U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Project1 – Distributed Backup Service

T3G08

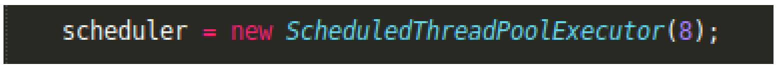
João Alves – up201605236
Tiago Cardoso – up201605762

Contents

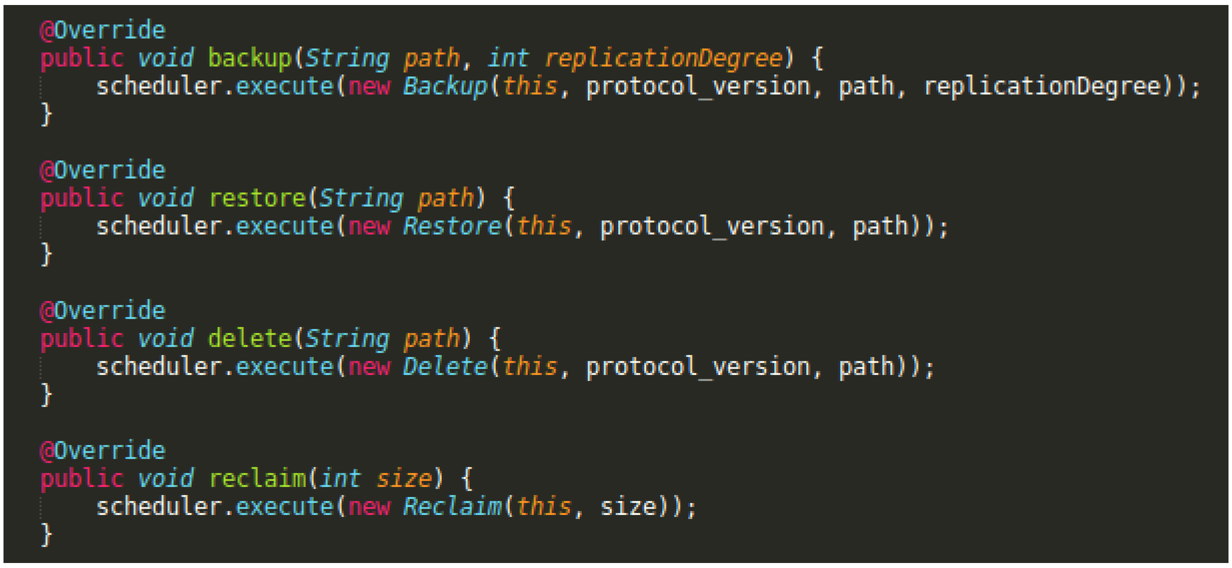
Concurrency Implementation

The approach taken to the peer-side service was to create a handler for the executed threads with *ScheduledThreadPoolExecutor,* using these type of *ThreadPoolExecutor* we are able to perform some tasks(commands) after a given delay (as required by the script).

By using the executor it is possible to, manage the multiple threads as they are running asynchronously, and to execute many protocol instances at a time.

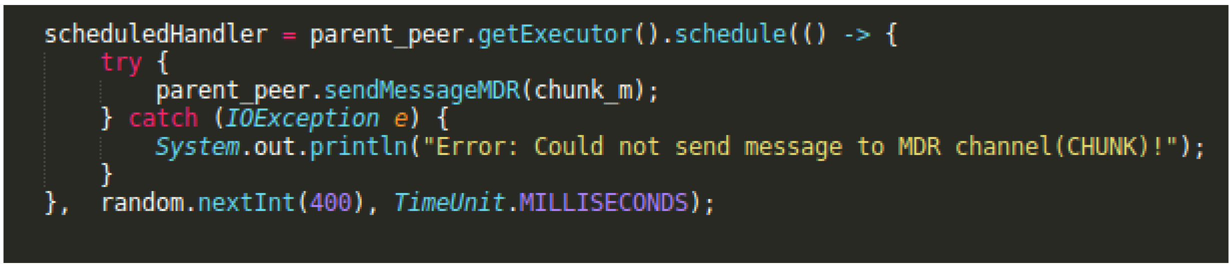


```
scheduler = new ScheduledThreadPoolExecutor(8);
```



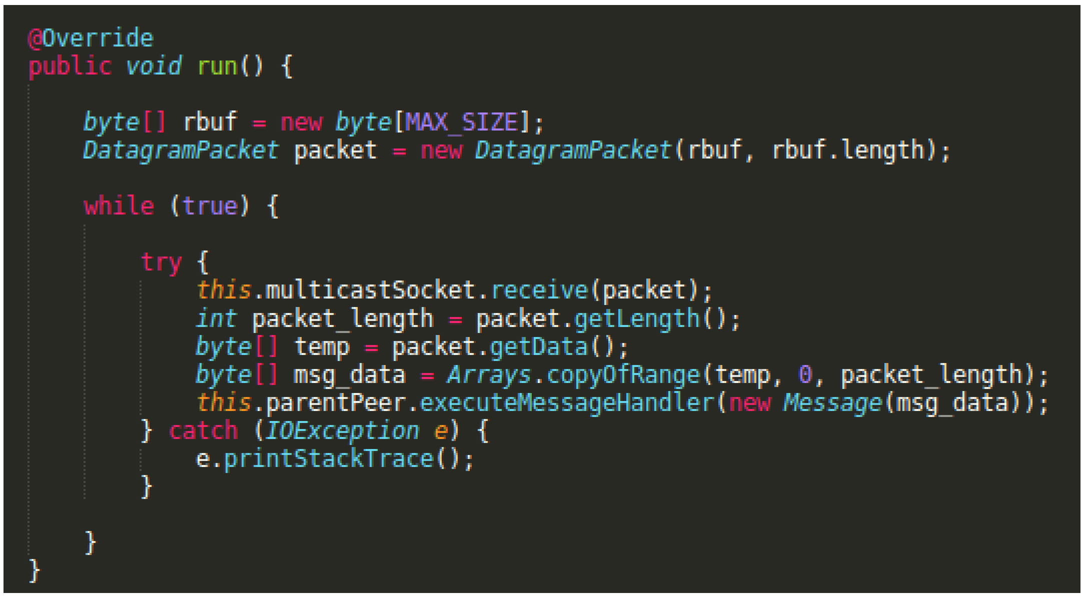
```
@Override
public void backup(String path, int replicationDegree) {
    scheduler.execute(new Backup(this, protocol_version, path, replicationDegree));
}

@Override
public void restore(String path) {
    scheduler.execute(new Restore(this, protocol_version, path));
}

@Override
public void delete(String path) {
    scheduler.execute(new Delete(this, protocol_version, path));
}

@Override
public void reclaim(int size) {
    scheduler.execute(new Reclaim(this, size));
}
```

Example on performing a scheduled task :

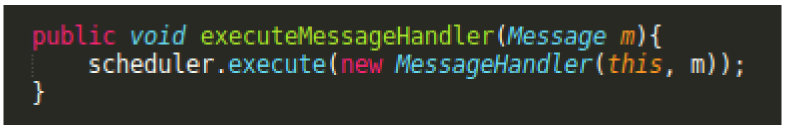


```
scheduledHandler = parent_peer.getExecutor().schedule(() -> {
    try {
        parent_peer.sendMessageMDR(chunk_m);
    } catch (IOException e) {
        System.out.println("Error: Could not send message to MDR channel(CHUNK)!");
    }
},  random.nextInt(400), TimeUnit.MILLISECONDS);
```

To support the concurrent processing of different messages on the same channel we developed a class named *MessageHandler* that runs a thread for every message received in each of the channels.
A channel is a running Thread that upon receiving a message calls the executor from the peer to execute a *MessageHandler.*

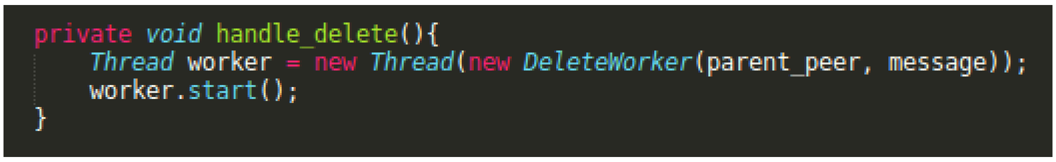


```java
@Override
public void run() {

    byte[] rbuf = new byte[MAX_SIZE];
    DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);

    while (true) {

        try {
            this.multicastSocket.receive(packet);
            int packet_length = packet.getLength();
            byte[] temp = packet.getData();
            byte[] msg_data = Arrays.copyOfRange(temp, 0, packet_length);
            this.parentPeer.executeMessageHandler(new Message(msg_data));
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

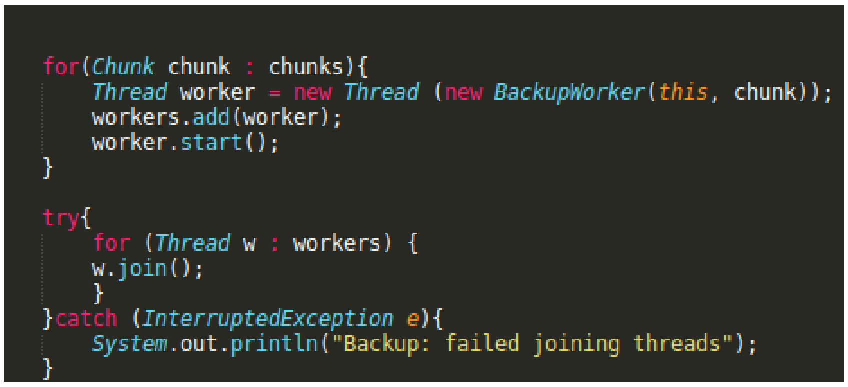


```java
public void executeMessageHandler(Message m){
    scheduler.execute(new MessageHandler(this, m));
}
```

To work with the protocols procedure we developed a class for every one (e.g. *BackupWorker)* which corresponds either to the handling of a received message (involved in the protocol) or to the execution of the current protocol thread.



```java
private void handle_delete(){
    Thread worker = new Thread(new DeleteWorker(parent_peer, message));
    worker.start();
}
```

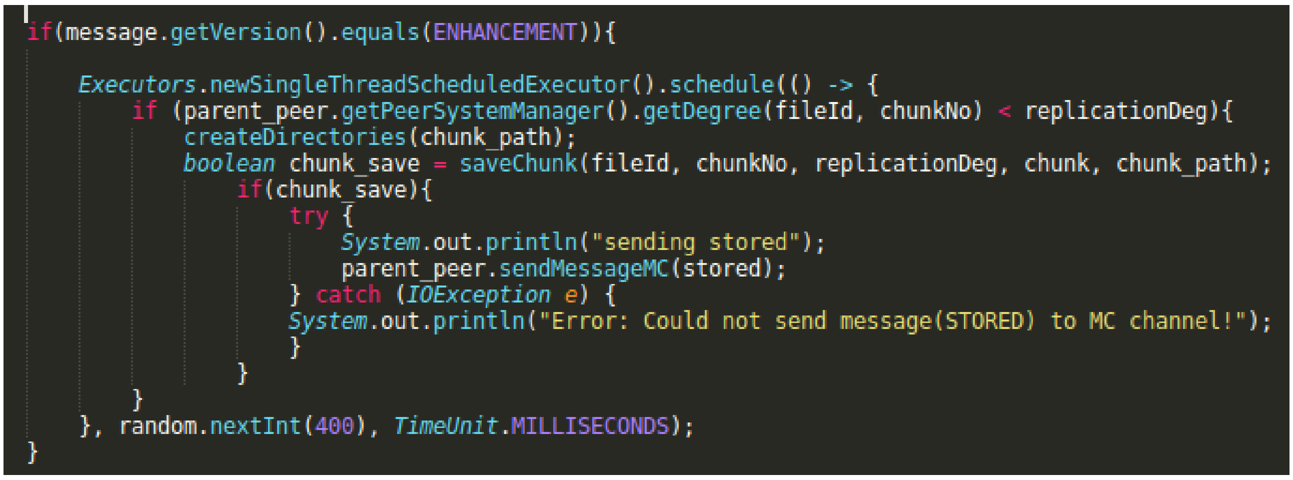


```java
for(Chunk chunk : chunks){
    Thread worker = new Thread (new BackupWorker(this, chunk));
    workers.add(worker);
    worker.start();
}

try{
    for (Thread w : workers) {
    w.join();
    }
}catch (InterruptedException e){
    System.out.println("Backup: failed joining threads");
}
```

Protocol Enhancements

"**Enhancement:** This scheme can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full. Can you think of an alternative scheme that ensures the desired replication degree, avoids these problems, and, nevertheless, can interoperate with peers that execute the chunk backup protocol described above?"

The strategy used to make this enhancement was to make a scheduled task using the *newSingleThreadScheduledExecutor.*
Inside the task is checked if the the current replication degree is less than the desired one, if true then the commands to save the chunk and send the message (stored) are executed, else they are not.



```
if(message.getVersion().equals(ENHANCEMENT)){

    Executors.newSingleThreadScheduledExecutor().schedule(() -> {
        if (parent_peer.getPeerSystemManager().getDegree(fileId, chunkNo) < replicationDeg){
            createDirectories(chunk_path);
            boolean chunk_save = saveChunk(fileId, chunkNo, replicationDeg, chunk, chunk_path);
                if(chunk_save){
                    try {
                        System.out.println("sending stored");
                        parent_peer.sendMessageMC(stored);
                    } catch (IOException e) {
                    System.out.println("Error: Could not send message(STORED) to MC channel!");
                    }
                }
        }
    }, random.nextInt(400), TimeUnit.MILLISECONDS);
}
```