Universidade do Porto

FEUP Faculdade de Engenharia

Distributed Systems

Project 1 - Distributed Backup Service

Master in Informatics and Computing Engineering

2020/2021

Class 7 | Group 3

Clara Gadelho | up201806309

Leonor Gomes | up201806567

Index

Chunk Backup Subprotocol Enhancement

The enhancement of the chunk backup subprotocol was to guarantee that after the desired replication degree is achieved, no more peers store that chunk, even if they receive the PUTCHUNK message.

To make this happen, it was needed to keep track across all peers of how many times each chunk is stored. This was achieved by having a ConcurrentHashmap called **storedOccurs** in each peer that increments/decrements the amount of times each chunk is stored accordingly to the STORED/REMOVED messages it receives.

Every time that a PUTCHUNK message is received, the peer waits a random time between 0 and 400ms and checks if the **storedOccurs** of that chunk are less than the desired replication degree before starting to backup the chunk. If in the meantime another peer stored that same chunk and sent the STORED message it will be registered in storedOccurs so this peer aborts the writing.

It should be noted that there is still a small chance of the achieved replication degree being bigger than the desired replication degree. If two peers start writing at a very close time, it won't be possible for them to get each others' STORED message before writing the file.

Even with that in mind, this enhancement is a good step up from the base version, allowing for significant memory savings.

File Deletion Subprotocol Enhancement

The enhancement of the file deletion subprotocol was to allow to reclaim storage space when a peer that backs up chunks of a certain file isn't running at the time the initiator peer sends a DELETE message for that file.

In order to make this happen, there was the need to implement two more messages: **AWAKE** and **DELETED**. Additionally, there were two more data structures added to FileStorage: **filesStoredinPeers** (Concurrent HashMap) and **filestoDelete** (ArrayList).

The **AWAKE** message has the following format:

<Version> AWAKE <SenderId> <CRLF><CRLF>

This message is sent when a peer starts being available using the Control Channel. This way, the other peers know that the *SenderId* is available.

The **DELETED** message has the following format:

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

This message is sent after a peer deletes a file and it sends it on the Control Channel. When a peer receives this message, it knows that the *SenderId* doesn't have chunks from that file anymore.

When a peer receives a STORED message, it updates the **filesStoredinPeers** structure by adding the file the chunk belongs to and the *SenderId* as the peer that has part of that file. Additionally, when a peer receives a DELETE message, it adds the *FileId* to the **filestoDelete** structure. This way, when a peer receives an AWAKE message, it can verify if the *SenderId* has files that need to be deleted. If that's the case, it will send again a DELETE message for it to delete the file.

Concurrency

In order for the implementation to support the concurrent execution of the protocols, the group made several choices regarding the writing of the code.

First, there's no use of *Thread.sleep()*. The use of this function can lead up to several threads running at the same time and the scalability would be limited. Therefore, when there was the need to have a timeout, the group chose to use *java.util.concurrent.ScheduledThreadPoolExecutor* that allows to schedule a thread in several parts of the project just like this one.

```
// handles GETCHUNK messages
public void handleGetChunk(){
  decomposeGetchunkHeader();
  if (Peer.getId() != senderID) { //if the peer who sent the message isnt the one who is receiving it
    Random random = new Random();
    Peer.exec.schedule(new HandleGetchunkThread(fileID, chunkNo), random.nextInt(401), TimeUnit.MILLISECONDS);
  }
}
```

In this example, after the peer receives a GETCHUNK message, it schedules a thread after a random number between 0 and 400 milliseconds.

Additionally, the group also resorted to synchronization in Java, allowing threads to access shared resources. Therefore, the use of *synchronized* in those shared resources grants threads access. One example of this is this function.

```
// verifies if the chunk is stored in the storage
public synchronized boolean isStored(String fileId, int chunkNo){

    for (Chunk storedChunk : this.storedChunks) {
        if (storedChunk.getFileID().equals(fileId) && (storedChunk.getId() == chunkNo))
            return true;
    }
    return false;
}
```

For each multicast channel, it executes a thread where it receives the messages. Every time it receives a message, the channel creates another thread to interpret the message: **InterpretMsgThread**. This way, it's possible to process several messages simultaneously and there's one thread per channel.

```
exec = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(N_THREADS_PER_CHANNEL);
controlChannel = new ControlChannel(controlAddress, controlPort);
backupChannel = new BackupChannel(backupAddress, backupPort);
restoreChannel = new RestoreChannel(restoreAddress, restorePort);
storage=new FileStorage();
```

```
exec.execute(controlChannel);
exec.execute(backupChannel);
exec.execute(restoreChannel);
```

Finally, regarding the data structures, when there was the need to use HashMaps, the implementation uses *ConcurrentHashMap* instead. This data structure is a concurrent version of a HashMap which means it has internally maintained concurrency and it's ideal when a high level of concurrency is required.

```
// string: fileID_chunkNo
// integer: number of stored occurrences
private ConcurrentHashMap<String, Integer> storedOccurr;
```

To save the state of the storage of each peer between executions, the **Serializable** mechanism provided by Java was used: the **FileStorage** of each peer is loaded from a file in the beginning of execution and written to the file at the end of execution. As each peer runs until it is stopped by Ctrl+C, it was needed to add a thread running in each peer that handled SIGINT signals and serialized the **FileStorage** before terminating the peer's execution.

```
public class EndThread implements Runnable{

    @Override
    public void run() {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            public void run(){
                Peer.storage.save();
                System.out.println("\nSaving and shutting down gracefully!");
            }
        });
    }
}
```