Relatório Final

Sistemas Distribuídos
2017/2018

Mestrado Integrado em Engenharia Informática e Computação

Sistema Distribuído de Download e Upload de Ficheiros

Bárbara Sofia Silva up201505628@fe.up.pt
Carlos Miguel Freitas up201504749@fe.up.pt
Julieta Frade up201506530@fe.up.pt
Luís Martins up201503344@fe.up.pt

27 de Maio de 2018

Conteúdo

1 Introdução

Este projeto tem como finalidade a implementação de um sistema distribuído de *download* e *upload* de ficheiros, do estilo ***BitTorrent***.

Como foi proposto na especificação, o grupo escolheu melhorar o primeiro projeto, que consistia num serviço de *backup* distribuído sem servidor para uma rede local. Posto isto, foi melhorada a segurança, a tolerância a falhas e a confiança do sistema.

Assim, neste segundo projeto, foi feita uma ligeira alteração à arquitetura, optando agora por um sistema distribuído ***peer-to-peer*** com ***tracker***, onde cada *peer* pode fazer *seed* de ficheiros. Em relação ao *peer*, este tanto pode funcionar simultaneamente como cliente e como servidor, ou seja, pode fazer *download* de ficheiros que pretende, ao mesmo tempo que disponibiliza à rede os ficheiros que tem e está a dar *seed*.

Quanto a relatório, o seu objetivo é expor e explicar toda a componente teórica presente neste segundo trabalho, tendo a seguinte estrutura:

- **Arquitetura**: Exibição de blocos funcionais e interfaces presentes.
- **Implementação**: Descrição e explicação detalhada de toda a implementação.
- **Aspetos Relevantes**: Exposição de funcionalidades relevantes do projeto.
- **Conclusão**: Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

2 Arquitetura

Esta aplicação é constituída por vários peers e um tracker. O tracker é responsável por manter o estado do sistema, nomeadamente manter que peers estão ativos e que ficheiros cada peer mantém em seed.

O tracker é fundamental para manter a integridade do sistema, pois é este elemento central que gere a que peers poderá um outro peer pedir o ficheiro. Funciona assim como servidor de estado, pois mantém o estado do sistema, sendo imprescindível para que tudo funcione.

Já relativamente a cada peer, este é responsável por ao mesmo tempo fazer download dos ficheiros pretendidos pelo cliente e responder a pedidos de chunks de outros ficheiros aos quais está a dar seed (peer 2 peer)

Assim este projeto está dividido em 4 packages: peer, Sockets, Messages e view.

2.1 peer

Este package é responsável por gerir todas as interações existentes entre o peer e o sistema, aqui também estão as funções relativas à leitura e escrita de ficheiros bem como a sua criação.

2.2 Sockets

Este package é responsável pela criação dos sockets de ligação segura de protocolo TLS com o uso de SSLSockets da biblioteca JSSE.

2.2.1 SecureSocket

Classe mãe que representa o socket seguro. Contém a porta de ligação, dois KeyStores (onde ficam "guardadas"a chave privada e a chave pública de forma a autenticar de forma segura uma ligação) e um SSLContext (que funciona como fábrica para gerar Sockets seguros).

2.2.2 ReceiverSocket

É um SecureSocket que se comporta como um servidor.Este é constituído por um SSL-ServerSocket (gerado a partir do SSLContext) que está aberto na porta especificada.

O SSLServerSocket fica infinitamente à espera de ligações, se uma conexão for estabelecida o SSLServerSocket cria um SSLSocket que é mandado para uma thread do tipo MessageHandler que lida com as trocas de mensagens nessa ligação.

2.2.3 SenderSocket

Dito isto, este package é constituído por dois tipos de sockets seguros, o ReceiverSocket e o SenderSocket. O ReceiverSocket contém um SSLServerSocket comportando-se como um servidor que cria uma thread que fica infinitamente a espera de conexões. O SenderSocket comporta-se como cliente e tenta se ligar (sabendo a port e o host) ao servidor através de um SSLSocket que é gerado a partir de um SSLSocketFactory.

2.3 Messages

Esta package é constituído pelas mensagens que são trocadas nas comunicações entre sockets.

Message

Classe mãe que representa uma mensagem de comunicação

OnlineMessage

Mensagem mandada periodicamente pelo peer de forma a confirmar ao tracker que se encontra na rede. A mensagem encontra-se estruturada da seguinte forma:

- ONLINE <senderID><CRLF><CRLF>

RegisterMessage

Mensagem de registo no sistema. Esta é enviada ao tracker de forma a ele guardar as informações necessárias (IP,porta e chave pública) para posteriormente serem estabelecidas ligações de outros peers com quem mandou esta mensagem. A mensagem encontra-se estruturada da seguinte forma:

- REGISTER <senderID><address><port><CRLF><CRLF><publicKey>

HasFileMessage

Mensagem do peer que informa ao Tracker que esta disposto a dar seed ao ficheiro especificado. A mensagem encontra-se estruturada da seguinte forma:

- HASFILE <senderID><fileID><CRLF><CRLF>

peerInfoMessage

Mensagem enviada pelo Tracker a um peer na resposta a uma mensagem GetFileMessage. Esta mensagem contém a informação necessária para ser possível ligação entre peers. A mensagem encontra se estruturada da seguinte forma:

- peerINFO <fileID><address><port><CRLF><CRLF><publicKey>

peerInfoEndMessage

Mensagem escrita após o envio de todas as mensagens peerInfoMessages de forma a indicar que não há mais peers disponíveis para dar seed ao ficheiro. A mensagem encontra-se estrutura da seguinte forma:

- peerINFOEND <fileID><CRLF><CRLF>

GetFileMessage

Mensagem direcionada ao Tracker, de forma ao peer saber a quem pode pedir o ficheiro. A mensagem encontra-se estruturada da seguinte forma:

- GETFILE <senderID><fileID><CRLF><CRLF>

GetChunkMessage

Mensagem de pedido de um pedaço de ficheiro. A mensagem encontra-se estruturada da seguinte forma:

- GETCHUNK <fileID><chunkNr><CRLF><CRLF>

ChunkMessage

Mensagem de resposta a um pedido GetChunk com o pedaço de ficheiro pedido. A mensagem encontra-se estruturada da seguinte forma:

- CHUNK <fileID><chunkNr><CRLF><CRLF><chunkData>

3 Implementação

Para implementação dos protocolos existentes neste trabalho recorremos à utilização de várias ferramentas

3.1 Concorrência através de Multithreading

Para aumentar ao máximo a possibilidade de concorrência entre processos e, deste modo, diminuir o tempo necessário para que os peers consigam obter os ficheiros desejados recorreu-se então à utilização de multithreading. Esta técnica consiste na execução de tarefas paralelamente de forma a que o tempo de execução global seja diminuido.

Deste modo de cada vez que um peer (cliente) pretende comunicar com outro peer(servidor) para enviar um ou vários pedidos para obter chunks de um ficheiro, é então criada uma thread que se responsabiliza por esta comunicação criando um socket como ponto final o serverScoket existente no peer servidor.

Posteriormente, é através desta ligação que o cliente fará todos os pedidos e receberá as respetivas respostas do servidor.

Já no que toca à parte do peer enquanto servidor este manda uma thread a executar constantemente que mantém aberto um serverSocket por forma a que outros peers se possam ligar e pedir chunks de ficheiros que o peer servidor esta a dar seed.

3.2 Lease

Para manter consistência no tracker relativamente aos peers que estão atualmente ativos no sistema e quais os ficheiros que cada um está a fazer seed, o sistema utiliza então a técnica de lease. Esta técnica permite que o tracker esteja quase sempre atualizado sem que exista um excesso de mensagens trocadas. Lease é um "contrato"que cede ao contratador os direitos para utilizar/gerir algum recurso por um período limitado de tempo. No caso deste projeto, esta técnica foi utilizada entre o tracker e os peers através do envio da mensagem *Online*. Assim, um peer a cada minuto reenvia uma mensagem de *Online* para o tracker a confirmar que ainda está ativo. Deste modo o tracker por cada mensagem *Online* recebida faz reset ao tempo de lease do peer em questão. Por outro lado, a cada 2 minutos, o tracker verifica o tempo de lease que cada peer tem, apagando a informação relativa aos peers cujo tempo está expirado. Assim, garante-se que o tracker está atualizado mantendo-se a consistência da rede.

3.3 Protocolo de Seed

Quando um peer se encontra ligado à rede, este pode fazer seed de um ficheiro, ou seja, ele dispõe-se a fazer upload do ficheiro indicado aos peers que o pedirem.

Se um peer indicar que quer fazer seed de um certo ficheiro é criado um ficheiro do tipo de XML que contém as informações necessárias para outro peer poder fazer download desse mesmo ficheiro. O ficheiro XML de download encontra se estruturado da seguinte maneira:

```
<torrent>
    <tracker IP="trackerIP" port="trackerPort"></tracker>
    <Chunk length="chunkSize"></Chunk>
    <File ID="fileID" name="fileName" length="fileLength"></File>
</torrent>
```

Portanto se um peer quiser fazer download de um ficheiro necessita do ficheiro XML para iniciar este protocolo.

Após a criação do ficheiro XML, o peer informa ao Tracker (através da mensagem *HASFILE*) que está disposto a fornecer esse ficheiro a quem o pedir. Assim se um peer quiser fazer download de um ficheiro quando lê o XML, o Tracker pode informa-lo acerca dos peers a quem ele pode pedir esse ficheiro.

3.4 Protocolo de Download

Um peer ao ter acesso a um ficheiro XML gerado por outro peer que fez o protocolo de seed, pode iniciar o protocolo de Download. Note-se que, após um peer fazer o download do ficheiro, ele passa a estar disposto a dar seed do mesmo.

Assim, numa fase inicial,após o parse do ficheiro XML, o peer começa por mandar uma mensagem do tipo *GETFILE* ao Tracker com o ID do ficheiro presente no XML. O Tracker lê a mensagem e identifica os peers que se encontram disponíveis a dar seed desse ficheiro, mandando como resposta ao *GETFILE* um conjunto de mensagens *peerINFO* que informa ao peer requerente quais os peers a quem este deve pedir os ficheiros. Após enviar todas as mensagens *peerINFOs* necessários, o Tracker envia uma nova mensagem *peerINFOEND* de forma a indicar o fim da comunicação. Numa segunda fase, o peer requerente tenta pedir ao máximo número de peers disponíveis os chunks através de mensagens *GETCHUNK*. Por cada mensagem *GETCHUNK* que o peer "servidor"recebe é lido assíncronamente do ficheiro a chunk pretendida e a mesma é mandada como resposta numa mensagem *CHUNK*. O cliente ao receber o *CHUNK*,

sabe o seu offset no ficheiro a transferir, logo escreve direta e assíncronamente no mesmo a informação na posição exata. É importante referir que um peer cliente nunca recebe a mesma chunk pedida mais do que uma vez. Por fim, após acabar o download do ficheiro, o peer cliente manda uma mensagem *HASFILE* a informar ao tracker que esta disponível a dar upload ao ficheiro obtido.

3.5 Conexões seguras com uso de SSLSockets

De forma a proporcionar o melhor nível de segurança na troca de comunicações peer to peer foram usados SSLSockets da biblioteca JSSE do Java com o uso de chaves públicas.Deste modo, sempre que é ligado um peer pela primeira vez, são geradas chaves públicas e privadas únicas a esse peer, em que o seu identificador é gerado automaticamente através do uso da RandomUUI.As chaves são formadas através dos seguintes comandos:
Primeiro é gerado a chave privada:

```
keytool -genkey -alias peerNameprivate -keystore peerName.private
-storetype JKS -keyalg rsa -dname "CN=Your Name, OU=Your
Organizational Unit, O=Your Organization, L=Your City, S=Your State,
C=Your Country" -storepass peerNamepw -keypass peerNamepw
```

Numa segunda fase é então extraído da chave privada criada, uma chave pública:

```
keytool -export -alias peerNameprivate -keystore peerName.private -file
    temp.key -storepass peerNamepw;
keytool -import -noprompt -alias peerNamepublic -keystore peerName.public
    -file temp.key -storepass public;
rm -f temp.key
```

Estas chaves são sempre geradas em todos os peers, exceto no Tracker.Como o Tracker é único e necessário, as suas chaves são pré geradas e todos os peers que se ligam têm acesso à sua chave pública de forma a haver logo de início uma comunicação segura entre os peers e o Tracker.

Após serem geradas as keys, cada peer (como também o Tracker) inicia o seu ReceiverSocket, que funciona como servidor com um SSLServerSocket aberto na primeira porta disponível:

```
controlReceiver = new ReceiverSocket(0);
controlReceiver = connect(peerID);
```

O ReceiverSocket antes de gerar o SSLServerSocket através de um SSLContext necessita de estipular as KeyStores necessárias para estabelecer a ligação.Como ficou decidido a autenticação ser apenas do servidor, para um ReceiverSocket só é necessario gerar a KeyStore do servidor e não do cliente,ou seja, só necessita de ter acesso à sua chave privada. Deste modo a KeyStore é gerada da seguinte maneira:

```
public void setupSocketKeyStore(String peerName) throws
    GeneralSecurityException, IOException{
        this.socketKeyStore = KeyStore.getInstance("JKS");
        String filename;
        if(peerName.equals("tracker")){
            this.setPassphrase("tracker");
        }
        this.socketKeyStore.load(new FileInputStream(filename),
                                 this.passphrase.toCharArray());
    }
```

Após a geração da KeyStore é possivel criar o SSLContext, da seguinte maneira:

```
public void setupSSLContext() throws GeneralSecurityException, IOException{
        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init(publicKeyStore);

        KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
        kmf.init(socketKeyStore,passphrase.toCharArray());

        this.sslContext = SSLContext.getInstance("TLS");
        this.sslContext.init(kmf.getKeyManagers(),
                             tmf.getTrustManagers(),
                             secureRandom);
    }
```

Por fim, um ReceiverSocket é então iniciado da seguinte maneira:

```
public void connect(String connectFrom) {
        try {
            setupSocketKeyStore(connectFrom);
            setupSSLContext();

            SSLServerSocketFactory sf = sslContext.getServerSocketFactory();
            this.serverSocket = (SSLServerSocket)
                sf.createServerSocket(this.port, 2);
```

```
            // Require client auth
            this.serverSocket.setNeedClientAuth(false);
            System.out.println("Listening on " + serverSocket.getLocalPort());
            Runnable accepter;
            if(connectFrom.equals("tracker")){
                accepter = new connectionAccepter(true);
                Tracker.getExec().execute(accepter);
            }else{
                    accepter = new connectionAccepter(false);
                    peer.getExec().execute(accepter);

            }
        } catch (GeneralSecurityException ge) {
            ge.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public class connectionAccepter implements Runnable {

        boolean isTracker;
        public connectionAccepter(boolean isTracker) {this.isTracker =
            isTracker;}

        @Override
        public void run() {
            while (true) {


                if(!isTracker){
                    if(peer.getReceiverCounter() >= 10){
                        continue;
                    }
                }


                try {
                    SSLSocket socketConnected = (SSLSocket)
                        serverSocket.accept();
                    //System.out.println("Got connection from " +
                        socketConnected);
                    MessageHandler handler = new
                        MessageHandler(socketConnected);
```

```
                handler.updateState(Transition.RECEIVER);
                if(isTracker)
                    Tracker.getExec().execute(handler);
                else{
                    peer.incReceiverCounter();
                    peer.getExec().execute(handler);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

        }

    }
}
```

Como cada peer funciona simultaneamente como cliente e servidor, de forma a um peer estabelecer um canal TCP com outro peer é necessário o acesso à sua chave pública. Para tal, o Tracker mantém em si guardado a informação de cada peer essencial para poder estabelecer uma conexão, tal como, o endereço de IP, a porta em que está aberto e a sua chave pública.Quando um peer se liga, envia ao Tracker as suas informações e este as guarda. Assim sendo, um peer necessita de fazer um pedido ao Tracker para iniciar uma comunicação com outro peer.

Uma comunicação é estabelecida através de um SenderSocket que funciona de forma semelhante ao ReceiverSocket só que gera um SSLSocket e necessita,para além da KeyStore da sua chave privada, a KeyStore pública do servidor, que é gerada da seguinte maneira:

```
public void setupP2PPublicKeyStore(byte[] key)throws
    GeneralSecurityException, IOException{
        this.publicKeyStore = KeyStore.getInstance("JKS");
        String publicpw = "public";
        this.publicKeyStore.load(new
            ByteArrayInputStream(key),publicpw.toCharArray());

    }
```

Com isto podemos garantir que todas as comunicações feitas no nosso sistema distribuído são seguras.

4 Aspetos Relevantes

4.1 Segurança

Foi utilizado o protocolo de segurança TLS (Transport Layer Security) para todas as comunicações com a ajuda da biblioteca JSSE do Java através de SSLSockets.

4.2 Consistência

Tal como referido anteriormente a técnica de Lease permite que o sistema se mantenha consistente, pois garante que o conhecimento do Tracker sobre o sistema é, em grande parte do tempo, o mais atualizado possível.Para além desta técnica, a utilização de conexões TCP garante que as mensagens são recebidas pelo respetivo destinatário.

4.3 Tolerância a falhas

De forma a garantir ao máximo a estabilidade do sistema, são feitas algumas verificações relativamente ao download e upload dos ficheiros.Assim, são tidos em considerações os seguintes casos:

- Falha do peer cliente

  Caso um peer que está a fazer download de um ou mais ficheiros falhe, este automaticamente se desconecta dos peers que lhe estão a fornecer os ficheiros, sendo guardado, através de serialização, o seu estado aquando desta falha.Esta serialização contém informação acerca dos ficheiros que o peer estava a fazer download e disponível a fazer upload. Assim, quando este mesmo peer recuperar da será possivel retomar os serviços interrompidos aquando da falha, nomeadamente tentar terminar o download dos ficheiros que ficaram pendentes.

- Falha do peer servidor

  Caso um peer que está a servir outros peers falhe, as suas ligações são automaticamente terminadas. Com esta ação os peers que estabeleceram conexão com o peer que falhou conseguem tratar as exceções lançadas por esta falha. Assim, quando um peer deteta uma falha no servidor, descarta a última mensagem caso ela não tenha sida recebida totalmente,considerando a seguir apenas como pedido os peers servidores que estão disponíveis. Caso o peer cliente não conheça nenhum peer a quem possa pedir um ficheiro possíveis, o peer cliente pede ao Tracker se existe peers com quem ele possa comunicar.

5 Conclusão

Todo o desenvolvimento deste projeto contribuiu para uma melhor compreensão do funcionamento de um sistema distribuído, tendo sido finalmente aplicada a matéria lecionada nas as aulas. Utilizando o uso de leases garante que o sistema seja o mais consistente possível. É garantido alguma tolerância de erros, bem como a falha nas comunicações e a recuperação as mesmas. Como melhoramento, seria interessante implementar uma versão *trackerless* do sistema, que não foi feita devido à falta de tempo e complexidade. Também seria interessante implementar um servidor local que guardasse todos os ficheiros "torrents"criados, de forma a que um peer possa fazer download dos torrents disponíveis de forma a fazer iniciar a transferência de um ficheiro. Em suma, o projeto foi concluído com sucesso, tendo-se cumprido todos os objetivos e ultrapassado todos os obstáculos.