File deletion subprotocol enhancement

In the file deletion subprotocol enhancement we try to reclaim storage space that should’ve been deleted by a peer. This is caused by a peer who failed to receive the first DELETE message that was sent.

To fix this issue, when a peer receives a delete message and is storing at least one chunk of the file it refers to, it sends an IDELETED message (created for the purposes of this enhancement). This message indicates to the initiator peer that the sender of the reply message as the deleted the file.

Furthermore, an initiator which has deleted a file keeps track of all the peers who haven’t deleted the given file using the IDELETED messages. When a peer which didn’t respond to a previous DELETE message sends any message, the initiator peer resends the DELETE message. This allows the peer to notify peers that should’ve deleted a file when it detects their activity. If the detected activity comes from a chunk that doesn’t support the enhanced version of the protocol (2.0), the DELETE message is still re-sent, but it is assumed that the peer deletes the file, because no reply is expected.

Chunk backup subprotocol enhancement

Our implementation of the enhancement of the Chunk backup subprotocol makes use of the fact that we always know when our storage changes size: saving and/or deleting a chunk. With this in mind, we can easily check if the storage is full in these places, and act based on the answer.

Each of the multicast channels is an instance of our *SockThread* class. This class has a *join* and a *leave method* that make its multicast socket join or leave the given multicast channel, respectively. When a peer's storage is full, the peer leaves the MDB multicast channel, rejoining it when its storage is no longer full.

This allows the peer to not receive/have to process backup requests (PUTCHUNK messages) when it is not possible for him to store them, thus reducing activity on nodes whose storage space is full.

Chunk restore subprotocol enhancement

When both the peer that sends the GETCHUNK message and the one that receives it support the enhanced protocol version (2.0), the body of the CHUNK message (reply of the GETCHUNK message) is changed.

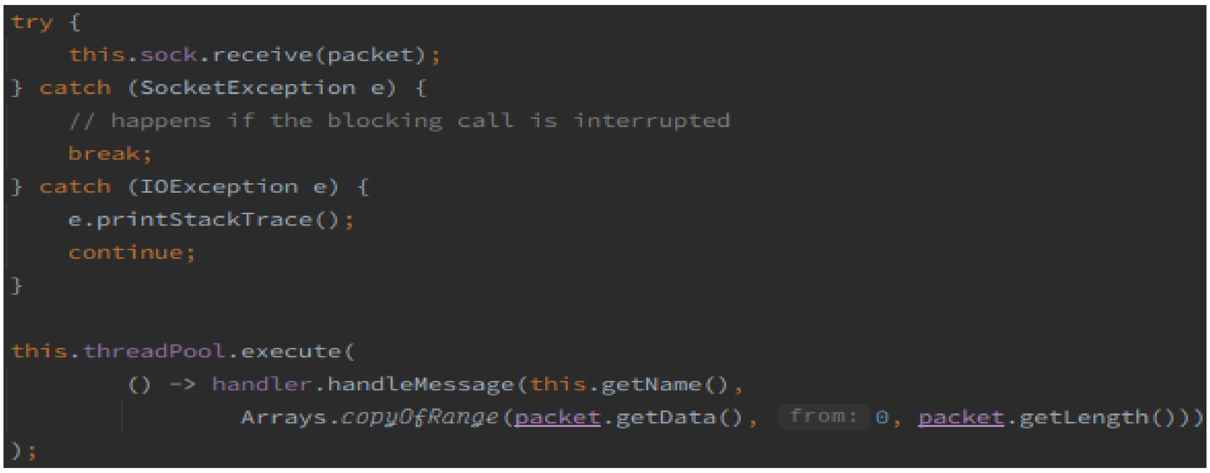
The new body of the CHUNK message will contain the IP address and the port of a TCP server socket opened by the peer that sent the CHUNK message. Upon receiving this reply, the peer the requested the chunk can open a TCP connection to the given IP address and port in order to read the chunk content.

The peers still wait a random amount of time between 0 and 400 ms before opening the TCP server socket and sending the CHUNK message in order to check that no one else already replied to that request (avoiding repeated replies). The sockets time out after 10 seconds of no one showing up.

If one of the peers in this exchange doesn’t support the enhanced version of the protocol (2.0), the CHUNK message will arrive with the chunk contents and be still be used. This means that the enhanced version of the protocol (2.0) can fully interoperate with peers both peers that don’t support the enhancement and that do.

Concurrent execution of instances of the protocols

Each instance of the *SockThread* class (that handles the interactions with a multicast socket) has a thread pool. Whenever a message is received in a socket, its *SockThread* instance dispatches a thread from its thread pool to handle that message and the respective reply.



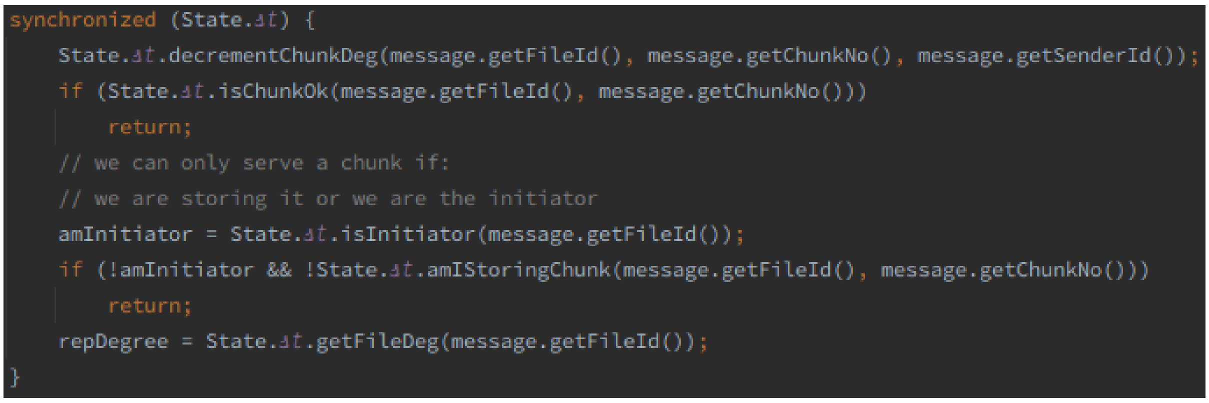
```
try {
    this.sock.receive(packet);
} catch (SocketException e) {
    // happens if the blocking call is interrupted
    break;
} catch (IOException e) {
    e.printStackTrace();
    continue;
}

this.threadPool.execute(
        () -> handler.handleMessage(this.getName(),
                Arrays.copyOfRange(packet.getData(),  from: 0, packet.getLength()))
);
```

sender/SockThread.java, line 86

The handling of messages is done by the *MessageHandler* class. All classes on the project have access to the singleton class, *State*.

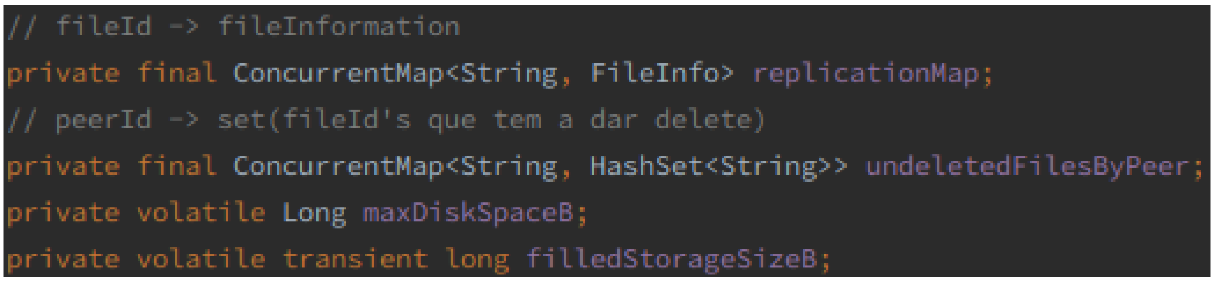
The *State* class handles all information about files (both initiated by the peer and by the other peers), chunks, storage space, and running tasks (for recovery in case of crashes). Access to this class is almost always made inside synchronized statements (similar to database transactions), which allows a thread to have full control over the reading and writing of information when handling requests.



```
synchronized (State.st) {
    State.st.decrementChunkDeg(message.getFileId(), message.getChunkNo(), message.getSenderId());
    if (State.st.isChunkOk(message.getFileId(), message.getChunkNo()))
        return;
    // we can only serve a chunk if:
    // we are storing it or we are the initiator
    amInitiator = State.st.isInitiator(message.getFileId());
    if (!amInitiator && !State.st.amIStoringChunk(message.getFileId(), message.getChunkNo()))
        return;
    repDegree = State.st.getFileDeg(message.getFileId());
}
```

sender/MessageHandler.java, line 143

There are some places in the code, where the access to the information on this class doesn't need to be made atomically, so we make of use of Java's *ConcurrentHashMap*, *concurrent.atomic* package, and volatile data members to guarantee safety.



```
// fileId -> fileInformation
private final ConcurrentMap<String, FileInfo> replicationMap;
// peerId -> set(fileId's que tem a dar delete)
private final ConcurrentMap<String, HashSet<String>> undeletedFilesByPeer;
private volatile Long maxDiskSpaceB;
private volatile transient long filledStorageSizeB;
```

State/State.java, line 18