ELEC-H417

Communications networks: protocols and architectures

Project report

Creation of a BitTorrent-like client and tracker

Academic year: 2016-2017

Section: M-IRELE

Pierre Baudoux Cédric Hannotier

Mathieu Petitjean

Brussels Faculty of Engineering

ULB - VUB

Contents

1 Introduction

This report presents the implementation of a BitTorrent-like service as a part of the ELEC-H417 course at the Brussels Faculty of Engineering: Communication networks, protocols and architectures.

In this project, two peers (Alice and Bob), a client (Charlie) and a tracker will be implemented. The scheme is represented in Figure 1.

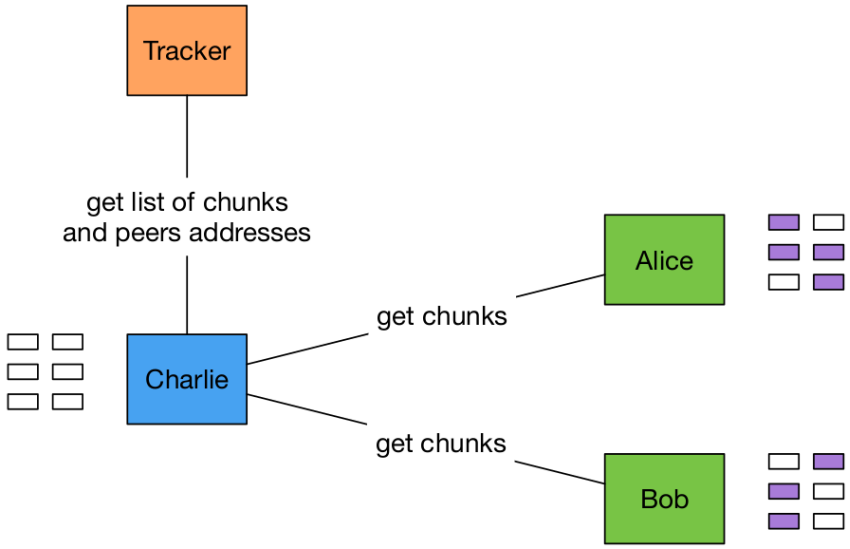

Figure 1: Scheme of the architecture

A file will be split in chunks, which will be distributed between the two peers. The client should be able to connect to the tracker in order to be informed of which peers own which chunks, and then download the chunks and recover the original file.

The project is divided in three steps, the third one being optional. This report will, for each step, describe the objective of the step, present the description of the proposed solution and its particularities. Moreover, a sequence diagram will be provided.

2 Step 1

2.1 Description of the objective

In the first step, only three entities are created: Alice, Bob and Charlie. The two peers need to read configuration files to know their own IP addresses. The tracker is not implemented yet, the client has to read a configuration file in order to know where the chunks are located.

Knowing the message format used in the project[1], the peers will analyse the requests they receive from the client and they should be able to either send the chunk that was asked or to return an error message.

2.2 Proposed solution

Using object-oriented Python programming, classes `Peers` and `Client` have been created. Alice and Bob are thus `Peers` objects and Charlie is a subclass of `Client`. Charlie will be `Clientv1`, `Clientv2` or `Clientv3` depending of the step. A superclass `Super` (superclass of `Peers`, `Client` and `Tracker`) has been implemented to avoid code repetitions.

All the data transfer between the peers and the client are handled using TCP to ensure that every requested chunk is correctly transmitted (provided a peer owns it). Moreover, the code was designed to limit the opening and closing of sockets.

2.2.1 Peers

The way peers are implemented is quite straightforward. When instantiated, the peer opens a socket and stays idle while waiting for a connection. When a connection is received from the client, a single thread by peer (and by connection) is launched.

This thread will analyse the incoming messages and will send back the appropriate errors when necessary. If the message format and content are correct, the peer will send the requested chunk to the client.

The thread will run and send errors through the socket while there are incoming messages from the client. If the client stops communicating, the peer will assume that all the requests have been sent and it will close the connection.

2.2.2 Client

The client is implemented in order to achieve the maximum file transfer speed, while handling cases where the configuration file declaring which peer owns which chunk is not correct. Furthermore, the client works regardless of the number of peers.

Two threads are created so that the chunks are requested and downloaded from Alice and Bob in a parallel way. The library `Queue` is used, and four queues are created when the client is

[1] `https://github.com/jpagex/elec-h-417-project/blob/master/statement.pdf`

instantiated. The first and second queues contain the list of chunks that are only owned by one of the two peers (qA for Alice only and qB for Bob only). In this way, each thread is sending requests to a different peer and it is only asking for chunks that are not shared. When the queue associated to a thread is empty, it will request chunks that are owned by the two peers, which are listed in the third queue (qAB). The fourth (qTot) contains all the chunks needed. It allows to know if all chunks have been downloaded. This implementation allows to ensure that if one of the connections is slower that the other, the impact on the file transfer speed will be as little as possible.

Moreover, when a chunk which was placed in the queue qAB is not found where it was requested, it will be put in qA or qB, depending on which peer sent a `NOT FOUND ERROR`. This implies that the client will handle cases where the chunk is supposed to be owned by Alice and Bob but is actually owned only by one of the two.

2.3 Sequence diagram

A simplified version of the sequence diagram of the step 1 code is shown in Figure 2. It can be seen that the client uses two threads, each dedicated to the communication with one peer. The interactions with the second peer (Bob) are not shown for sake of clarity, since they are very similar to those concerning the first peer (Alice).

Three specific cases are described :

- Single chunk, no error: the thread gets the chunk to ask in the queue qA, which is filled with chunks only owned by Alice. It sends a request to the corresponding peer. If no error occurs, it writes the chunk on the disk and signals that the chunk was correctly received.
- *Multi Chunk*, no error: the thread has finished to ask for chunks listed in qA, and is now getting them for the queue qAB, containing chunks that are supposed to be owned by both peers. The follow-up is the same as in the previous case.
- *Multi Chunk*, `NOT FOUND`: if the thread is asking for a chunk listed in qAB but Alice responds that she does not actually own it, it is concluded that the other peer (Bob) owns it. It will thus add the given chunk in qB, so that the thread 2 will ask Bob for it.

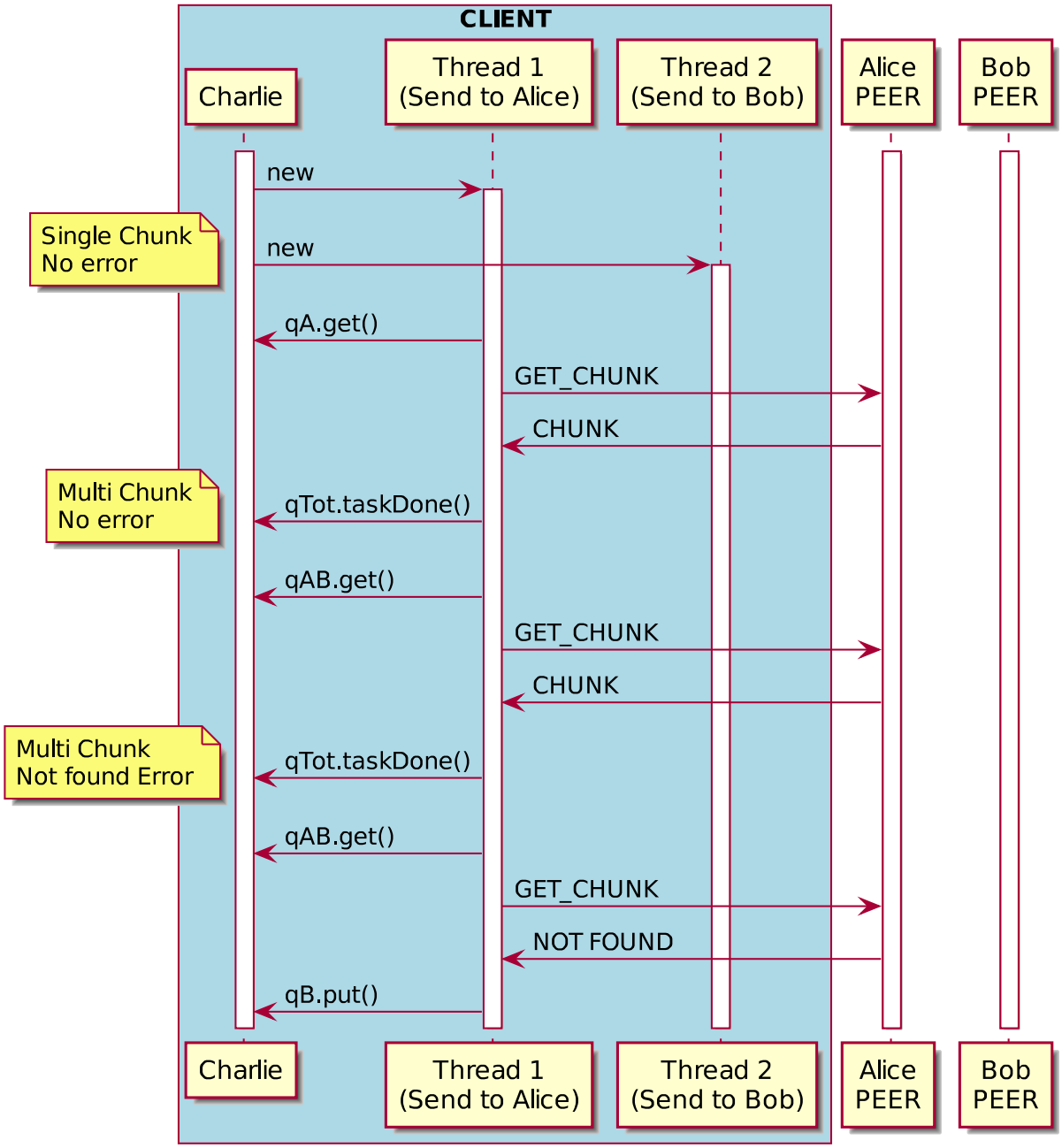

Figure 2: Sequence diagram of step 1

3 Step 2

3.1 Description of the objective

This step consists of the addition of the tracker. It will provide the client the list of chunks and the peers that own each chunk. The client will thus not have to fetch this information in a configuration file but still needs to know the IP address and port number of the tracker.

3.2 Proposed solution

The `Tracker` class now implements the reading of the configuration file as the client did in step 1, in order to know the addresses, ports and chunks of the peers.

When a request is sent by the client to the tracker, all the file information is sent in a message. According to the content of this message, the list of chunks are placed in the adequate queues (only Alice, only Bob or owned by both). The file `file1.ini` (to avoid overriding `file.ini` for testing simplicity) is created and the follow-up is exactly the same as it was in step 1. The request to the tracker is sent using TCP to ensure that the correct IP and port will be received, as these are essential to establish a connection with the peers.

3.3 Sequence diagram

The sequence diagram of the second step is shown in Figure 3.

The principle is very similar to the first step once the information of the file is received by the tracker. The connection with the tracker is established similarly to the connection with the peers, and the queues still offer the maximum transfer speed possible.

3.4 Bonus - contacting the lab tracker

2 bonus points could be earned by contacting a tracker at the lab and downloading provided in chunks. As the IP address and the port number of this tracker were known, the code of the step 2 was able to contact the tracker, download the chunks and reconstruct the file (5 minutes of the *Start Wars III* movie). It is to be noted that some chunks that were supposed to be owned by both peers were not, but the implementation with the queues allowed to handle it without any problem.

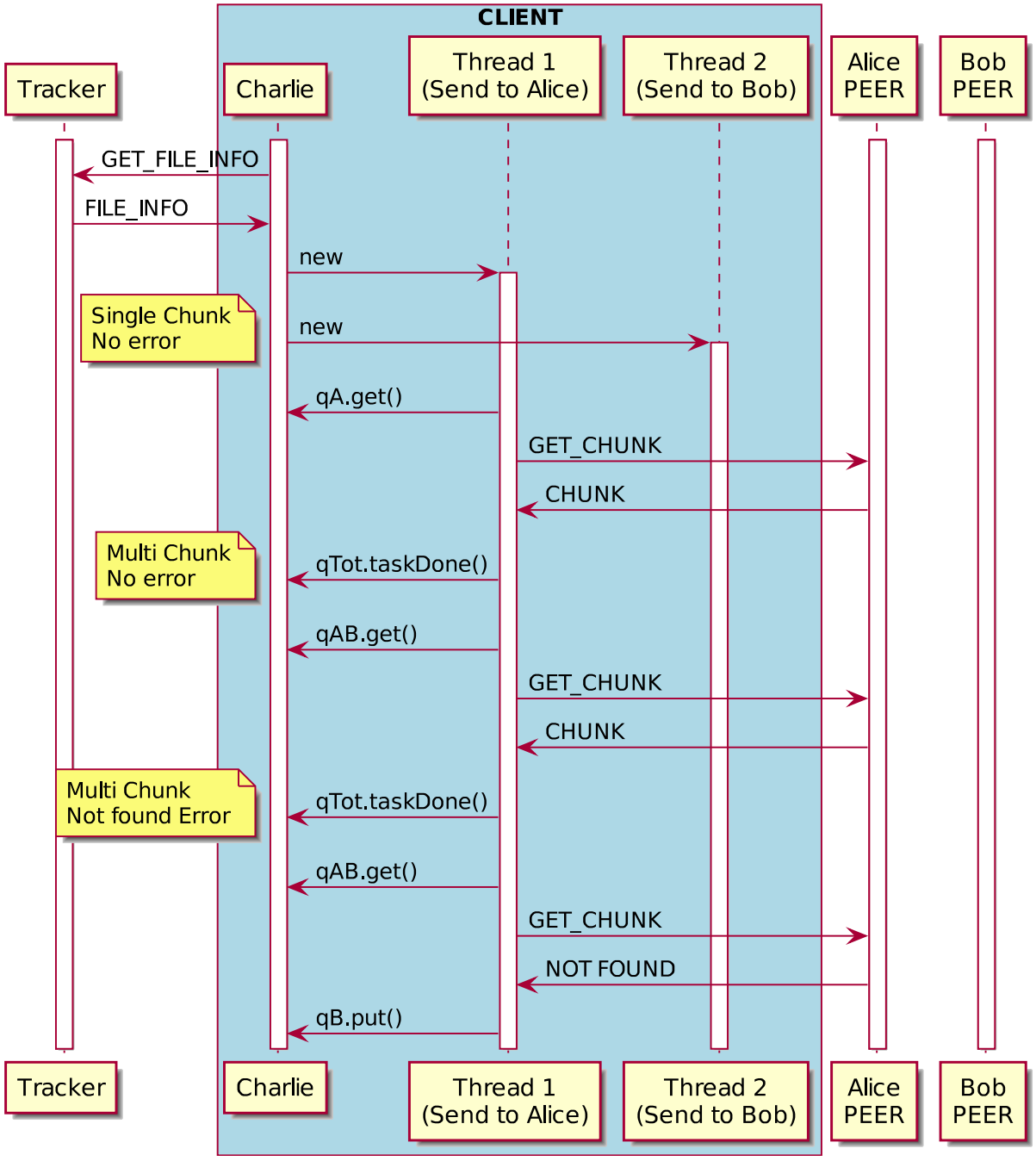

Figure 3: Sequence diagram of step 2

4 Step 3 (optional)

4.1 Description of the objective

The last step consists of modifying the client and the tracker in such a way that the client will not need a configuration file anymore. The client will instead try to discover the tracker by sending a broadcast message, and the tracker should respond to these discover requests.

4.2 Proposed solution

When launched, the tracker is listening to broadcast messages and to TCP requests in separated threads. This broadcast message is sent by the client when wanting to start a file download. The message is sent using UDP because this protocol does not require an established connection between the two agents.

4.3 Sequence diagram

The sequence diagram of the third step is shown in Figure 4.

The tracker now implements two threads, the first one listening to broadcast message and the second one handling TCP connection with the client as it did in step two.

5 Conclusion

The BitTorrent-like service is working without any trouble. It handles the cases where the chunks are supposed to be owned by two peers but are not, and should be working if the connection has to be established with more than two peers.

Thanks to the use of `Threads` and `Queues`, the file transfer is as fast as possible with a TCP connection. Moreover, the tracker can handle UDP broadcast requests while sending chunk informations via TCP.

The two bonus tasks were achieved, as the file on the lab server was correctly retrieved and the step 3 was implemented.

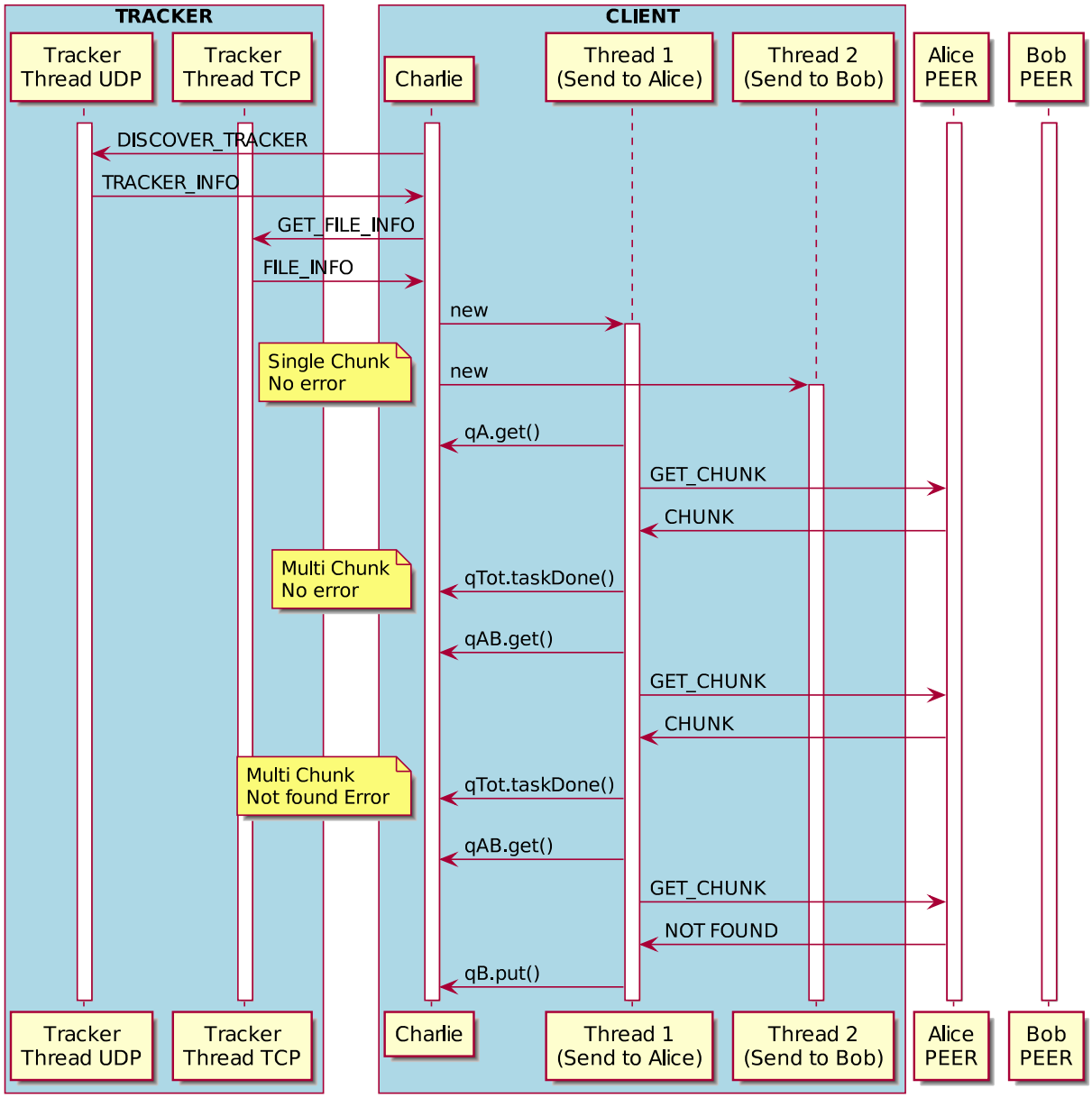

Figure 4: Sequence diagram of step 3