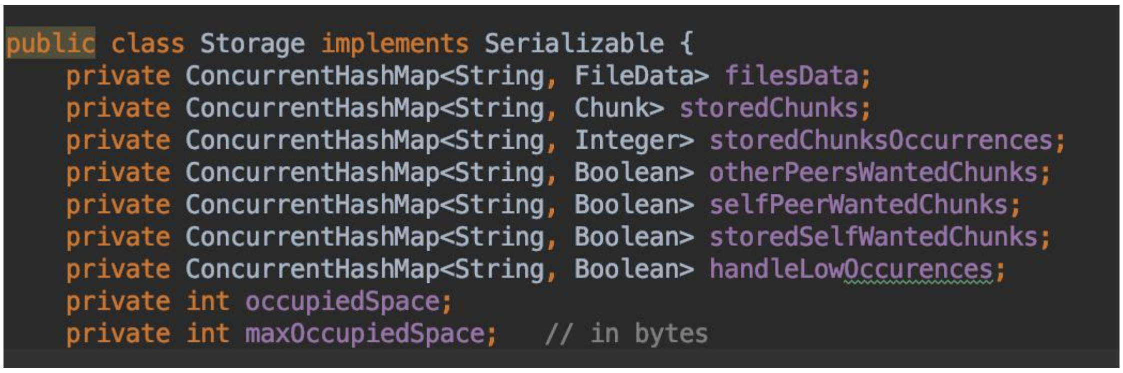
SDIS Project 1:
Distributed Backup Service

Paulo Roberto Dias Mourato, up201705616
Tiago Candeias Verdade, up201704003

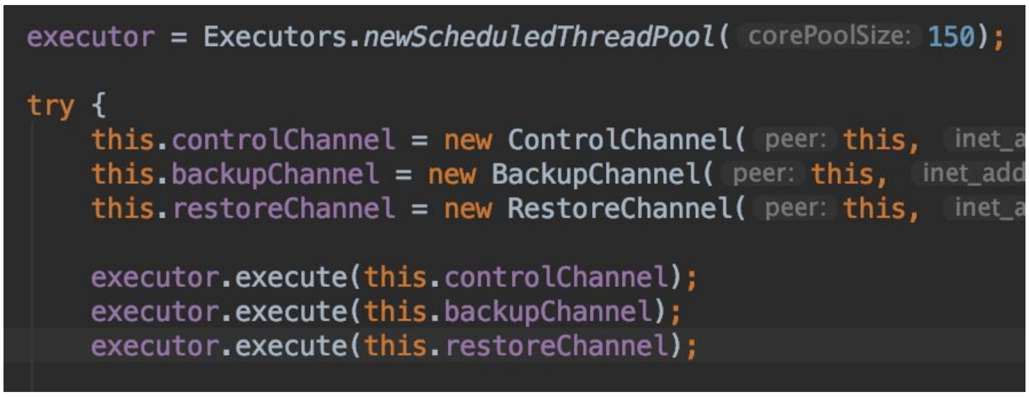
Concurrent execution of protocols

Since a lot of times it would be beneficial to use multiple protocols running at the same time we took that into account and tried to make it possible without having any inconsistencies. To achieve this, each peer has a Storage object that represents its files and chunks state. This Storage object contains various attributes, the majority being *ConcurrentHashMap* to ensure thread safety, which can be modified in multiple threads thus serving as a mean to communicate with each other.



```
public class Storage implements Serializable {
    private ConcurrentHashMap<String, FileData> filesData;
    private ConcurrentHashMap<String, Chunk> storedChunks;
    private ConcurrentHashMap<String, Integer> storedChunksOccurrences;
    private ConcurrentHashMap<String, Boolean> otherPeersWantedChunks;
    private ConcurrentHashMap<String, Boolean> selfPeerWantedChunks;
    private ConcurrentHashMap<String, Boolean> storedSelfWantedChunks;
    private ConcurrentHashMap<String, Boolean> handleLowOccurences;
    private int occupiedSpace;
    private int maxOccupiedSpace;   // in bytes
```

Each peer also executes a thread for each *multicast channel*, ensuring that it receives all messages present in all channels

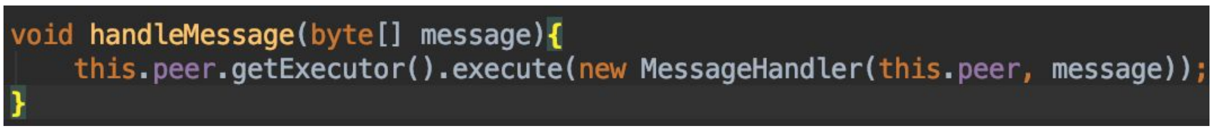


```
executor = Executors.newScheduledThreadPool( corePoolSize: 150);

try {
    this.controlChannel = new ControlChannel( peer: this,  inet_a
    this.backupChannel = new BackupChannel( peer: this,  inet_add
    this.restoreChannel = new RestoreChannel( peer: this,  inet_a

    executor.execute(this.controlChannel);
    executor.execute(this.backupChannel);
    executor.execute(this.restoreChannel);
```

Since if we had a thread for each channel and we were also processing each message in it, it could lead to some messages being lost, we decided to create a new thread for received messages. Thus ensuring that the peer is always listening to all channels at all times.



```
void handleMessage(byte[] message){
    this.peer.getExecutor().execute(new MessageHandler(this.peer, message));
}
```

Restore Enhancement

In order to enhance the Chunk Restore Protocol, we managed to pull a method that can decide the peer that should send the desired chunk, and also we used TCP communication. Each peer has a HashMap ‘otherPeersWantedChunks’, so like the name says it stores as keys the “FileID_ChunkNo” of the chunks wanted by other peers, and as values a boolean representing it’s desirability, if True the chunk is desired, if False the chunk is not desirable, so that a peer knows if it should send a Chunk or not. The HashMap values for each chunk are set to True each time a peer gets a GetChunk message, and set to False everytime a peer sends a CHUNK message, or the peer receives a CHUNK message by other peers. We established a Client/Server type of communication, being the Initiator Peer the Server, and the Other Peers the Clients. The Initiator Peer only sets the port to open the ServerSocket and waits for the other peers to make the connection, using serverSocket.accept(). We also set this port used for the communication as reusable using socket.setReuseAddress(true). The other Peers only send the content of the desired chunks, so that the Initiator Peer can rebuild the initial file.