Project documentation

Distributed Systems

Group 01

Sergio Anguita Lorenzo

Aitor Brazaola Vicario

Este tercer entregable tiene por objeto completar la funcionalidad del sistema incorporando la implementación del protocolo UDP del Tracker y la funcionalidad de gestión y almacenamiento de los peers en el repositorio de información. Dicha funcionalidad debe cubrir los siguientes aspectos:

Implementación la parte del tracker del protocolo bitTorrent peer-tracker en su versión UDP

Implementación de la funcionalidad de almacenamiento persistente de la información de los swarms activos. La información se almacenará únicamente mientras el tracker esté activo. Por lo tanto, habrá que borrar completamente el repositorio de información al "parar ordenadamente" el tracker.

Gestión de la coordinación a la hora de almacenar la información en el repositorio de información.

Más información: https://alud.deusto.es/mod/resource/view.php?id=121249

Implementation of UDP peer-tracker interaction

For this purpose, each tracker has an UDP server listening on a random port between **1000** and **5000**. This UDP will be listening for incoming connections from clients.

Implemented UDP server follow next code style:

```
//server initialization
udpServer = new TrackerUDPServer();
udpServer.backgroundDispatch();
```

A random port between **1000** and **5000** is selected when deploying the UDP server. Next line, selects the port:

```
this.port = MIN_PORT + (int)(Math.random() * ((MAX_PORT - MIN_PORT) + 1));
```

And UDP server main follows this code:

```
private void startServer() {
    try {
        ServerSocket welcomeSocket = new ServerSocket(port);
        System.out.println("socket online");
        System.out.println("Waiting for clients to connect...");

        handler = new UDPThread(this, welcomeSocket);
        handler.start();

    } catch (IOException e) {
        System.err.println("Error handling client request.");
        System.err.println("Error details: "+e.getLocalizedMessage());
    } catch (Exception e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
}
```

AS you can see, **UDPThread** class is is charge of receiving and parsing all incoming **UDP** messages. This is done in separate class following the conventions CCD (Clean Code Development)

UDPThread class inherits from **Thread** and is the one that is in charge of listening UDP messages. This class is as follows:

Class Constructor:

```
private boolean active;
private ServerSocket soc;
private TrackerUDPServer ms;

public UDPThread(TrackerUDPServer mainServer, ServerSocket soc) {
    this.soc = soc;
    active = true;
    ms = mainServer;
}
```

Thread run() method.

```
@Override
public void run() {
    while (active) {
        SocketManager sockManager;
        try {
            sockManager = new SocketManager(soc.accept());
            Request request = new Request(ms, sockManager);
            Thread thread = new Thread(request);
            thread.start();
            // soc.close();
        } catch (Exception e) {
            System.err.println(e.getLocalizedMessage());
        }
    }
    //service stopped. close opened streams
    try {
        soc.close();
    } catch (IOException e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
    System.out.println("UDP THread stopped");
}
```

And finally, **Request** class the one that reads line by line, received data from UDP stream. The code of **Request** class is shown below:

Class attributes:

```
private SocketManager sockManager;
private TrackerUDPServer ms;
```

Class Constructor:

```
// Constructor
public Request(TrackerUDPServer ms, SocketManager sockMan) throws Exception {
    sockManager = sockMan;
    this.ms = ms;
}
```

Thread run() method:

```
@Override
public synchronized void run() {
    try {
        String requestLine = sockManager.read();
        // aqui viene toda la logica de negocio del server
        if (requestLine != null) {
            System.out.println("[received data] " + requestLine);
            // process data
            ClientRequestParser crq = new ClientRequestParser(sockManager, ms);
            crq.setClientRequest(requestLine);
```

```
            crq.parse();
            //responder
            crq.responder();
            // Close streams and socket.
            sockManager.closeStreams();
            sockManager.closeSocket();
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Implementation of persistent storage of swarm information

When a new node instance is created, a persistent storage is created too automatically. This storage has the same model described in the first chapter and allow us to save the swarms information. This information is deleted when the node is stopped but meantime, this info is updated in two different scenarios only:

- When a **DATABASE_CLONE** request is sent using **BINARY_MESSAGE** class. This is useful when a copy of a whole database is needed.
- When a SYNC request is sent using a **DATA_SYNC_MESSAGE** class. This is useful when tracker **MASTER** needs to notify to the cluster about data changes.

In both cases, when a message is received, data is updated automatically. However, the behaviour each message triggers is different:

- **DATABASE_CLONE** behaviour: it deletes an existing local database and when it is sure that it is deleted, it creates the same file with updated information. This new information is given as a byte array that is saved as raw data on the file.

```
public void onReceivedEvent(String currentNodeId) {
    TrackerInstance thisNode = TrackerInstance.getNode(currentNodeId);
    TrackerInstance remoteNode = TrackerInstance.getNode(sourceTrackerId);
    //read message, only, if it is for me
    if (remoteTrackerId.equals(currentNodeId)) {
        //message is for me. otherwise, drop
        if (thisNode != null) {
            //read binary content
            System.out.println(thisNode.getTrackerId()+binaryContent.length + " bytes");
            thisNode.overwriteLocalDatabase(binaryContent);
        }
    }
}

public void overwrite(byte[] data) {
    System.out.println(tracker.getTrackerId() + " overwritting LOCAL DATABASE");
    try {
        //close connection
        connection.close();
        //delete file
        boolean deleted = new File(databaseName).delete();
        if (deleted) {
            System.out.println(tracker.getTrackerId() + " Old file deleted");
            //create new one
            Files.write(Paths.get(databaseName), data);
```

```
            System.out.println(tracker.getTrackerId() + " New file created");
        } else {
            System.out.println(tracker.getTrackerId() + "Could not complete overwrite");
        }
    } catch (SQLException e) {
        System.err.println(e.getLocalizedMessage());
    } catch (IOException e) {
        System.err.println(e.getLocalizedMessage());
    }
}
```

Delete information when tracker is stopped

When a **GOOD_BYE** message is sent, it automatically triggers some actions on the node. This actions are defined in the method **onBroadcastEvent()** of the class **GoodByeMessage**. In this case, those actions are:

```
@Override
public void onBroadcastEvent(String currentNodeId) {
    TrackerInstance thisNode = TrackerInstance.getNode(currentNodeId);
    System.out.println(remoteNodeId + " Tracker goodbye message sent. Stopping");
    thisNode.stopNode();
}
```

The method **stopNode()** is in charge of stopping all background processes and listeners of that node.

SYNC behaviour: it updates a local database with specific **SQL** query. This query is inside received message and it is created by tracker **MASTER**. When tracker **SLAVE** receives this query, the only thing it has to do, is execute it. Since information is the same in both databases, it won't be any problem.

```
public void onReceivedEvent(String currentNodeId) {
    TrackerInstance remoteNode = TrackerInstance.getNode(sourceTrackerId);

    //proces message only, if it is for me
    if(currentNodeId.equals(remoteTrackerId)) {
        TrackerInstance thisNode = TrackerInstance.getNode(currentNodeId);
        if(thisNode!=null && this.query!=null){
            //update data sync request on local storage
            thisNode.syncData(query);
        }
    }
}

public void syncData(String query) {
    System.out.println(trackerId+" Synchronization received");
    persistenceHandler.sync(query);
}

public void sync(String query) {
    try {
        connection.createStatement().execute(query);
    } catch (SQLException e) {
        System.err.println(e.getLocalizedMessage());
    }
}
```

Information sync management

When a new data change is detected by the tracker **MASTER**, the **MASTER** will send a **DATA_SYNC** message to provided topic to notify rest of the tracker **SLAVES** of that change. Tracker **SLAVES** will read that message and update their local storage. Since the **MASTER** is the one who starts this process it won't be any problem of source synchronization. However, incoming messages will be queued in each tracker avoiding concurrent modifications. In this way, we solve this problem and optimize its performance.