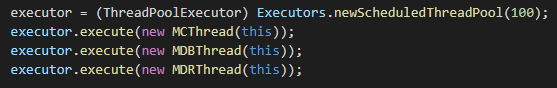
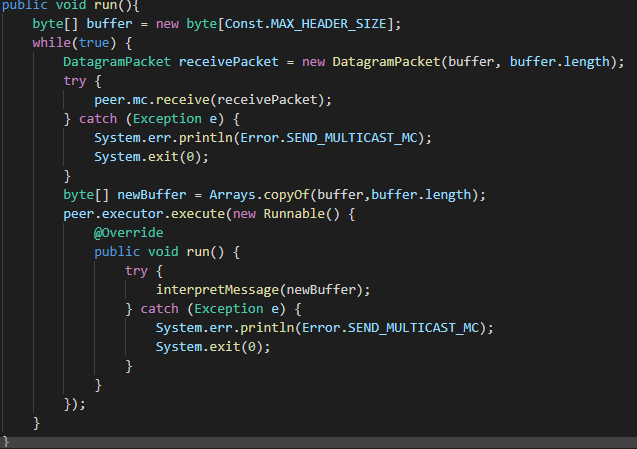
**Concurrency Execution of Protocols**

In our project, we chose to use RMI in the communication between the test application and the peer. With RMI, a new thread is created to handle every request that reaches a peer. So, with different threads to handle each request, it is possible for a peer to be executing multiple protocols at the same time.

RMI only creates different threads for each protocol, so we had to worry about each protocol being executed not to conflict with others. For each protocol, we will describe our implementation in terms of its concurrency.

Also, for each channel where information flows, we had to be careful so that information would not conflict. For that, for each channel a thread is created to listen to every message in that thread and whenever a message is received, a new thread is created to handle that message. Creating and Terminating threads has some overhead, so we used a *ThreadPoolExecutor* to execute every thread needed, being it in the handling of protocol execution request from test application or messages received on any channel.

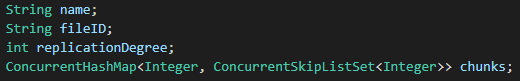
Creating *ThreadPoolExecutor* and executing the 3 channel listener threads

Receiving a message and executing a thread to handle it

**Backup subprotocol concurrency**

In the backup protocol, the file to backup is first divided into chunks and then, for every chunk, a thread is executed to backup a given chunk. So that, backed files information does not collide, we used a *ConcurrentHashMap* to store information from every file a peer has started the backup. For every file, we store its name, fileID, intended replication degree, and a *ConcurrentHashMap* with chunk information. For every chunk, we store *ConcurrentSkipListSet* with the id of every peer that stored it.



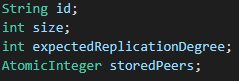
Backed up files *ConcurrentHashMap*

*BackupFile* information

The thread created for each chunk sends the backup message, waits some time (according to how many times it has tried to send a message) and then checks if the number of stored peers for that chunk is at least the intended replication degree, if not tries again (up to 5 tries). The mc channel listener thread is responsible for updating the list of servers that have stored a given chunk every time a stored message is received.

From the side of the non-initiator peers, a backup message is received for every chunk, and if the peer decides to store that chunk (if he has enough memory left) the chunk information is stored on a *ConcurrentHashMap*. For every chunk the peer has its id, size, expected replication degree and the number of peers that also stored it. It also stores the chunk data in non-volatile memory.

Stored chunks *ConcurrentHashMap*



*Chunk* information

**Restore subprotocol concurrency**

When a peer receives a restore request, it creates an entry on a *ConcurrentHashMap* of restored files with the file id, file path, the number of chunks of that file and a *ConcurrentHashMap* where every chunk data will be stored. After that, it sends the restore chunk messages. When the MDR listener thread receives a chunk, it updates the restored file entry adding an entry to the chunks *ConcurrentHashMap*. When all chunks are present, the file is rebuilt in order.



Restored files *ConcurrentHashMap*



*RestoredFile* information

**Delete subprotocol concurrency**

For a file delete request, the peer removes that file from the back up files *ConcurrentHashMap* and sends the delete message. Every peer that receives a delete file message removes all chunks it has from that file from the stored files *ConcurrentHashMap* and removes the chunk data from non-volatile memory.

**Reclaim subprotocol concurrency**

When a peer receives the reclaim space request, it updates its available memory. After that, it will remove chunks until the space it is using is below the available space. To decide which chunks to remove, it sorts them acording to how non-important they are (replication degree is higher than the expected replication degree) and the chunk size (bigger chunks are removed first). For every chunk it decides to remove, it sends the removed message to the MC channel.

Whenever a peer receives a removed message on its MC channel, it updates the amount of peer that have stored that chunk. If that number goes below the desired replication degree, it will start a backup for that chunk if no peer has started it yet.