GROUP 07

DELETE ENHANCEMENT

This enhancement is available for every version since 1.1. To make it possible we created 2 additional headers:

<Version> DELETED <SenderId> <FileId> <CRLF> <CRLF>
<Version> ON <SenderId> <CRLF> <CRLF>

When a peer receives a DELETE message, it will save, in non volatile memory, all the peers that should respond to the message, based on the peers that sent a STORED message from any chunk of that file.
Every time it receives a DELETED message, it will remove that from the waiting list.

Once a peer starts it will automatically send a ON message and all the peers that have that peer on the list of the peers that should've responded with a DELETED message will send a DELETE message for the initial file.

BACKUP ENHANCEMENT

This enhancement is available for every version since 1.2.

We saved every replication degree from every chunk that is a list of the peers that have saved that chunk.
If when trying to backup a new chunk, it's replication degree (the size of the list of peers that have saved that specific chunk) is equal or higher to its desired replication degree, the chunk is not saved. This avoids the use of unnecessary disk space.

RESTORE ENHANCEMENT

This enhancement is available for every version since 1.3.

To implement this we create a SocketServer for each of the chunks the initiator-peer is supposed to receive and create a socket from that server. That peer sends a GETCHUNK message to all the active peers via the MC channel with 2 additional parameters, the address and the port of the SocketServer (in our program we used 'localhost' and a random port in the range 8000-9000).

After receiving this message the peers that have the chunk will try to send it to the initiator-peer via the address and port specified in the message. When the initiator-peer receives the chunk it will close that server in order to ‘free’ all the peers who are still trying to send the chunk and proceed to the next chunk until the last one.

CONCURRENT DESIGN

In order for the program to handle multiple operations at the same time we used a ScheduledThreadPoolExecutor and started 1 thread for each of the 3 channels, and those threads start another one for each of the messages they receive.

Even with ScheduledThreadPoolExecuter we still have some Thread.sleep() functions which can lead to a large number of co-existing threads, each of which requires some resources, therefore limiting the scalability of the design.