Team 8
Fernando Blanco Albendea, Alvaro Morante

Final Project: Bittorrent

This report will cover the principal aspects of the Bittorrent project and all the issues, difficult concepts and coding problems we have encountered throughout the last month.

The project consists of a Bittorrent client that receives the name of a torrent file from the user input, reads the information in the file, connects to the tracker, receives the IP addresses and ports from the different peers from which the pieces of data will be downloaded and connects to these peers. The peers will send pieces of data and our client will store them and write them in order into a file.

First of all, our client asks the user for the name of the file. After that, it opens the file and stores all the data. The file contains multiple dictionaries with information about the tracker, the number of pieces, the length of each piece, and the hash of each piece in order. In order to read all this information, we had to decode the data using the function `be_decode`. This function was taken from the Internet and it reads the Bittorrent bencode dictionary and stores it into a node. The library can be found in the bencode.c and bencode.h files. This library was retrieved from https://github.com/cwyang/bencode. From the nodes, we obtained the tracker URL, the length of the file to be downloaded in bytes, the length of each piece and the number of pieces, everything with the function `be_dict_lookup`, that receives the node and the key of the information that we are willing to retrieve and returns the information in another node.

Once we had retrieved the tracker URL and its port, we could send the GET/ announce message. This message contains a hash of the `info` content URL encoded. Our `peer_id` in this connection is just the string "abcd". The hashing and URL encoding are done in the functions `hash` and `url_enc` respectively.

At the beginning, we were sure that the compact mode would be much easier, since we would just have to read the IP (4 bytes) and the port (2 bytes) in each 6 bytes on the received buffer. However, during the last week of the project we found out that most trackers (almost all of them) were sending the responses with the information about the peers in extended mode, even though the compact was set to 1 in our GET/ announce message. For this reason, our client is assuming that no tracker will respond with a compact format. After connecting to the tracker and receiving the dictionaries with the peers, we decode the data with `be_decode` and read the IP addresses and the ports for each peer using `be_dict_lookup` with the keys "`ip`" and "`port`".

It was time to send the `handshake` message to all the peers, but in order to do that, we had to create as many threads as number of peers through a function (called `function` in our code) that would connect to each peer and receive the pieces. We will now see what each of the threads does to connect to the peer, read the bitfield, request and download pieces.

In `function`, each thread has an associated peer. At the beginning, the thread connects to the peer and sends the `handshake`. We made a structure that contains the port, IP address, index and peer ID (empty for now) and passed it through the `pthread_create` to make things easier. After that,

our thread expects to receive a handshake that contains the ID of the peer. Then, it repeats a procedure until all the file is downloaded (i.e., until the thread dies), which is the following: Receive 4 bytes, which will be the length, and compute the length. If the length = 0, the message received is a keep-alive. Do nothing. Else, receive one more byte for the message type.

If the type is 5, it is the bitfield. We will read it and store the information in a global array of numbers called `bitfield`. If there is a 1 in `bitfield[i]`, it means that we will be able to receive that piece because at least one peer has it. We will do the same with another array (local) that will store the bitfield only for our peer. This is done to avoid requesting a piece to a peer without making sure that the peer has the piece. After that, we will send an interested to tell the peer that we want to receive pieces from him, and that we want him to unchoke.

If the type is 4, we have received a have message. We will calculate the index from the received message and update the global bitfield and our peer's bitfield.

If the type is 1, our peer has unchoked us. We will store this information in a global array of flags called `choke_array` that we will use to know which peers are choked and which peers are unchoked. This is important because at the beginning, we sent useless requests to peers that were choked, i.e. that were not listening. We will also create a new thread that will execute the function `request`. We will see how the requests work later on.

If the type is 0, our peer has choked us. Similarly to the unchoke message, we will update the `choke_array` to tell other threads that the peer is not listening anymore.

Finally, if the type is 7, we are about to receive a piece. I will say "subpiece" from now on, because we are receiving each piece in smaller subpieces. The length of the subpiece was obtained from the torrent file, so we simply had to receive the subpiece and store it in a global 3D array called `piecefield`. It had to be a 3D array because it needs to store the pieces, but each piece is divided into subpieces, and each subpiece stores a certain number of characters. After that, if an entire piece was received, we hash it and compare it to the hash stored in our torrent file that corresponds to our piece. We have to make sure that they are equal because we need to know which pieces have been downloaded. We will store that information in the global array `bitfield`. Once the piece `i` has been downloaded, we will set `bitfield[i]` to 3. This will avoid requesting and downloading pieces that have already been downloaded. We will do this differently if our piece is the last subpiece of the last piece, because it has a different length. After storing our subpiece and checking if a piece has been downloading, we will check if all the pieces have been downloaded. In such case, we will kill the thread.

We now know what to do depending on the message we receive from the peer. However, we will never receive a piece before requesting it to the corresponding peer. After receiving the unchoke message, a new thread is created. That thread will run at the `request` function. At the beginning, it will request all the pieces that the peer has. That is why we stored that information locally, to know which pieces each peer has. However, it also checks that the global array `bitfield` is set to 1 at each position, since we do not want to request a piece that has already been downloaded or requested. Since each piece has several subpieces, one request message will be sent per subpiece, with a different offset. The last piece will be sent differently, since it has a different number of

subpieces and a different length. After requesting the piece `i`, `bitfield[i]` will be set to 2 to avoid requesting the same piece twice. We found out that requesting the same piece twice means downloading the same piece twice, and we do not like that. After requesting all the pieces, since we were having problems when downloading all the pieces, we decided that each thread would request again all the pieces that have not been downloaded yet. This is to make sure that the file would be perfectly downloaded.

One of the biggest problems we have had during this project were to find torrent files that we could download. Some of them were too large (around 600MB), and in other shorter files, the peers would reset the connection, which led our program to finish unexpectedly. Another problem was that when a thread failed to receive, it died, and this gave us a lot of performance problems. In a 200 pieces file with 5 peers, if 2 threads died, it would take twice as much to download the file, and maybe some pieces could not been downloaded because the 3 peers remaining did not have them. In order to fix that, every time a `receive` fails, in `function`, a new thread is created before killing the current thread. We did the same when a `send` fails in `request` to make sure that all the pieces were requested. Other problem was that after downloading all the files, the program took around one minute to finish and write the pieces into the file, since we could not write into the file until all the threads died. What we did to fix this problem was to create the flag `downloaded` that would be set to 1 as soon as all the pieces were downloaded. This flag is checked very often in our functions to make sure that when all the pieces are downloaded, all the threads die. A last minute change was also the `pthread_cancel` function when we set the `downloaded` flag to 1. We did not know that it existed until two days before the deadline. Since we were storing all the threads into a global array, we could kill all the threads as soon as all the pieces were downloaded, and our problem was fixed. Another last minute problem we had was that 1 in every 5 tries, we had a segmentation fault right after connecting with the tracker. Our problem was that our `peer_id` was a 20-bytes random string, and sometimes that id was invalid. Setting the `peer_id` to “abcd” got rid of the segmentation fault, because “abcd” is always a valid `peer_id`.

We both worked together over the whole project, just minor updates were done independently when we couldn’t meet to work over it. Therefore, every feature was developed by the two of us. Working together made the development of the project easier, as we could be checking by the same time that we were programming if what we were doing was right. That also helped us to know how every part of the code worked, as we had both collaborated over it.