# SDIS - Distributed File Backup System

Angelo Teixeira - up201606516
Henrique Lima - up201606525
Turma 5 - Grupo 11

## Concurrency

The application features a concurrent design, allowing for multiple clients executing various protocols at the same time. To achieve this, we use a *Thread Pool,* with *ScheduledExecutorService API.*

At the beginning, we create 3 listeners, one for each channel (MC, MDB, MDR), which will be infinitely listening for messages. As a message arrives, it is processed in a new thread, allowing for concurrent listening without losing new messages while processing current ones.

When a Protocol is requested through RMI to an initiator Peer, it starts processing the request in a new thread. As it receives messages, the respective processing function is called in order to change the state, and cancel some ongoing tasks, if need be. For example, to deal with the PUTCHUNK retries, a thread is scheduled with the putchunk sending, which recursively calls itself with the updated delay (0,1,2,4,8) seconds. However, if when receiving a STORED message, the replication degree is reached, the PUTCHUNK sending is cancelled. A similar approach is used on GETCHUNK/CHUNK protocol.

To know the current peer's state, including the perceived replication degree, and the ongoing tasks, we have *ConcurrentHashMap* that abstracts the concurrency problems, when trying to access it by two different threads at a given time, which could happen since each message is processed in a different thread.

Another problem we had had to do with sending two messages through the same socket at the same time, since every protocol execution is running in a different thread, that would happen quite often. To overcome this problem, we abstracted the communications in a *Communicator* class that had a synchronized block when sending messages.

## Backup Enhancement

The enhancement consisted of achieving the replication degree, without overcoming it. To achieve this, as we have a *ConcurrentHashMap* with the perceived replication degree mapped for each <File ID, Chunk No.> tuple, when receiving a STORED message, the replication degree is updated and, if the peer is the initiator, it stops sending PUTCHUNK messages, as the replication degree is reached. On the other hand, if the peer is a storer, as it waits the random time (0-400 ms), it is listening for the other STORED messages and updating its local perceived replication degree. When the random time passes, if the replication degree was reached, it cancels the storing process, thus guaranteeing the desired replication degree, without overcoming that value.

## Restore Enhancement

The enhancement consisted of using a TCP/IP socket to send CHUNKs instead of using a multicast (UDP) connection to avoid flooding the network unnecessarily with multiple CHUNK messages.

To do this, we designed a new type of message:
CHUNK 1.1 <sender_id> <file_id> <chunk_no> <CRLF><hostname> <port> <CRLF><CRLF>

This message is sent by a peer when receiving a GETCHUNK with version = 1.1.

The logic behind this enhancement is based on an FTP transfer with a passive connection, where the initiator peer asks for a chunk (sends GETCHUNK) and a peer that has the requested chunk creates a TCP socket in an available port and sends the above message, which, instead of having the chunk data directly, has the details of a TCP socket created for the purpose of sending the chunk.
The initiator peer then connects to the first socket received and reads the chunk.

Since many peers can have the requested chunk, many peers will respond with the new CHUNK message, and then wait for a connection from the initiator peer, in order to transfer the chunk. Since the initiator only receives the chunk from a single peer, the others will cancel the reading after an established timeout (15 seconds) if no peer connects to the socket.