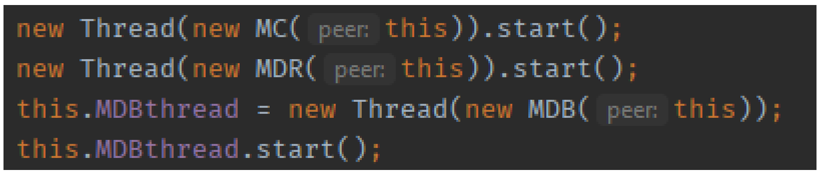
Concurrent execution of protocols

We implemented our application with support for multiple protocols running at the same time on the same peer as well as multiple instances of the same protocol at the same time ensuring consistency across all operations.

Firstly, we use three threads that run all the time:

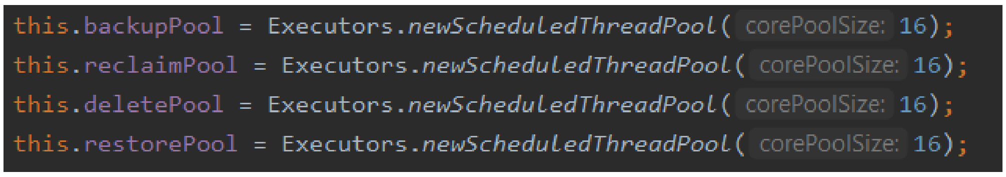


```
new Thread(new MC( peer: this)).start();
new Thread(new MDR( peer: this)).start();
this.MDBthread = new Thread(new MDB( peer: this));
this.MDBthread.start();
```

These three threads are responsible for listening to each of the multicast channels and process each message according to the protocol specified. These threads go uninterrupted through the execution of the program except for the MDBThread that may be interrupted on the enhancement of the backup protocol as we will explain below.

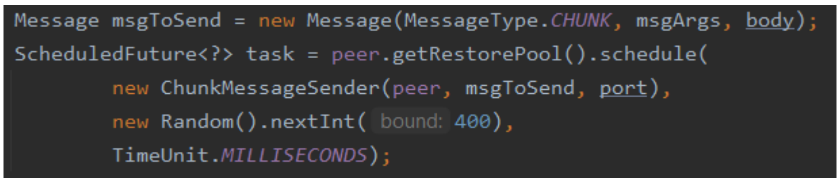
To receive and send messages on the corresponding channel, these threads make use of the **Channel** class: three Channel objects are constructed in the peer, representing the MC MDR and MDB channels, containing the **send** and **receive** methods, used in the channel threads.

Upon receiving a message, a channel checks the type of the message and chooses an operation to run accordingly. To handle multiple messages to be processed and different protocols and operations to be executed simultaneously, we used Java's **ScheduledExecutorService** interface to create **thread pools** in the peer, one for each protocol. The operations are then dispatched as a Runnable to the pool that it's related to, using the **execute()** method. This implementation with multiple pools also avoids starvation, allowing the resources to be split more correctly to different tasks.



```
this.backupPool = Executors.newScheduledThreadPool( corePoolSize: 16);
this.reclaimPool = Executors.newScheduledThreadPool( corePoolSize: 16);
this.deletePool = Executors.newScheduledThreadPool( corePoolSize: 16);
this.restorePool = Executors.newScheduledThreadPool( corePoolSize: 16);
```

Using the scheduled thread pool, we are also able to schedule certain tasks, and cancel them when necessary. For example, when sending a CHUNK message, the peer must wait a random time up to 400ms, and we use the **schedule()** method, storing the task as a ScheduledFuture object. If another CHUNK message is received before the task is executed, we execute the method **cancel** to cancel the task, preventing the transmission of the message.



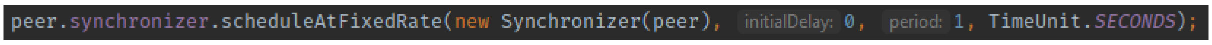
```
Message msgToSend = new Message(MessageType.CHUNK, msgArgs, body);
ScheduledFuture<?> task = peer.getRestorePool().schedule(
        new ChunkMessageSender(peer, msgToSend, port),
        new Random().nextInt( bound: 400),
        TimeUnit.MILLISECONDS);
```

Example of scheduling the dispatch of a message, in this case a CHUNK message

With this implementation we managed to clear all uses of **Thread.sleep()**, Because the use of the collections within the Peer class, for example, the *storedChunks* and *files* maps, have its contents read and modified by multiple threads simultaneously, we chose to use Concurrent Collections (java.util.concurrent), more specifically, the class **ConcurrentHashMap** for Maps and **ConcurrentHashMap.newKeySet()** for Sets.

We also have another ScheduledExecuterService with a fixed rate, used to periodically write the state/information of the peer on disk, along with checking the backed up files' states, that is, checking if any file is deleted or modified. If so, the delete protocol will be executed for that file.

This is done via the **Synchronizer** class. It saves the **serializable PeerState** class to a file on the peer storage tree that gets read on each startup and is used to recover information about backups and ongoing operations that were interrupted on the previous execution. Currently the fixed rate is 1 second.

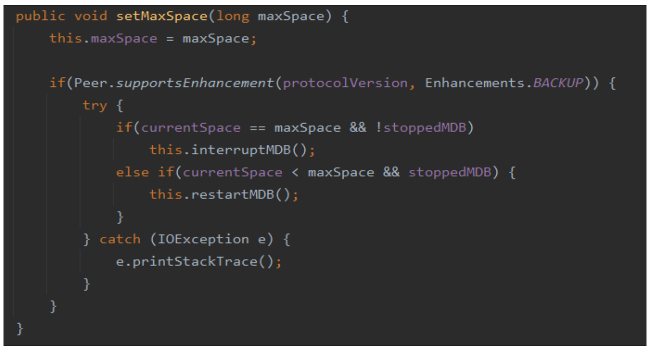


```
peer.synchronizer.scheduleAtFixedRate(new Synchronizer(peer), initialDelay: 0, period: 1, TimeUnit.SECONDS);
```

Backup protocol enhancement

When a peer executes the backup protocol, the ID of every peer that replies with the STORED message is stored in that peer. This is used to calculate the perceived replication degree on the backup initiator peer and, because the peer is informed every time one of those peers removes the chunk (through the REMOVE messages), peers that already stored that chunk and that have its capacity already full do not need, necessarily, to reply with a STORED message. And as no space is available, any PUTCHUNK message is descartable. So, we made it so that, with this enhancement, peers that have no more space available for the backup service in its storage don't listen to the MDB channel, reducing unnecessary activity.

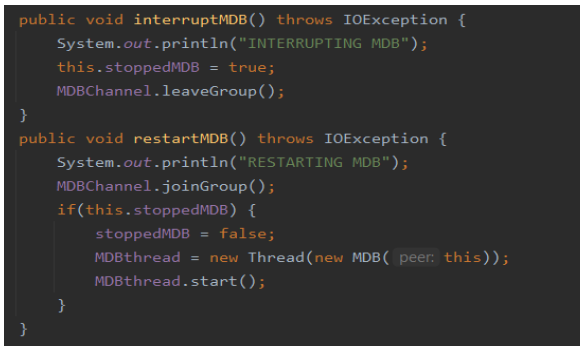
When the maximum size allocated to the backup service is modified or the peer deletes stored chunks, the MDB channel is reactivated on that peer, if space becomes available as a result.



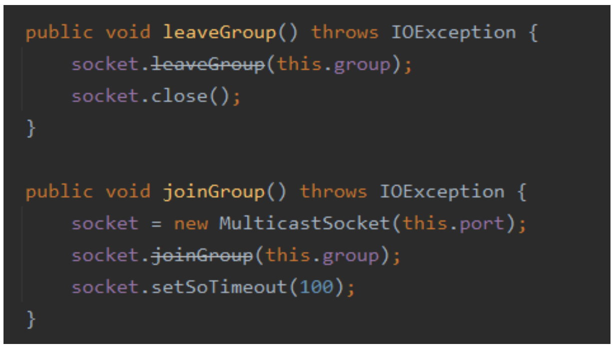
```
public void setMaxSpace(long maxSpace) {
    this.maxSpace = maxSpace;

    if(Peer.supportsEnhancement(protocolVersion, Enhancements.BACKUP)) {
        try {
            if(currentSpace == maxSpace && !stoppedMDB)
                this.interruptMDB();
            else if(currentSpace < maxSpace && stoppedMDB) {
                this.restartMDB();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 1 - setMaxSpace method in Peer class. One example of when to check to open/close MDB channel.

We implemented this by creating a flag *stoppedMDB* that, when true, stops the loop in the MDB channel thread, and so ending the thread execution. We also close the socket used for that channel in the peer. When space is available again, we create a new socket and a new thread dedicated to listening to the channel.



```
public void interruptMDB() throws IOException {
    System.out.println("INTERRUPTING MDB");
    this.stoppedMDB = true;
    MDBChannel.leaveGroup();
}
public void restartMDB() throws IOException {
    System.out.println("RESTARTING MDB");
    MDBChannel.joinGroup();
    if(this.stoppedMDB) {
        stoppedMDB = false;
        MDBthread = new Thread(new MDB( peer: this));
        MDBthread.start();
    }
}
```



```
public void leaveGroup() throws IOException {
    socket.~~leaveGroup~~(this.group);
    socket.close();
}

public void joinGroup() throws IOException {
    socket = new MulticastSocket(this.port);
    socket.~~joinGroup~~(this.group);
    socket.setSoTimeout(100);
}
```

Fig. 2 & 3 - On the left, Peer methods for interrupting and starting MDB. On the right, the Channel methods.

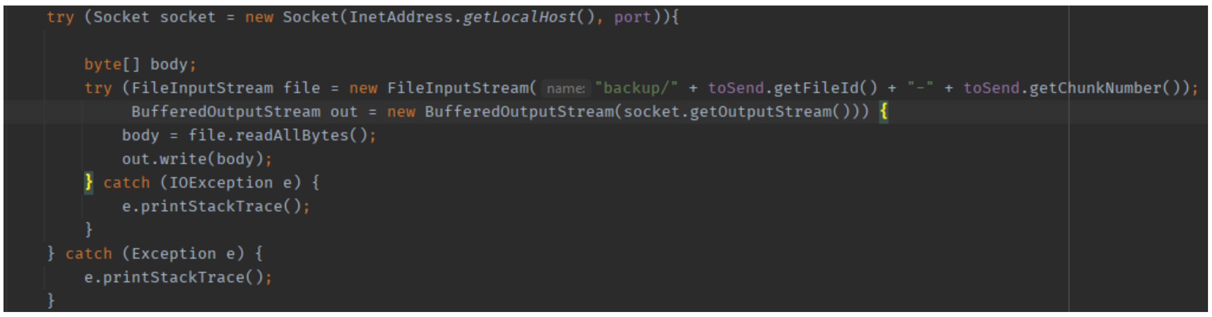
Although this implementation worked, we noticed that it would cause the reclaim protocol to not work as intended, because, when a stored chunk drops its replication degree to a value below desired, the peer must send a PUTCHUNK message and, before sending, listen to the MDBchannel for other messages for the same chunk to avoid unnecessary traffic in the channel. So, if the peer had no space available, it would not listen to the channel in this particular situation, so we created another condition that the peer would reopen the channel when this happened, closing it again after finally sending the PUTCHUNK message or when receiving that message from another peer.

Recovery protocol enhancement

The implemented enhancement intends to prevent the sending of big messages through multicast when using the restore protocol. The first change is in the restore messages themselves. For each chunk/message we open a TCP connection on a specific port chosen from a range of allocated ports to that peer. Then this port number is sent in the second line of the header of the GETCHUNK message.

Each peer then receives this message and proceeds normally according to the restore protocol (waiting between 0 and 400 ms before sending the CHUNK message). Then we have two possible situations. If the peer does not implement the enhancement then it will just send the normal CHUNK message with the body. However if the peer implements the enhacement the peer will attempt to make a TCP connection to the initiator peer. This way if we have some peers that implement the protocol and others that don't we can't predict which method will be used. This was made to ensure the required interoperability.

Once the connection is established the peer will send a CHUNK message with an empty body to alert other peers that may or may not implement the enhancement that the CHUNK will be delivered and then proceed to send the body of the chunk through the TCP connection. This way we only need to send the header through multicast and not the entire chunk.



```java
try (Socket socket = new Socket(InetAddress.getLocalHost(), port)){

    byte[] body;
    try (FileInputStream file = new FileInputStream( name: "backup/" + toSend.getFileId() + "-" + toSend.getChunkNumber());
         BufferedOutputStream out = new BufferedOutputStream(socket.getOutputStream())) {
        body = file.readAllBytes();
        out.write(body);
    } catch (IOException e) {
        e.printStackTrace();
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Fig. 4 - Sending the chunk body through the TCP connection

Delete protocol enhancement

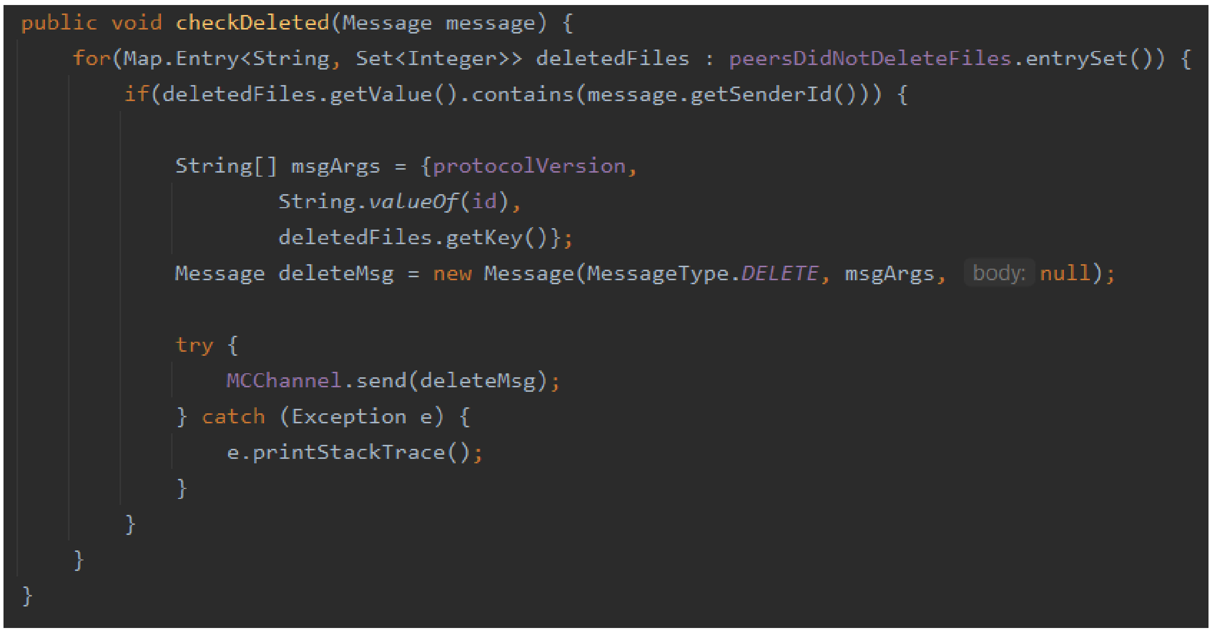
As a solution to the present problem in the delete protocol, we created a Map in the Peer class (*peersDidNotDeleteFiles*), that maps a file hash to a set of integers, that stores peer IDs and represents the peers that did not delete the chunks associated with the file. Every time a peer executes a delete protocol on a file, its hash is stored in the map, along with all the peers that have at least one of the chunks stored in its system. This information is stored in the *files* map, within the FileInfo objects.

The initiator peer still has to keep track of the peers that have already deleted it, and for that we decided to create a new type of message DELETED that follows the following structure:

<Version> DELETED <SenderID> <FileID>

This message is sent by each peer that deletes the chunks after the initiator executes the protocol, and, as it receives these messages, removes the corresponding sender from the set associated with the file, if it exists in the map. Afterwards, the IDs of all the peers that failed to receive the DELETE message or that were disconnected at the time of the execution, will be stored in the map.

To check if those peers have reconnected and execute the delete protocol again, the initiator listens to messages from them in all channels. When any message from any of those peers is received, the protocol is re-executed.



```
public void checkDeleted(Message message) {
    for(Map.Entry<String, Set<Integer>> deletedFiles : peersDidNotDeleteFiles.entrySet()) {
        if(deletedFiles.getValue().contains(message.getSenderId())) {

            String[] msgArgs = {protocolVersion,
                    String.valueOf(id),
                    deletedFiles.getKey()};
            Message deleteMsg = new Message(MessageType.DELETE, msgArgs,  body: null);

            try {
                MCChannel.send(deleteMsg);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Fig. 5 - method *checkDeleted* in Peer class, executed when receiving any message and the *peersDidNotDeleteFiles* is not empty. Used to check if the sender has chunks yet to delete.

Although, between the first execution and the reconnection of the peers, the initiator could have executed the backup protocol again. In that case, the entry in the *peersDidNotDeleteFiles* related to that file is removed, as the disconnected peers should keep the chunks upon connection.