U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Distributed Backup Service

Beatriz Mendes up201604253@fe.up.pt,
Ana Rita Norinho up201606003@fe.up.pt

Contents

1 Concurrency Design

Whilst developing this project, we tried to structure it in a way that allows the concurrent execution of protocols.

On the one hand, to store data we choose *ConcurrentHashMap* instead of HashMap. The first is more secure, scalable and has a good performance when the number of reading threads is higher than the number of writing threads. For this, we conclude that this data structure is appropriate to multithreading environments.

On the other hand, we chose to use the *ScheduledThreadPoolExecutor* instead of the use of *Thread.sleep()* when we had the to implement timeouts. For once, *Thread.sleep()* leads to a large number of coexistent threads and thus to limited scability. Therefore, everytime we needed to implement a timeout, we used the first option that allows us to schedule the execution at a specific time without execute other thread before this time runs out. So, this approach justifies our choice. There can be found several examples of this in the class Peer, namely in the restore function.

In the main method of the class Peer, it is executed one thread per channel (MC,MDB,MDR) where the messages are received and sent to the *AnalizeMessageThread* which processes them. This architecture leads to the existence of only one thread per channel.

Lastly, we take advantage of synchronization in Java language. It lets us control the access of multiple threads in shared resources. So, the synchronized keyword was used in many methods so that only one thread has access to a shared resource at a time.

2 Enhancements

2.1 Back-up Enhancement

The proposed scheme for the backup protocol leads to not only high channel activity but also to a fast occupation of all available memory. Thus, it is suggested to find a strategy to improve this protocol.

For that, the group reflected and the way we found to improve this protocol was to prevent each chunk from being saved more times than the exact necessary. For this, before saving each chunk, the Peer in question "falls asleep" during a random time between 0 and 1.5 seconds and when it wakes up it verifies if the degree of replication of that chunk has already been reached. If so, it discards it and does not save it. If not, save it by adding it to the 'savedChunks' data structure - instantiated in the Memory class - and increasing the number of times that chunk was saved, updating this value in the 'savedOcurrences' data structure - also instantiated in the Memory class. The Peer can check whether the degree of replication has already been reached through the data structure 'savedOcurrences' previously stated, which is composed of the tuple [chunkId, replication degree]. This is updated on all Peers every time one of them replicates a chunk.

2.2 Restore Enhancement

The purpose of this enhancement was to reduce the network traffic when restoring a chunk by implementing TCP to send the requested chunks directly to the Peer initiator instead of multicasting them to all peers.

To support the implementation of this enhancement, we introduced a new message: 'CONFIRMCHUNK Version SenderId FileId-ChunkNo Port'.

On the receiving end of the 'GETCHUNK' message, if the peers have record of the chunk asked for, they initiate a server socket (in a random port) and, while waiting for the Peer initiator to connect, they send a 'CONFIRMCHUNK' message with that same port. When a connection is established, the peers send the requested chunk with the same 'CHUNK' messages of the original 1.0 version.

While these procedures are running, the peer initiator, upon receiving the 'CONFIRMCHUNK' messages, collects the port field of each message stored - note that it only stores the first message reached of each specific chunk. The Peer initiator then establishes a connection with each port and received the requested chunk.

2.3 Delete Enhancement

In the file deletion protocol, it is requested that when a file is deleted from the initiator peer, it also be erased by all peers who backed it up. However, one of the problems this protocol can take is that if a peer is not running and a file that had been backed up was deleted during the peer's absence, it will still have the chunks of that file unnecessarily. Thus, it is suggested to find a strategy that even when it is not running, the chunks are deleted. To meet this requirement, the best way the group found was to create a new message to be exchanged when the peer is started. First, every time a file is deleted, its fileId is added to a deletedFiles data structure, instantiated in the Memory class. In addition, when a peer is started it sends a message with the following structure ALIVE 'version' 'senderId' and if the other peers have something in the data structure deletedFiles then execute the delete protocol for each fileId in it and so the Peer in question erases the chunks of this file if it has them. Thus, with this enhancement, peers never have unnecessary information and consequently have more free space to store relevant data.