Relatório Final

Sistemas Distribuídos
2019/2020

Mestrado Integrado em Engenharia Informática e Computação

Distributed Backup Service for the Internet

Diogo Ferreira de Sousa - up201706409@fe.up.pt
João Pedro Pinto Mota - up201704567@fe.up.pt
Margarida Alves Pinho - up201704599@fe.up.pt
Maria Gonçalves Caldeira - up201704507@fe.up.pt

29 de Maio de 2020

Índice

1. Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Sistemas Distribuídos e teve como finalidade a implementação de um sistema distribuído *peer-to-peer* de um serviço de *backup* para a Internet. À semelhança do primeiro projeto, este serviço também suporta o *backup*, *restore* e *deletion* de ficheiros. Com este relatório pretende-se explicar a nossa abordagem e descrever o design implementado que permite a execução simultânea de protocolos.

2. Visão geral

Para este projeto decidimos implementar um sistema centralizado, que utiliza a arquitetura do tipo cliente/servidor, onde um ou mais peers estão conectados a um servidor central. O servidor tem a função de gerir os peers e os seus pedidos tornando-se então fundamental para o bom funcionamento do sistema.

```java
public class Server {
    static public int server_port;
    static public ArrayList<String[]> peers; // format is {"id" "ip" "port"}

    public static void main(String[] args) throws Exception {
        parse_args(args);
        peers = new ArrayList<String[]>();

        Server_coms coms = new Server_coms();
        (new Thread(coms)).start();
    }

    private static void parse_args(String[] args) throws Exception {
        if (args.length != 1) {
            throw new Exception("Usage: <server port>");
        } else {
            Server.server_port = Integer.parseInt(args[0]);
        }
    }

}
```

```java
class Server_coms implements Runnable {
    protected SSLServerSocketFactory server_factory;

    public Server_coms() {
        this.server_factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
    }

    public void run() {
        SSLServerSocket server;

        try {
            server = (SSLServerSocket) this.server_factory.createServerSocket(Server.server_port);

            System.out.println(
                    "Started server at " + server.getInetAddress().getHostAddress() + ":" + server.getLocalPort());

            while (true) {
                SSLSocket socket = (SSLSocket) server.accept();
                DataInputStream in = new DataInputStream(socket.getInputStream());
                DataOutputStream out = new DataOutputStream(socket.getOutputStream());

                String[] received = in.readUTF().split(" ");

                if (received[0].equals("HI")) {
                    this.addPeer(received[1], socket.getInetAddress(), received[2]);
                } else if (received[0].equals("GETPEERS")) {
                    out.writeUTF(this.getPeersString(received[1]));
                    out.flush();
                }

                socket.close();
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

3. Protocolos

Usamos RMI para procedermos às implementações do *backup*, *restore*, *delete*, *reclaim* e *state*. Para todos esses serviços a nossa implementação inicializa o controlador de cada um e adiciona-o à lista de controladores do respetivo comando.

Exemplo para o comando BACKUP:

```
public class RMI_connection implements RMI_interface {
    private Controller controller;

    public RMI_connection(Controller controller) {
        this.controller = controller;
    }

    @Override
    public String backup(String path, int repDregree) throws RemoteException {
        Backup_controller backup_controller = new Backup_controller(this.controller, this.controller.get_listener(),
                path, repDregree);
        System.out.println("Starting backup protocol");
        try {
            this.controller.addAction_controller(backup_controller);
        } catch (Exception e) {
            e.printStackTrace();
        }

        (new Thread(backup_controller)).start();
        return "Backing up file";
    }
```

No que toca ao TCP, podemos dar o exemplo da classe *Peer* em que para que possam ser trocadas mensagens o servidor tem que ser inicializado e criar uma nova *socket* para a comunicação. As mensagens analisadas nesta parte do código são do tipo “"HI " + App.id + " " + App.local_port ” (onde App.id é o id do peer e o App.local_port é o App.id + Appport_offset = 51234) ou do tipo “"GETPEERS " + App.id”. A partir da primeira palavra das mensagens, numa das funções do *Server* são feitas algumas ações mostradas na figura abaixo.

```
try {
    server = (SSLServerSocket) this.server_factory.createServerSocket(Server.server_port);

    System.out.println(
            "Started server at " + server.getInetAddress().getHostAddress() + ":" + server.getLocalPort());

    while (true) {
        SSLSocket socket = (SSLSocket) server.accept();
        DataInputStream in = new DataInputStream(socket.getInputStream());
        DataOutputStream out = new DataOutputStream(socket.getOutputStream());

        String[] received = in.readUTF().split(" ");

        if (received[0].equals("HI")) {
            this.addPeer(received[1], socket.getInetAddress(), received[2]);
        } else if (received[0].equals("GETPEERS")) {
            out.writeUTF(this.getPeersString(received[1]));
            out.flush();
        }

        socket.close();
    }
```

4. Execução simultânea de protocolos

O design implementado no projeto permite a execução simultânea de protocolo uma vez que recorre à utilização de várias threads.

A classe App (que inicia os peers) quando é chamada processa os argumentos da linha de comando e cria um objeto da classe Controller que gere a execução do novo peer, criando o seu fileSystem. Este objeto executa uma thread para comunicação e outra para cada ação “externa” (iniciada por outro peer) e para cada controlador de ação inicializada pelo próprio.

```
public class Controller implements Runnable {
    public final static int chunkSize = 64000;

    private ArrayList<Action_request> actions;
    private ArrayList<Backup_controller> backup_controllers;
    private ArrayList<Restore_controller> restore_controllers;
    private ArrayList<Delete_controller> delete_controllers;
    private Listener action_listener;
    private Storage_controller storage;
    public int id = 0;

    public Controller(int id) throws Exception {
        this.id = id;

        this.storage = new Storage_controller(this, id);

        // initializes actions "pipe"
        this.actions = new ArrayList<Action_request>();
        this.backup_controllers = new ArrayList<Backup_controller>();
        this.restore_controllers = new ArrayList<Restore_controller>();
        this.delete_controllers = new ArrayList<Delete_controller>();

        // creates a thread for each mc channel listener (so far the 3 classes could be
        // reduced to 3 instances of the same)
        this.action_listener = new Listener(this);
        (new Thread(this.action_listener)).start();

        // rmi stuff
        RMI_connection rmi = new RMI_connection(this);
        RMI_interface stub = (RMI_interface) UnicastRemoteObject.exportObject(rmi, 0);

        // bind to registry
        Naming.rebind(App.access_point, stub);
    }
```

Na classe TestApp é feito o parser do comando executado, de seguida é criado um objeto da classe RMIconnection que ao ser inicializado cria uma thread para o protocolo

escolhido, tornando possível serem executados vários ao mesmo tempo, excepto para o serviço STATE. Na imagem em baixo podemos ver o excerto de código executado para o comando BACKUP.

```
public class RMI_connection implements RMI_interface {
    private Controller controller;

    public RMI_connection(Controller controller) {
        this.controller = controller;
    }

    @Override
    public String backup(String path, int repDregree) throws RemoteException {
        Backup_controller backup_controller = new Backup_controller(this.controller, this.controller.get_listener(),
                path, repDregree);
        System.out.println("Starting backup protocol");
        try {
            this.controller.addAction_controller(backup_controller);
        } catch (Exception e) {
            e.printStackTrace();
        }

        (new Thread(backup_controller)).start();
        return "Backing up file";
    }
```

5. Tolerância a erros

Na nossa implementação optámos por criar apenas um servidor que é usado para ir buscar os endereços e portas dos clientes, não interferindo na comunicação em si. Deste modo, a probabilidade dele falhar é muito mais reduzida.

6. JSSE

No nosso trabalho recorremos ao uso de *SSLSockets*. Este é usado nas classes *Server* e *Listener*. Na classe *Server* este é usado para ir buscar o nosso *server socket* (com *SSLServerSocketFactory*), assim como no *Listener*, estando ligado à inicialização do nosso principal e único servidor. O *SSLServerSocket* e o *SSLSocket* também são utilizados no *Server* e no *Listener*, enquanto que o *SSLSocketFactory* é só usado no *Listener.*

```
class Server_coms implements Runnable {
    protected SSLServerSocketFactory server_factory;

    public Server_coms() {
        this.server_factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
    }

    public void run() {
        SSLServerSocket server;

        try {
            server = (SSLServerSocket) this.server_factory.createServerSocket(Server.server_port);
```

```java
public class Listener implements Runnable {
    protected InetAddress socket_addr;
    protected int socket_port;
    protected Controller controller;
    protected SSLServerSocketFactory server_factory;

    public Listener(Controller controller) throws Exception {
        this.controller = controller;
        this.server_factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

        // says HI to the server
        SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
        SSLSocket socket = (SSLSocket) factory.createSocket(App.server_addr, App.server_port);
        socket.startHandshake();
        DataOutputStream out = new DataOutputStream(socket.getOutputStream());
        out.writeUTF("HI " + App.id + " " + App.local_port);
        out.flush();
    }

    @Override
    public void run() {
        SSLServerSocket server;

        try {
            server = (SSLServerSocket) this.server_factory.createServerSocket(App.local_port);

            while (true) {
                SSLSocket socket = (SSLSocket) server.accept();
                socket.startHandshake();
                DataInputStream in = new DataInputStream(socket.getInputStream());
```

7. Escalabilidade

O nosso trabalho tem io assíncrona porque cada protocolo tem uma thread independente.