inicialmente falha.

Quando um **PeerManager** se junta ao sistema já existente, ou seja, onde já existe outro **PeerManager**, este envia a seguinte mensagem com o seu ponto de acesso:

MANAGER\_JOIN <address> <port> <CRLN><CRLN>

Como resposta a esta última mensagem, o *PeerManager* recetor irá enviar toda a informação relativa a *peers*, outros gestores e ficheiros que contém, através das mensagens:

PEER\_INFO <num\_peers> <CRLN><CRLN>

Onde *num\_peers* é a quantidade de *peers* que existem, seguindo-se a informação de cada *peer*, que contém o seu identificador, ponto de acesso e número de ficheiros guardados:

PEER <peer\_id> <address> <port> <count\_files> <CRLN><CRLN>

Recebidos todos os *peers*, é enviada a informação sobre os ficheiros, sendo enviado numa primeira fase a seguinte mensagem que informa quantos ficheiros existem no sistema:

FILE\_INFO <num\_files> <CRLN><CRLN>

Segue-se, então, informação sobre cada ficheiro, que contém o seu identificador e os identificadores dos *peers* que o recebem:

FILE\_P <file\_id> <peer\_id\_1> <peer\_id\_2> … <CRLN><CRLN>

Finalmente, é enviada a mensagem que indica quais os outros gestores existentes no sistema, constando os respetivos pontos de acesso:

MANAGER\_INFO <address\_1> <port\_1> <address\_2> <port\_2> … <CRLN><CRLN>

Para avisar todos os outros gestores que existe um novo gestor na rede, existe a seguinte mensagem, que contém o ponto de acesso do novo gestor:

MANAGER\_ADD <address> <port> <CRLN><CRLN>

### ***Protocolo de Backup***

O protocolo de **BACKUP** inicia-se com o envio da seguinte mensagem do **Peer** para o **PeerManager**:

BACKUP <peer\_id> <rep\_degree> <file\_size> <CRLN><CRLN>

Em que *peer\_id* é o identificador do *peer* que faz o pedido, *rep\_degree* é o grau de replicação desejado e *file\_size* é o tamanho do ficheiro, o que permite ao gestor identificar se um *peer* tem espaço para fazer backup do mesmo. O **PeerManager** responde a esta mensagem com o seguinte:

AVAILABLE <address\_1> <port\_1> <address\_2> <port\_2> … <CRLN><CRLN>

Onde cada conjunto *address port* é um ponto de acesso a um *peer* que possa fazer backup do ficheiro. Os *peers* retornados são sempre aqueles que têm mais espaço em memória e que tenham espaço suficiente para guardar o ficheiro.

O *peer* vai então enviar a seguinte mensagem a cada peer recebido na mensagem *available*:

P2P\_BACKUP <file\_id> <num\_chunks> <CRLN><CRLN>

Onde *file\_id* diz respeito ao identificador do ficheiro a ser enviado e *num\_chunks* a quantidade de *chunks* que o ficheiro contém. A esta mensagem segue-se o envio dos *chunks* do ficheiro, pelo mesmo *socket*, sendo que é retornada uma mensagem “ACK” por cada *chunk* recebido.

Recebidos todos os *chunks*, o recetor guarda o ficheiro em *peer<id>/backup* e envia a mensagem **STORED** para o **PeerManager**, para o informar que está a fazer backup desse ficheiro:

STORED <peer\_id> <file\_id> <free\_space> <CRLN><CRLN>

O **PeerManager** guarda a informação que o *peer* com identificador igual a *peer\_id* tem uma réplica do ficheiro *peer\_id* e que tem *free\_space* de espaço livre para guardar ficheiros, terminando o protocolo.

### ***Protocolo de Restore***

O protocolo de restore inicia-se com o envio da seguinte mensagem do Peer para o PeerManager:

RESTORE <file\_id> <CRLN><CRLN>

Em que *file\_id* diz respeito ao identificador do ficheiro a ser restaurado. Como resposta, o **PeerManager** envia a mensagem de **AVAILABLE** apresentada no protocolo de backup em que são enviados os pontos de acesso dos *peers* que contêm uma réplica do ficheiro. O **Peer** liga-se, então a outro **Peer** através dos dados facultados pelo **PeerManager**, e manda a seguinte mensagem:

P2P\_RESTORE <file\_id> <CRLN><CRLN>

Caso o recetor não contenha o ficheiro, é enviada como resposta “ERR” e o emissor liga-se então a outro *peer* que contenha o ficheiro. Se tiver o ficheiro, o recetor envia a seguinte mensagem:

FILE <file\_id> <num\_chunks> <CRLN><CRLN>

Onde *num\_chunks* é o número de *chunks* que o ficheiro contém. Tal como no protocolo de backup, são enviados os *chunks* no mesmo *socket*, recebendo para cada um uma resposta “ACK” para assegurar que estão a ser recebidos os *chunks* com sucesso.

Quando todos os *chunks* foram recebidos, estes são guardados num ficheiro disponível na pasta *peer<id>/restore*. Caso algum erro ocorra, por falha do *peer* que esteja a enviar o ficheiro, o *peer* liga-se a outro dos recebidos na mensagem **AVAILABLE** e pede o ficheiro.

### ***Protocolo de Delete***

Para iniciar o protocolo de **DELETE**, é enviada a seguinte mensagem ao **PeerManager**:

DELETE <file\_id> <CRLN><CRLN>

FILE <file\_id> <num\_chunks> <CRLN><CRLN>

Que recebe como resposta a mensagem **AVAILABLE**, tal como nos protocolos anteriormente descritos, contendo informação sobre os **Peers** que contêm o ficheiro. É, então, enviada, para cada um dos *peers* aí presentes, a mesma mensagem de **DELETE**. Quando um *peer* elimina um ficheiro da sua memória, este envia ao **PeerManager** a mensagem:

DELETED <peer\_id> <file\_id> <free\_space> <CRLN><CRLN>

Onde *free\_space* é o espaço que o *peer* com identificador *peer\_id* tem para guardar ficheiros. Quando o ficheiro indicado não existe em mais nenhum *peer*, o **PeerManager** elimina-o da sua estrutura de dados que guarda informação sobre os ficheiros.

### ***Protocolo de Reclaim***

Para iniciar o protocolo de **RECLAIM**, é enviada a seguinte mensagem ao PeerManager:

RECLAIM <peer\_id> <free\_space> <CRLN><CRLN>

Que informa o gestor que o *peer* com identificador *peer\_id* tem *free\_space* bytes de espaço para guardar ficheiros. Este número pode ser negativo, o que significa que o *peer* terá que eliminar ficheiros durante o protocolo.

Caso o *peer* tenha ficheiros para eliminar, começa por fazer *backup* dos mesmos, enviando a seguinte mensagem ao **PeerManager**:

B\_RECLAIM <peer\_id> <file\_id> <rep\_degree> <file\_size> <CRLN><CRLN>

Esta mensagem assemelha-se à mensagem de **BACKUP** e serve para pedir ao gestor informação sobre os *peers* que tenham espaço para o backup do ficheiro e que ainda não possuam uma réplica do mesmo. O **PeerManager** responde com uma mensagem de **AVAILABLE** tal como nos outros protocolos.

O *peer* liga-se, então, a um dos pontos de acesso recebido na mensagem **AVAILABLE** e envia as mensagens de **P2P\_BACKUP** tal como no protocolo de *backup*, e o *peer* recetor trata de enviar a mensagem **STORED** para o **PeerManager**.

Após o *backup* do ficheiro estar completo, o *peer* elimina o ficheiro do seu sistema e envia a mensagem de **DELETED** ao **PeerManager**, informando-o que já não tem uma réplica desse ficheiro.

Quando o espaço livre do *peer* volta a ser um inteiro positivo, termina o protocolo.

### ***Verificação de Peers Ativos***

Existem uma mensagem adicional que tem como objetivo verificar se um *peer* está ativo no sistema. É enviada pelo **Peer**, a cada minuto, que informa o **PeerManager** que este ainda se encontra ativo:

ACTIVE <peer\_id> <CRLN><CRLN>

Ao receber esta mensagem o **PeerManager** irá atualizar o tempo em que o **Peer** mandou a mensagem de ativo pela última vez na sua estrutura de dados. A cada minuto e meio, o **PeerManager** vai verificar os tempos relativos à última mensagem de **ACTIVE** enviada por cada **Peer** e se esta for superior a dois minutos, ele elimina o **Peer** da sua base de dados. Para tal, é importante que quando os *peers* falham e voltam, que enviem informação sobre os ficheiros que possuem ao **PeerManager**. Tudo isto serve para garantir a integridade da informação guardada pelo **PeerManager** sobre o sistema.

### ***Mensagens entre PeerManagers***

Todas as mensagens recebidas por um **PeerManager** que alterem a informação que o mesmo contém são redirecionadas para os outros **PeerManagers**, sendo adicionado “\_M” ao fim da mensagem, para informar o recetor que não precisa de reenviar essa mensagem.

Estas mensagens são: **STORED\_M**, **DELETED\_M**, **RECLAIM\_M**, **JOIN\_M** e **ACTIVE\_M**.

## ***Concorrência***

O sistema garante concorrência através da utilização da classe **ScheduledThreadPoolExecutor**, para executar as *threads* correspondentes aos diferentes protocolos, tanto nos recetores como emissores das mensagens, e as conexões, e garantir concorrência.

As classes das designadas anteriormente que são *threads* incluem as quatro classes respetivas aos protocolos existentes: **BackupThread**, **RestoreThread**, **DeleteThread** e **ReclaimThread**, as classes que dizem respeito às conexões e à execução das ações consequentes da receção de mensagens: **ConnectionThread**, **AcceptConnectionThread**, **PeerMessageHandler**, **ManagerMessageHandler** e as classes que servem para verificar atividade dos *peers*: **CheckActiveThread**, **SendActiveThread**.

Para além dos *thread-pools* existentes, utilizamos também da classe **AsynchronousFileChannel** de **java.nio** para executar operações de Input/Output sem bloquear. Deste modo, a leitura e escrita de ficheiros essencial aos protocolos existentes utiliza este mecanismo sem bloqueio. Tal está implementado na classe **SaveFile**, nas funções *read()* e *write().*

## ***Protocolos***

As ações e mensagens trocadas na execução dos protocolos foi explicada no tópico respetivo à troca de mensagens, no entanto, existem ainda alguns pontos a referir. Os ficheiros são enviados como um todo, sendo separados em *chunks* apenas devido às limitações de bytes que podem ser enviados através de *SSLSockets*. Deste modo, todos os *chunks* são enviados na mesma conexão entre *peers*.

Sucintamente, no protocolo de **BACKUP**, o *peer* pede ao gestor os pontos de acesso de *peers* com espaço suficiente em memória para *backup* de ficheiros e envia o ficheiro para cada um destes. Quando os recetores fazem backup de um ficheiro, informam o gestor com uma mensagem de **STORED**, como descrito no tópico das mensagens de **BACKUP**.

No protocolo de **RESTORE**, o *peer* pede ao gestor os pontos de acesso de todos os *peers* que possuem réplicas do ficheiro pedido, enviando para um desses um pedido de **RESTORE** e recebendo o ficheiro. Se um *peer* falhar durante o envio do ficheiro o *peer* que pede o ficheiro pede-o ao seguinte *peer* na lista de pontos de acesso recebida.

No protocolo de **DELETE**, o *peer* pergunta ao gestor quais os pontos de acesso de todos os *peers* que possuem réplicas do ficheiro a eliminar e envia para cada um desses o pedido de eliminação do ficheiro. Cada um dos *peers* que elimina o ficheiro informa o gestor que o fez.

No protocolo de **RECLAIM**, o *peer* informa o gestor sobre o espaço que tem disponível. Se o seu espaço disponível ficar negativo, este tem de eliminar ficheiros, fazendo antes disso backup dos mesmos. Para cada ficheiro a eliminar, o *peer* pergunta ao gestor sobre *peers* que não tenham o ficheiro e que tenham espaço livre para o guardar, enviando o pedido de backup e eliminando o ficheiro, posteriormente, informando o gestor que o fez.

## ***Conexões seguras usando JSSE***

Utilizamos **JSSE** para garantir conexões seguras entre **Peers** e **PeerManagers**. Para tal, existem os ficheiros *keystore.jks* e *truststore.ts*, que contêm as chaves utilizadas e cuja palavra-passe é “password”, comum a ambos.

Utilizamos a classe **SSLContext**, para obter *SSLServerSocketFactory*, para cada um dos *peers* e gestores e todas as conexões utilizam a classe **SSLSocket** a variável *NeedClientAuth* foi colocada a true, sendo necessária deste modo a autenticação do cliente que quer fazer uma conexão. A classe **SSLServerSocketFactory** que origina um *SSLSocketFactory*, é necessária a ambos *peers* e gestores, pois todos eles podem receber pedidos externos.

# ***Aspetos relevantes***

## ***Segurança***

Foi utilizada a biblioteca JSSE para garantir comunicação segura, como explicado anteriormente.

## ***Escalabilidade***

No que toca a escalabilidade, como foi previamente dito, utilizamos *thread-pools* e operações de input e output assíncronas, para garantir escalabilidade a nível da implementação.

## ***Tolerância a falhas***

No que toca a tolerância a falhas, optamos por um sistema centralizado onde existem vários gestores da informação sobre as réplicas presentes no sistema. Cada Peer, ao juntar-se ao sistema, recebe informação sobre todos os gestores existentes no sistema. Se um gestor falhar, o peer liga-se automaticamente a outro quando necessita de mandar uma mensagem ao sistema.

Quando um peer falha, os gestores eliminam a sua informação da base de dados, sendo que quando este retorna este informa-os sobre que ficheiros ele tem guardado na sua pasta relativa a backup. Se quando o peer voltar, o seu identificador original estiver ocupado por outro peer que entretanto se juntou ao sistema, o peer deve entrar no sistema com outro identificador, alterando o nome da pasta onde guarda os ficheiros de backup (peer<id>/backup), atualizando-a com o seu novo identificador.

A nível do protocolo de Restore, quando um peer falha enquanto está a passar um ficheiro a outro, o outro peer, que quer o ficheiro, conecta-se a outro peer que contenha o ficheiro.

# ***Instruções de compilação***

Para compilar o programa, deve-se compilar com a instrução: ***javac \*.java***. Para facilitar a compilação, incluímos o ficheiro ***“compile.sh”*** e ***“make.bat”*** que compilam o programa. Em Ubuntu, deve-se executar a instrução ***sh compile.sh***, e em Windows a instrução ***make.bat***. Deve ser chamada também a instrução: ***rmiregistry &***, visto que é utilizado RMI na comunicação entre as classes **TestApp** e **Peer**.

Para correr o primeiro **PeerManager**, usa-se a instrução:

java -Djavax.net.ssl.keyStore=keystore.jks -Djavax.net.ssl.keyStorePassword=password -Djavax.net.ssl.trustStore=truststore.ts -Djavax.net.ssl.trustStorePassword=password PeerManager <port>

Onde *port* diz respeito à porta de acesso ao gestor. Para o segundo e restantes gestores, adiciona-se o ponto de acesso de um **PeerManager** existente no sistema:

java -Djavax.net.ssl.keyStore=keystore.jks -Djavax.net.ssl.keyStorePassword=password -Djavax.net.ssl.trustStore=truststore.ts -Djavax.net.ssl.trustStorePassword=password PeerManager <port> <manager\_address> <manager\_port>

Para correr um **Peer**, usa-se a instrução:

java -Djavax.net.ssl.trustStore=truststore.ts -Djavax.net.ssl.trustStorePassword=password -Djavax.net.ssl.keyStore=keystore.jks -Djavax.net.ssl.keyStorePassword=password Peer <id> <remote\_object\_name> <port> <manager\_ip> <manager\_port>

Onde *id* é um inteiro que representa o identificador do *peer*, *remote\_object\_name* é o nome do objeto utilizado para a comunicação rmi, *port* é a porta de acesso ao peer, *manager\_ip* é o endereço *ip* do **PeerManager** ao qual o **Peer** se irá ligar e ***manager\_port*** é a porta de acesso deste **PeerManager**.

Para correr a aplicação e executar os protocolos chama-se a classe TestApp:

java TestApp <address>:<remote\_object\_name> <protocol> <opnd\_1 > <opnd\_2 >

Onde ***address : remote\_object\_name*** são o endereço *ip* e o nome do objeto do **Peer** ao qual o cliente se quer ligar, *protocol* é o nome do protocolo a executar: **BACKUP**, **RESTORE**, **DELETE** ou **RECLAIM**, *opnd\_1* é o nome do ficheiro que se quer restaurar, eliminar ou do qual se quer fazer *backup*, ou o número de bytes máximo disponível para guardar ficheiros, no caso do protocolo de **RECLAIM**, *opnd\_2* é o grau de replicação do ficheiro quando se quer fazer backup e existe apenas no caso do protocolo de **BACKUP**.

# ***Conclusão***

O trabalho realizado ajudou-nos a compreender melhor como é possível criar um sistema distribuído para a Internet, com comunicação segura.

Foram desenvolvidos protocolos de backup, restauro e eliminação de ficheiros, tendo sido implementada uma solução com concorrência que permite tolerância a falhas, segurança na comunicação e escalabilidade, recorrendo ao uso de diversas ferramentas como JSSE, thread-pools e operações de input e output assíncronas.