U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Distributed Systems

First Project

2019/20

Class 1 - Group 7

Maria Helena Ferreira -- up201704508@fe.up.pt
Sofia de Araújo Lajes -- up201704066@fe.up.pt

Introduction

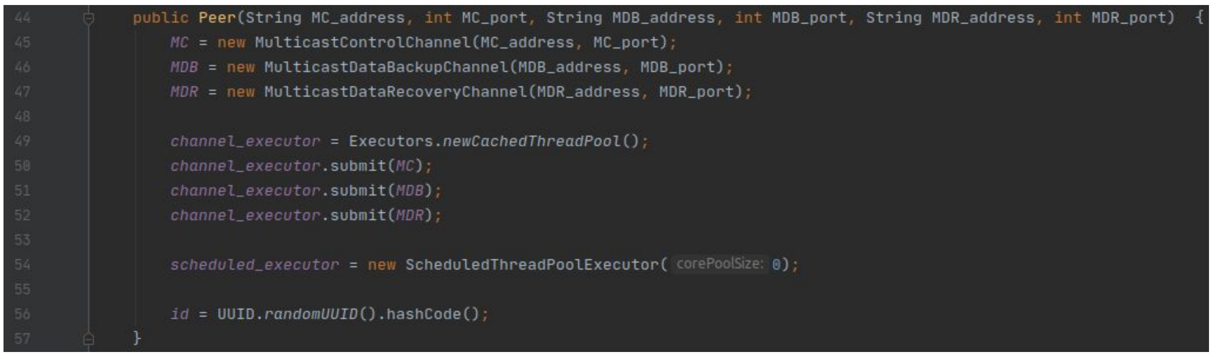
This report aims to describe how we designed and implemented concurrency in the protocols' development. The project was design to deal with high concurrency and scalability.

Concurrency Implementation

To increase the performance we had to add concurrency. The concurrency was obtained by using two sets of multi-Threaded executors.

The first is meant to deal with the high scalability of the necessity in the three channels. As the program is expected to handle a chaotic number of messages to be received, read and interpreted in each channel, it is necessary to process all of them as quick and simple as possible. To solve this we decided to process each message received in the multicast socket using the CachedThreadPool. This variant of the thread pool is useful when tasks submitted should not wait (need to be addressed immediately) - a cached thread pool can have up to 2^{31} - meaning this is suitable for short lived asynchronous tasks.

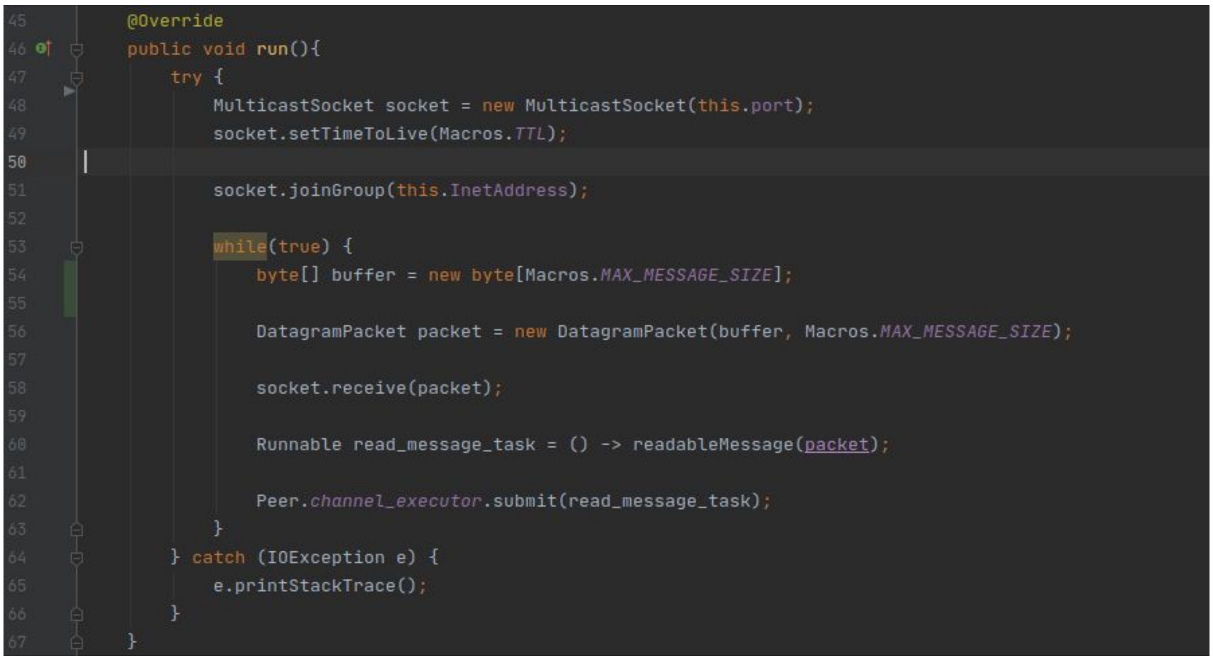
In the following picture it is possible to observe the creation of a CachedThreadPool, as well as the submission of the each channels (that implement the Runnable interface), in the Peer constructor:



```java
public Peer(String MC_address, int MC_port, String MDB_address, int MDB_port, String MDR_address, int MDR_port)  {
    MC = new MulticastControlChannel(MC_address, MC_port);
    MDB = new MulticastDataBackupChannel(MDB_address, MDB_port);
    MDR = new MulticastDataRecoveryChannel(MDR_address, MDR_port);

    channel_executor = Executors.newCachedThreadPool();
    channel_executor.submit(MC);
    channel_executor.submit(MDB);
    channel_executor.submit(MDR);

    scheduled_executor = new ScheduledThreadPoolExecutor( corePoolSize: 0);

    id = UUID.randomUUID().hashCode();
}
```

Picture 1: Peer class constructor

In the run method (an Override of the Runnable interface method) of the Channel (abstract class that parents the Data Backup and Data Recovery multicast channels), we can observe in line 62 the submission of a task (*readable_message*) that will process the received message (*DatagramPacket* object) in the channel_executor (the *CachedThreadPool* executor):

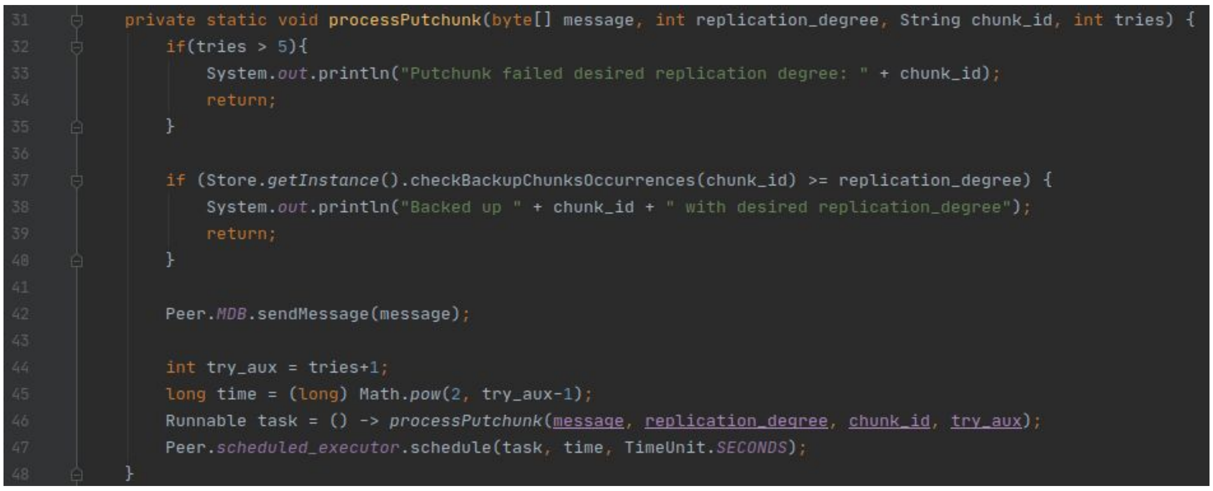


```java
    @Override
    public void run(){
        try {
            MulticastSocket socket = new MulticastSocket(this.port);
            socket.setTimeToLive(Macros.TTL);

            socket.joinGroup(this.InetAddress);

            while(true) {
                byte[] buffer = new byte[Macros.MAX_MESSAGE_SIZE];

                DatagramPacket packet = new DatagramPacket(buffer, Macros.MAX_MESSAGE_SIZE);

                socket.receive(packet);

                Runnable read_message_task = () -> readableMessage(packet);

                Peer.channel_executor.submit(read_message_task);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Picture 2: Run method of abstract class Channel

In picture 1 (line 54) it is also possible to observe the instantiation of a ScheduledThreadPoolExecutor (of java.util.concurrent library), this is the second executor.

In our implementation there is no use of Thread.sleep(), instead, we adopted the schedule method of the class ScheduledThreadPoolExecutor, which allowed to start the operation in order to satisfy the condition of sending messages at random times. This executor's approach is mainly administered in the act of sending a message (through a channel), and its usages can be found in the *protocols* module, for example, in *BackupProtocol* class:

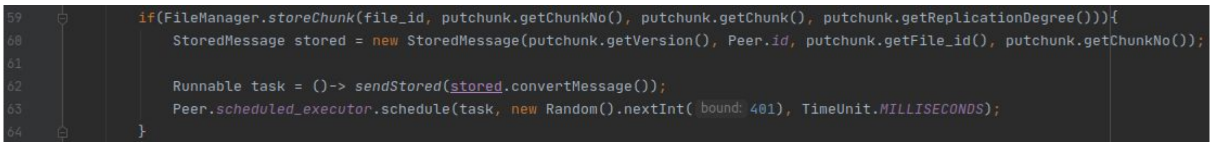


```java
31      private static void processPutchunk(byte[] message, int replication_degree, String chunk_id, int tries) {
32          if(tries > 5){
33              System.out.println("Putchunk failed desired replication degree: " + chunk_id);
34              return;
35          }
36
37          if (Store.getInstance().checkBackupChunksOccurrences(chunk_id) >= replication_degree) {
38              System.out.println("Backed up " + chunk_id + " with desired replication_degree");
39              return;
40          }
41
42          Peer.MDB.sendMessage(message);
43
44          int try_aux = tries+1;
45          long time = (long) Math.pow(2, try_aux-1);
46          Runnable task = () -> processPutchunk(message, replication_degree, chunk_id, try_aux);
47          Peer.scheduled_executor.schedule(task, time, TimeUnit.SECONDS);
48      }
```

Picture 3: processPutchunk method of BackupProtocol class located in project.protocols module

In line 42, the message is being sent through the Multicast DataBackup Channel (MDB; declared and instantiated in Peer), in line 47, the same task (referring to the current function) is scheduled to repeat itself, however with new arguments, through the ScheduledThreadPoolExecutor *scheduled_executor* (also declared and instantiated in Peer).

Another example of its usage is in waiting a random time to send a message ("STORED" message, in the following example, line 63):



```
59        if(FileManager.storeChunk(file_id, putchunk.getChunkNo(), putchunk.getChunk(), putchunk.getReplicationDegree())){
60            StoredMessage stored = new StoredMessage(putchunk.getVersion(), Peer.id, putchunk.getFile_id(), putchunk.getChunkNo());
61
62            Runnable task = ()-> sendStored(stored.convertMessage());
63            Peer.scheduled_executor.schedule(task, new Random().nextInt( bound: 401), TimeUnit.MILLISECONDS);
64        }
```

Picture 4: Code fragment for sending a STORED message, method *retrievePutchunk* in *RestoreProtocol* class of project.protocols module.

We also eliminated all blocking calls accessing files of the file system. For that we use the class java.nio.channels.AsynchronousFileChannel, instead of fileInputStream, fileOutputStream and RandomAccess, that block read() and write() operations. For example, the operation read() returns -1 if no byte is available because the end of the stream has been reached. However, this method blocks until input data is available, the end of the stream is detected, or an exception is thrown, making it a blocking call.

Enhancements

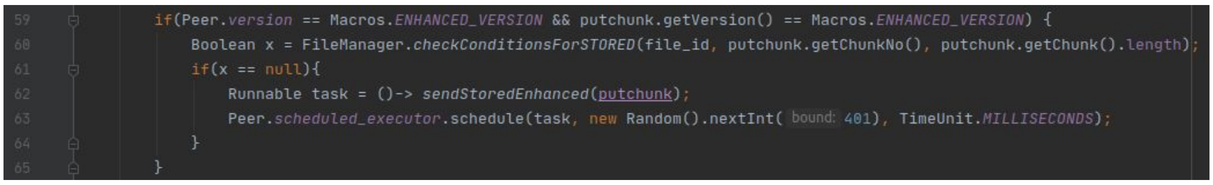
The project developed allows the possibility to run a distributed backup service in two versions: the version 1.0 and the improved version 2.0. The enhanced version is described in the sections below.

Backup

In order to avoid causing too much noise on other peers, the established solution is:

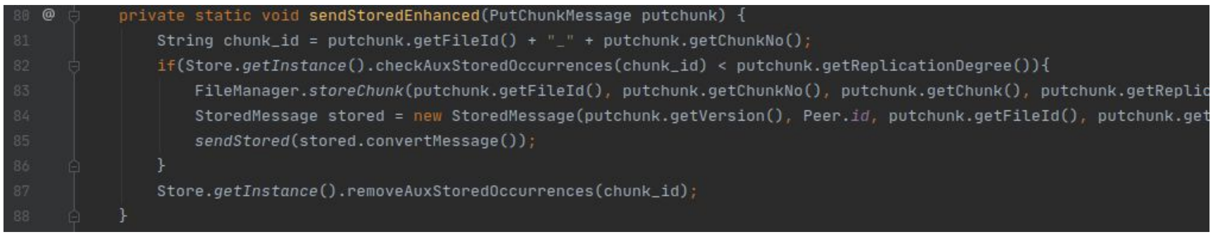
1) Initiator peer A requests the backup of a file
2) when a neighbour peer B receives one the PUTCHUNKs, the task of storing the chunk and sending a STORED message suffers a delay of random value between 0 and 400 ms (using a scheduled thread)
3) meanwhile, the STORED messages received by peer B (thanks to the implemented concurrency) will be registered in the ConcurrentHashMap named aux_stored_occurrences
4) when the task previously mentioned in step 2 is executed, the peer verifies the number of STORED messages registered in the aux_stored_occurrences relative to that specific chunk (by its file id and chunk number), if that number is less than the replication degree acknowledged in the PUTCHUNK, then the chunk is stored and the message STORED is sent, otherwise, such does not happen.

This is due to the fact that the replication degree desired by peer A will be accomplished by the stored messages sent by other peers and, therefore, there is no need for another peer to store it.



```java
    if(Peer.version == Macros.ENHANCED_VERSION && putchunk.getVersion() == Macros.ENHANCED_VERSION) {
        Boolean x = FileManager.checkConditionsForSTORED(file_id, putchunk.getChunkNo(), putchunk.getChunk().length);
        if(x == null){
            Runnable task = ()-> sendStoredEnhanced(putchunk);
            Peer.scheduled_executor.schedule(task, new Random().nextInt( bound: 401), TimeUnit.MILLISECONDS);
        }
    }
```

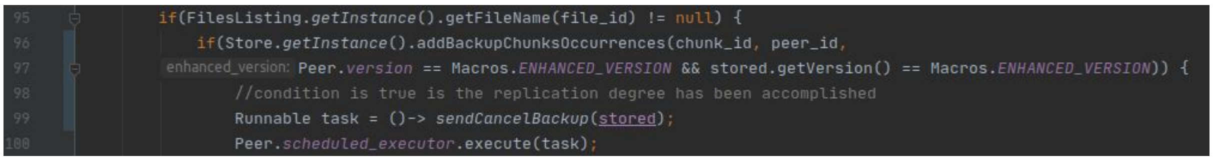
Picture 5: If both peers (initiator and receiver) are in the enhanced version, *sendStoredEnhanced* method is scheduled (step 2)



```java
private static void sendStoredEnhanced(PutChunkMessage putchunk) {
    String chunk_id = putchunk.getFileId() + "_" + putchunk.getChunkNo();
    if(Store.getInstance().checkAuxStoredOccurrences(chunk_id) < putchunk.getReplicationDegree()){
        FileManager.storeChunk(putchunk.getFileId(), putchunk.getChunkNo(), putchunk.getChunk(), putchunk.getReplic
        StoredMessage stored = new StoredMessage(putchunk.getVersion(), Peer.id, putchunk.getFileId(), putchunk.get
        sendStored(stored.convertMessage());
    }
    Store.getInstance().removeAuxStoredOccurrences(chunk_id);
}
```

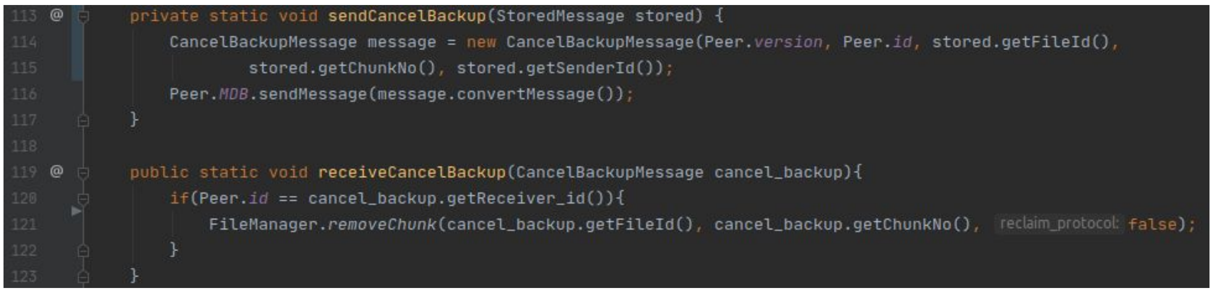
Picture 6: Chunk is only stored if the replication degree for the chunk has not been accomplished

Although it is a big efficiency step, this does not completely ensure that the exact replication degree is guaranteed. To do this, we created a new type of message: CANCELBACKUP. This message is sent when peer A receives a STORED message but the replication degree of the respective chunk has already been fulfilled. It identifies its receiver directly with the purpose of removing the specified chunk in the peer's storage.



```java
    if(FilesListing.getInstance().getFileName(file_id) != null) {
        if(Store.getInstance().addBackupChunksOccurrences(chunk_id, peer_id,
    enhanced_version: Peer.version == Macros.ENHANCED_VERSION && stored.getVersion() == Macros.ENHANCED_VERSION)) {
            //condition is true is the replication degree has been accomplished
            Runnable task = ()-> sendCancelBackup(stored);
            Peer.scheduled_executor.execute(task);
```

Picture 7: Code fragment calling a thread to cancel the backup of chunk in a specified peer (method *receiveStored*, class *BackupProtocol* in module *project.protocols*)



```java
private static void sendCancelBackup(StoredMessage stored) {
    CancelBackupMessage message = new CancelBackupMessage(Peer.version, Peer.id, stored.getFileId(),
            stored.getChunkNo(), stored.getSenderId());
    Peer.MDB.sendMessage(message.convertMessage());
}

public static void receiveCancelBackup(CancelBackupMessage cancel_backup){
    if(Peer.id == cancel_backup.getReceiver_id()){
        FileManager.removeChunk(cancel_backup.getFileId(), cancel_backup.getChunkNo(), reclaim_protocol: false);
    }
}
```

Picture 8: Sending and receiving the *CancelBackupMessage* (class *BackupProtocol* in module *project.protocols*)

Restore

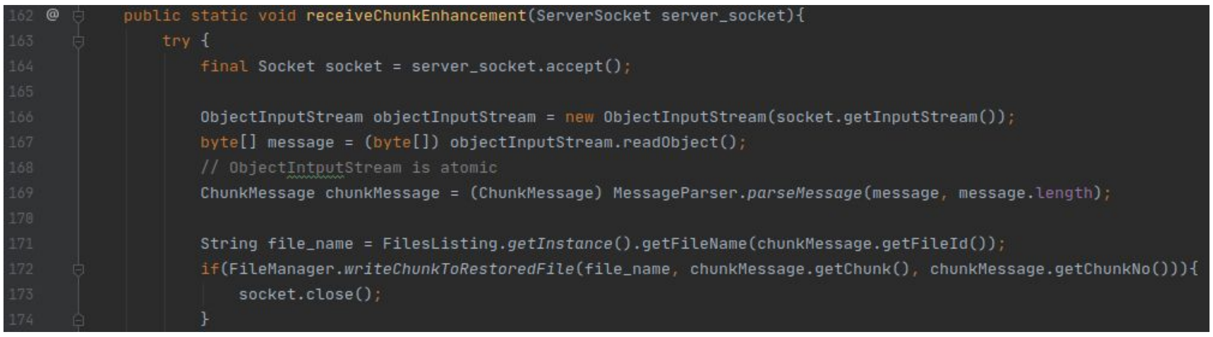
The basic version (1.0) exposed an undesirable problem when chunks were large: only one peer needs to receive the chunk, however the channel used to send the chunk is multicast. Our solution to the problem uses TCP as it is scalable and interoperable, therefore allowing cross-platform communications among large and heterogeneous networks.

First, for each GETCHUNK to be sent, a ServerSocket is opened using the constructor ServerSocket(0), sending the value 0 (referring to the port) binds the object to a free port, this is essential considering that two server sockets shall not have the same port.

We created a new message type: GETCHUNKENHANCED. This message has the same structure as GETCHUNK with two additional header parameters: port and address.

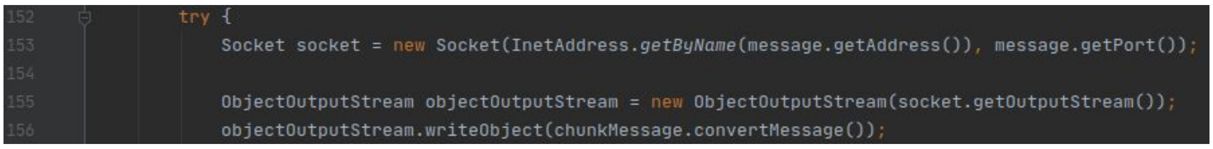
These two will be used by a peer who has the desired chunk to connect to the designated socket; the port value can be obtained through the method getLocalPort(); the IP address of the machine where the peer is running is fetched using the method InetAddress.getLocalHost().getHostAddress(). This message is sent via Multicast Control Channel. (This part is all done in *processGetChunkEnhancement*, class *RestoreProtocol* in module *project.protocols*)

In this case, the initiator peer is the equivalent of a "server" in is relation with the other peers. The "clients", after receiving the message GETCHUNKENHANCED, verify if they possess the desired chunk, and if so, try to connect with the socket using the address and port received. Since it can only connect one client at a time, if a peer tries to connect and fails to do so it means another peer has already connected. The peer that was able to connect to the socket sends the CHUNK through it using an ObjectOutputStream (the initiator peer receives it using a ObjectInputStream). This way the file is restored without overloading the multicast channel.



```
public static void receiveChunkEnhancement(ServerSocket server_socket){
    try {
        final Socket socket = server_socket.accept();

        ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputStream());
        byte[] message = (byte[]) objectInputStream.readObject();
        // ObjectIntputStream is atomic
        ChunkMessage chunkMessage = (ChunkMessage) MessageParser.parseMessage(message, message.length);

        String file_name = FilesListing.getInstance().getFileName(chunkMessage.getFileId());
        if(FileManager.writeChunkToRestoredFile(file_name, chunkMessage.getChunk(), chunkMessage.getChunkNo())){
            socket.close();
        }
```

Picture 7: Executed in a thread, this method is called by the initiator peer in *processGetChunkEnhancement*, in class *RestoreProtocol*, it describes the process of holding a connection to a "client" peer and receive its response.



```
try {
    Socket socket = new Socket(InetAddress.getByName(message.getAddress()), message.getPort());

    ObjectOutputStream objectOutputStream = new ObjectOutputStream(socket.getOutputStream());
    objectOutputStream.writeObject(chunkMessage.convertMessage());
```

Picture 8: Code fragment of method *receiveGetChunkEnhancement*, in class *RestoreProtocol*, where a "client" peer tries to connect to the socket using the address and port received, as well sending the message if the connection is successful.

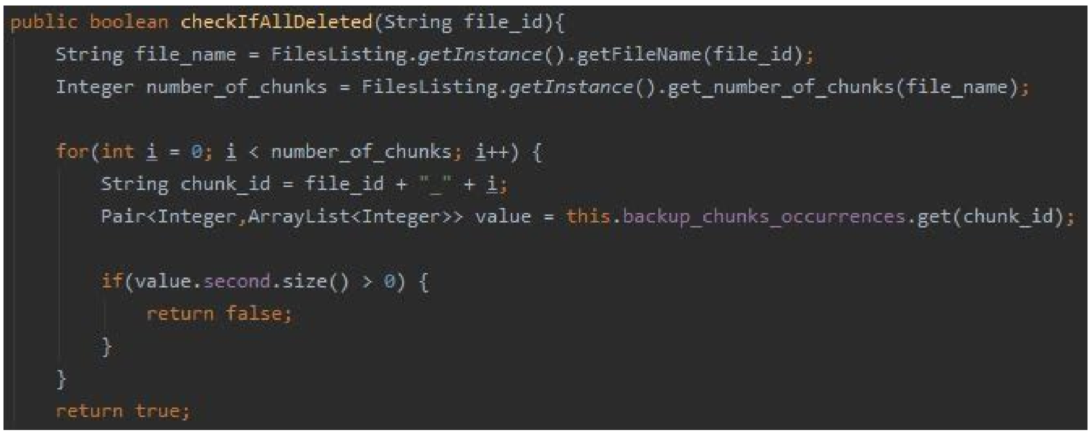
Delete

Both protocols, basic and enhanced, are demonstrated in class DeleteProtocol, in module project.protocols.

The problem of the version 1.0 presented as: "If a peer that backs up some chunks of the file is not running at the time the initiator peer sends a DELETE message for that file, the space used by these chunks will never be reclaimed." To solve this problem was necessary, among other actions, to create a new message, that was named DELETERECEIVED. As the name would intuitively let us know, this message is sent after the reception of the delete message, to inform the initiator peer that a peer X has, in fact, deleted the message.

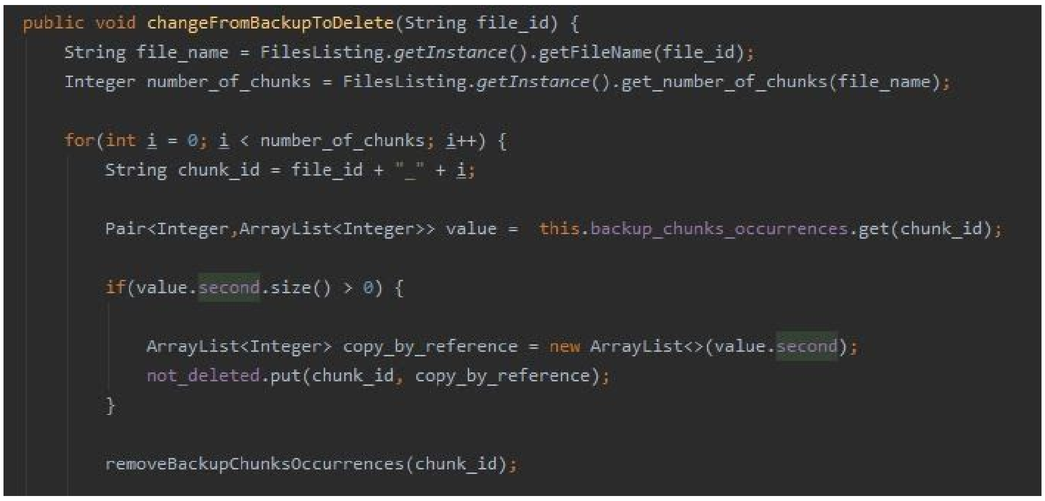
The version 2.0 of the protocol investigates if all peers that stored the chunk responded with the previously indicated message. If all peers have deleted the chunks of the file with file_id sent in the DELETE message, the protocol ends. If that is not the case, we try again, expecting that the peer connects to the network in the time between the last try and the current try. The delay increases the triple between each attempt. After the protocol has sent ten DELETE messages it stops trying and changes the puts the ids of the peers that haven't responded in the ConcurrentHashMap *not_deleted*, to keep the records.

Another idea we had, but we did not find applicable to the dimension of this current problem, was, in addition, inducing each peer to, periodically, verify the validity of each chunk, dispatching a message to the peer who requested its backup. If that peer responded by stating that the chunk should no longer exist, then the peer deletes it; if the peer has not responded at all, then we try again after some time, and perhaps delete it if we reach a sufficient number of attempts.



```java
public boolean checkIfAllDeleted(String file_id){
    String file_name = FilesListing.getInstance().getFileName(file_id);
    Integer number_of_chunks = FilesListing.getInstance().get_number_of_chunks(file_name);

    for(int i = 0; i < number_of_chunks; i++) {
        String chunk_id = file_id + "_" + i;
        Pair<Integer,ArrayList<Integer>> value = this.backup_chunks_occurrences.get(chunk_id);

        if(value.second.size() > 0) {
            return false;
        }
    }
    return true;
```

Picture 9: Code fragment responsible for verifying if all chunks of the file were deleted from all peers; Method of singleton Store, located in module project.store. This method is called in method *processDeleteEnhancement*, in *DeleteProtocol* class.



```java
public void changeFromBackupToDelete(String file_id) {
    String file_name = FilesListing.getInstance().getFileName(file_id);
    Integer number_of_chunks = FilesListing.getInstance().get_number_of_chunks(file_name);

    for(int i = 0; i < number_of_chunks; i++) {
        String chunk_id = file_id + "_" + i;

        Pair<Integer,ArrayList<Integer>> value =  this.backup_chunks_occurrences.get(chunk_id);

        if(value.second.size() > 0) {

            ArrayList<Integer> copy_by_reference = new ArrayList<>(value.second);
            not_deleted.put(chunk_id, copy_by_reference);
        }

        removeBackupChunksOccurrences(chunk_id);
```

Picture 10: Code fragment responsible for changing an occurrence from the backup to the *not_deleted* concurrentHashMap; Method of singleton Store, located in module project.store. This method is called in method *processDeleteEnhancement*, in *DeleteProtocol* class.