# Project documentation

Distributed Systems

Group 01

Sergio Anguita Lorenzo

Aitor Brazaola Vicario

This document presents a brief documentation about design and implementation of distributed torrent protocol based tracker system.

# Milestone 01

Distributed Systems

Entrega 01 (Fecha límite 18 de Octubre)

Modelo de arquitectónico de despliegue

Descripción en texto de la funcionalidad que implementará cada una de las entidades que participan en el sistema distribuido

Definición del esquema de datos de la información que almacenarán los trackers

Diseño de los modelos de interacción entre las diferentes entidades: tracker-peer y tracker-tracker

Definición de los modelos de fallo de cada una de las entidades; y de la interacción entre las entidades.

Implementación en Java de la interfaz gráfica del tracker; y además, creación de la estructura “base” del código del proyecto completo.

Documento entregado:

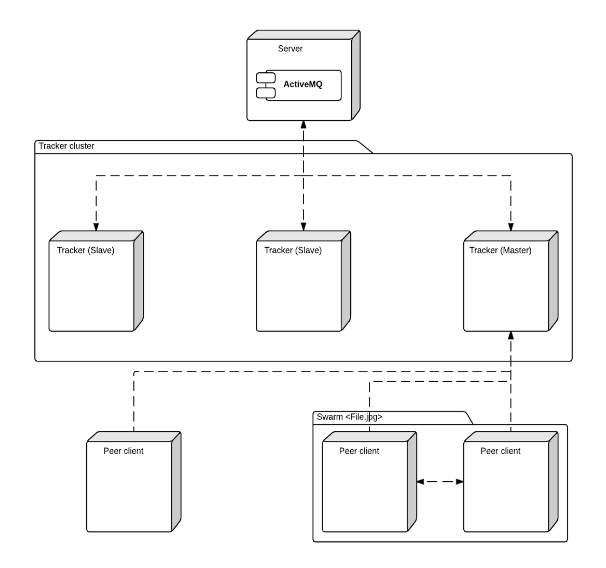
<https://docs.google.com/document/d/1iMAM6wQxRWPzL6W_UlcBE1CRVan2tj_kSEJQR_lVqg4/edit?usp=sharing>

Más información: <https://alud.deusto.es/mod/resource/view.php?id=115669>

## Deployment Diagram

This project will be focused in the tracker-tracker interaction, to synchronize the different trackers, there will be a central server *Apache ActiveMQ* with *JMS*. Although the following diagram tries to explain how the tracker cluster is arranged, it also contains a few peer clients to show the big picture of the system.

The trackers, can play two different roles, master or slave, at the same time only can exist one master responsible of attending all peer requests and synchronize the persistence operations of the other trackers. Slave trackers should pay attention if the master tracker fails to change their role to master.

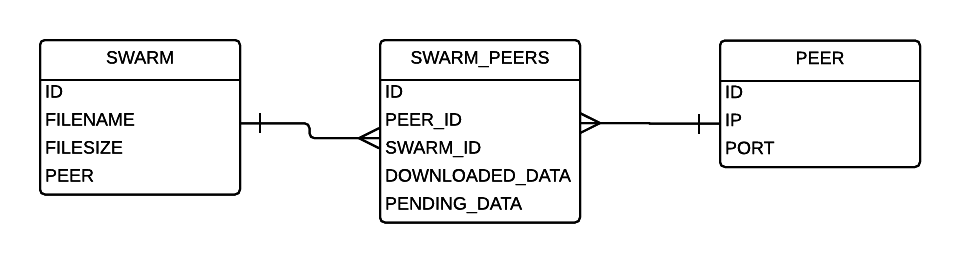


## Entity functional definition

|  |  |
| --- | --- |
| **Entity** | **Functionality** |
| Tracker (Master and slaves) | Send keep alive request to other trackers |
|  | Receive keep alive request to other trackers |
|  | Manage data redundancy of the cluster |
| Tracker (Master) | Receive ‘ready for save’ message from other trackers |
|  | Send ‘store’ command to slave trackers |
|  | Send swarm information to newcomer peer requests |
|  | Check who the slave trackers are |
| Tracker (Slave) | Send ‘store’ command from master tracker to save data |
|  | Receive ‘store’ command from master tracker to save data |
|  | Receive cluster structure information from the master when is recently added |
|  | Check who the master is |
| Peer | Request new tracker connection |
|  | Receive tracker connection response tracker message |
|  | Send file request information to the tracker |
|  | Receive peer information about the swarm of a file requested from the tracker |

## Entity relationship model

In the following diagram will be explained the data architecture to store the relevant information for each tracker, the data persistence technology used will be SQLite because of the ease of use and versatility of only having to manage one .sqlite file for each tracker entity.



|  |  |  |
| --- | --- | --- |
| **Entity** | **Attribute** | **Type** |
| SWARM | ID | INT (autoincremental) |
|  | FILENAME | VARCHAR (255) |
|  | FILESIZE | FLOAT |
|  | SEEDERS | INT |
|  | LEECHERS | INT |
| SWARM\_PEERS | ID | INT (autoincremental) |
|  | PEER\_ID | INT |
|  | SWARM\_ID | INT |
|  | DOWNLOADED\_DATA | FLOAT |
| PEER | ID | INT (autoincremental) |
|  | IP | VARCHAR (15) |
|  | PORT | INT |
| TRACKER | ID | INT (autoincremental) |
|  | IP | VARCHAR (15) |
|  | PORT | INT |
|  | ROLE | INT |
|  | LAST\_KEEPALIVE | LONG |

## Interaction model among entities

Llega un nuevo tracker (cluster vacío)

Espero KEEP ALIVE

No recibo

SOY MASTER

Creo la BD por primera vez

—

El tracker se va del cluster

Manda mensaje Adios

Se va

—

Tracker entra

Espera KEEP ALIVE

Recibe KEEP ALIVE

Si se conoce el master:

Se le pide la BD y se clona su estado y se establece como slave

Si no se conoce el master: Pregunta quien es el master

Si recibes que no hay master: Se entra en negociación. Se establece uno.

Si recibes que si hay master, le pides la BD

—

Se va el master

Manda adios

Los demás lo eliminan de la lista KEEP ALIVE

Como se sabe quien es el master

Los demás inician negociación

—

Se va un slave (VOLUNTARIAMENTE)

Manda adios

Los demás lo eliminan de la lista de KEEP ALIVE

—

Se cae un slave

Los demás no reciben su KEEP ALIVE

Los demás lo eliminan de la lista de KEEP ALIVE

—

Se cae un master

Los demás no reciben su KEEP ALIVE

Los demás lo eliminan de la lista de KEEP ALIVE

Los demás inician proceso de negociación

—

NEGOCIACIÓN

Si la lista está vacía eres master

—

Si la lista tiene elementos

Se elige el id numérico mas grande

—

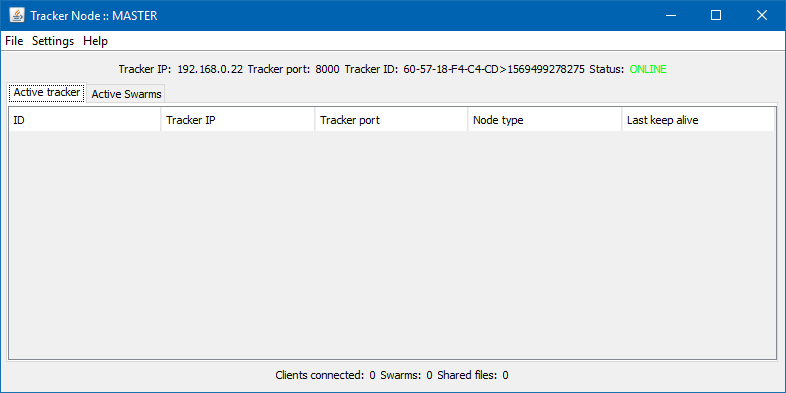
## JMS Messages

|  |  |
| --- | --- |
| Topic: alive | * Keep Alives (id, port, ip) * Frequency: 1s |
| Topic: handshake | * HELLO (slave: id, port, ip) * BYE (slave: id, port, ip) * Send message ‘I am the master’ (master: ip, port, id) * Frequency: on demand |
| Queue: sync  (For data isolation between tracker nodes, it is good to implement filters on activemq message receiver) | * Send in.memory objects * Request database * *Notify data changes ( data: object that changes its data). Not really needed since its considered part of the multicast* * Frequency: on demand |

## Failure models

|  |  |  |  |
| --- | --- | --- | --- |
| **#NUM** | **Related entity** | **Failure** | **Solution** |
|  | Tracker (Master) | Not every slave tracker has sent ready for write message | Wait 5 seconds and check again the list of confirmations received.If there is no confirmation after 5 seconds the tracker is deleted of the list of active trackers. |
|  | Tracker (Master) | A peer has no sent information about the status of the download after 1 minute | Probably the peer has been disconnected from the network and should be removed from the list of available peers of a swarm. |
|  | Tracker (Slave) | Not received keep alive message from the master | Start the new master selection process. |
|  | Tracker (Slave) | Database not received | Repeat database clone request to master |
|  | Tracker (Both) | No consensus on new Master | Repeat master tracker selectio process until a master is selected |

## Tracker Graphical User Interface



Tracker master interface for management

## Improvements

Current approach of the tracker system can have some improvements that can enlighten a complete distributed system architecture. Some of those improvements are:

* Real time download folder control for space management.
* Master selection algorithm improvement based on uptime and bandwidth speed parameters.

# Milestone 02

Distributed Systems

Este segundo entregable tiene por objeto la implementación de la funcionalidad relacionada con la gestión de redundancia de trackers y la sincronización del repositorio de información utilizando comunicaciones JMS.

Gestión de la incorporación al grupo y asignación de identificadores.

Proceso de elección del Master.

Gestión de tiempos de vida y detección de trackers que dejan el grupo.

Gestión de la sincronización del repositorio de información.

Más información: <https://alud.deusto.es/mod/resource/view.php?id=117261>

## Tracker design

Current section explain how different requirements are designed to accomplish them.

* Package hierarchy:

**CORE package**

es.deusto.ssdd.bittorrent.core;

It contains classes that belongs to system core. Current classes are:

* TrackerUtil.java: Contains different method utilities for proper execution.

**PERSISTENCE package**

es.deusto.ssdd.bittorrent.persistent;

It contains classes that belongs to data persistence. Current classes are:

PersistenceHandler.java: Contains a driver handler class for accessing to SQLite database instance.

**GUI package**

es.deusto.ssdd.bittorrent.gui;

This package contains all code related to tracker Graphical User Interface. This code is organized following MVC convention, so that the code is separated in three different packages: model, view and controller.

TrackerWindow.java: Represents a tracker window based on Java’s JFrame.

TrackerGUIEvents.java: A ENUM file representing all available events on the tracker interface, in one single place.

InterfaceRefresher.java: An interface that represents and output of the system. Current implementation is used for refreshing GUI.

**JMS package**

es.deusto.ssdd.bittorrent.jms;

This package contains all code related to JMS connection, topic definition and different message types used by the system. It is organized in three different packages:

Listener: JMS listener and sender daemons are placed here.

JMSMessageListener.java: Generic JMS message listener.

JMSMessageSender.java: Generic JMS message sender.

KeepAliveDaemon.java: Keep alive message sender daemon.

Message: JMS message generation classes and definition classes.

IJMSMessage.java: An interface that represents any kind of system compatible JMS message.

MessageCollection.java: a Java Factory class for JMS Message creation.

/wrapper: This package contains wrappers for messages that are sent over the network.

DataSyncMessage.java: a message holder for database information synchronization

GoodByeMessage.java: a message holder for Group membership management

HelloMessage.java: a message holder for Group membership management

KeepAliveMessage.java: a message holder for keep alive messages

Model: JMS specification parameters are defined here.

TrackerDaemonSpec.java: a Java ENUM that defines different tracker daemons: HANDSHAKE\_SERVICE, KEEP\_ALIVE\_SERVICE and DATA\_SYNC\_SERVICE.

TrackerInstanceNodeType.java: a Java ENUM that defines tracker instance node types: master or slave.

TrackerStatus.java: a Java ENUM that defines tracker status options: offline or online.

## Tracker implementation

Current section explain how different requirements are implemented to accomplish them.

* Group membership management:

Group membership management is done using two different messages: HELLO\_WORLD\_MESSAGE and GOODBYE\_MESSAGE. Each one is used for indicating to the cluster that a node is added or removed to the tracker.

* Tracker ID assignment:

Tracker node assignment is done using a local algorithm that assigns unique IDs to each of the nodes. Algorithm is based on Device MAC address and system nanotime. For this reason, there is no need of sending messages to the cluster and the ID is generated in a unique way.

* Master election process:

When a new node is added to the cluster, it is proclaimed as Master node by itself with no interaction at all with other nodes. After this, the new node sends a HELLO\_WORLD\_MESSAGE to JMS Server, so that rest of the cluster knows that a new node is added to the cluster. After this, rest of the nodes that are listening for incoming HANDSHAKE messages on tracker.hanshake topic, receive the node information and update their local info.

If for any reason, any node detects that given remote node ID and local node ID, there is a need of updating master node, the local node, will update its information.

When a master leaves the cluster, it will send a GOODBYE\_MESSAGE, and in the same way done with new nodes, evaluation of the node IDs will be done.

* TTL (time to life) management:

Each node of the cluster, independently of the type (MASTER or SLAVE), will have a background daemon running sending KEEP\_ALIVE messages every 2 seconds. In the same way, KEEP\_ALIVE messages receiver will be listening for incoming KEEP\_ALIVE messages with the different that, it will be counting time between last message and current message. If the time ends, with is set as five seconds by default, and no message has been received, daemon owner will understand that remote node is dead, and it will update its local information.

* Detection of trackers leaving the group:

Tracker nodes that leave the group is done sending a GOODBYE\_MESSAGE to JMS Server, so that any node of the cluster knows the state of the rest.

* Information sync between trackers:

When a new tracker is connected to the tracker cluster, tracker MASTER sends a copy of its own database via JMS Messages using BinaryMessage class, which is just a wrapper for binary streams.

## Detected issues

The current sections show software development and design related issues during this second delivery.

Detected issues are summarized below:

* **Tracker Master assignment process:** we already defined a method for tracker master definition among tracker cluster nodes. This method, works and is stable, however, we have found and issue that could lead to a bad tracker master assignment. The issue itself, is related to how messages are delivered to each node, since we are unable to detect how many of cluster nodes are before the send HELLO\_WORLD message. This blind situation, generates that, if for some reason one of the nodes, does not send a HELLO\_WORLD message, other cluster nodes won’t be able to detect that there is one more node on the cluster to take into account. With this problem in mind, we thought that this kind of integrity issues are out of the scope of the project, but we are aware of them.

# Milestone 03

Distributed Systems

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.

# Milestone 04

Distributed Systems

Implementar un peer Bittorrent (Fecha límite: 13 de Enero de 2017)

El objetivo de este proyecto es el diseño y desarrollo de un cliente de BitTorrent compatible con el servidor que has desarrollado como parte del Proyecto 1. El cliente debe contener la siguiente funcionalidad:

Abrir e interpretar ficheros .torrent

Conectarse e interactuar con tu clúster de trackers usando conexiones UDP.

Conectarse a diferentes peers mediante conexiones TCP para descargar y servir contenidos.

El funcionamiento básico del cliente debe contemplar los siguientes pasos:

Abrir e interpretar ficheros .torrent (de uno en uno por simplificar).

Reservar en el disco el espacio necesario para almacenar el contenido a descargar.

Establecer una conexión UDP con el Tracker que has implementado en el Proyecto 1.

Procesar la respuesta del Tracker y extraer la información de los Peers.

Abrir conexiones TCP con los Peers e interactuar con ellos (descarga y subida se valorará especialmente la interacción concurrente con más de un Peer).

Verificar el hash SHA-1 de cada pieza descargada y almacenarla en disco.

Notificar a los Peers con cada nueva pieza verificada correctamente.

Recibir y atender solicitudes de descarga de piezas por parte de otros Peers.

Repetir los pasos 5 - 8 hasta completar cada contenido a descargar.

Contactar periódicamente con el Tracker para actualizar el estado y solicitar la nueva lista de Peers.

Al finalizar la descarga, contactar de nuevo con el Tracker para indicar que se ha descargado completamente el contenido.

Además de estos pasos, en cualquier momento puede pausarse la descarga. En ese instante el cliente debe almacenar la información del estado de la descarga para que cuando se arranque de nuevo se continúe con el proceso de descarga. Sólo se tendrán en cuenta las piezas completas que se hayan descargado y verificado correctamente.

Se pide:

Define la arquitectura de hilos y sockets para gestionar la comunicación con el Tracker y cada uno de los Peers (diagramas de flujo y texto). Especifica claramente el número de hilos, sockets (relación entre ambos), así como si usarás alguna información de “estado” vinculada a dichos hilos y/o sockets. Indica también como gestionarás la información parcial descargada y la pausa/reanudación de una descarga.

Implementa el cliente de BitTorrent que interactúe con el Tracker que has implementado como parte del Proyecto 1. Puedes utilizar como ayuda el esqueleto de código del proyecto “SSDD\_BitTorrentClient”.

Valida tu cliente con tu clúster de Trackers y con dos instancias de tu propio Peer.

Más información: <https://alud.deusto.es/mod/resource/view.php?id=125476>

Documentacion entrega 4