Distributed Systems 2020/2021 - 2nd Semester
Project 1 - Distributed Backup Service
(Group 01, Class 7)

João Sousa (up201806613@edu.fe.up.pt)
Rafael Ribeiro (up201806330@edu.fe.up.pt)

Introduction

This report's goal is to explain the specifications of not only the enhancements that were implemented (backup and delete) but also how the concurrent execution of instances of the protocols was implemented. The default service is version **1.0**, and the service with the enhancements version **1.1**.

The project was built with **Java 15**.

Backup Enhancement

The easiest way to avoid chunks from being replicated way more times than their desired replication degree is by using the approach described in the restore protocol to *'avoid flooding the host with chunk messages'*: upon receiving a PUTCHUNK message, the peer would wait *a random time uniformly distributed between 0 and 400 ms* and only sending a STORED reply if by that time the perceived replication degree is still lower than the desired one. For this we had to add another data structure to each peer, a *concurrent set that stores all chunks in the system*, that is, both the ones stored locally, and the ones backed up by other peers.

This enhancement helps decrease the number of unnecessary backups, at the cost of extra time spent on the operation, thus avoiding such a rapid backup space depletion.

Since this change only affects the receiver peers, there are no issues with interoperability, as non-enhanced chunks will nonetheless receive the PUTCHUNK messages and naively back them up.

Delete Enhancement

This enhancement's objective is to correct the situation where, according to the "base" specification, if a peer backs up some chunks of the file and a DELETE

message is sent for that file while the peer is offline, the space that was being used for it would never be reclaimed.

Since every initiator stores the peers that are storing its backed up chunks, there is a possibility to verify what peers were actually able to delete these chunks, using a reply message. In our case, 1.1 DELETED <SenderID> <FileID> serves as this verification. Because a DELETE message removes ALL chunks belonging to said file, the verification can also be on the basis of files and not chunk by chunk.

The only way to remove these faulty peers' chunks is at the moment they initiate again. An enhanced peer will, as such, broadcast a 1.1 ONLINE <SenderID> message so that any other enhanced peer that knows this one is holding dead chunks, will initiate the DELETE protocol for it's file again.

This enhancement won't compromise interoperability because the new messages are tagged with the enhanced (1.1) version, and are ignored by normal peers.

Concurrency Design

Each Peer is initialized with a ScheduledThreadPoolExecutor (pre-initializes threads, avoiding the need to create them “manually” for every situation) that is then used to call certain methods on separate threads, via the schedule or scheduleWithFixedDelay methods, allowing for the necessary delays mentioned in the project handout without the use of Thread.sleep, such as the dispatching of STORED messages that must be sent after a random delay between 0 and 400ms.

The storage of a Peer is represented by an instance of the FileStorage class. As the name suggests, it stores critical information related to its state (which chunks of which files are backed up by it and the files whose backup was initiated by that peer, among others), being necessary the use of appropriate data structures that allow operations from multiple threads simultaneously: ConcurrentHashMap and ConcurrentHashMap.KeySet(). Besides, this class is itself Serializable, allowing saving and loading the state of a Peer to non-volatile memory and therefore preserving the service information.

In the codebase, the keyword synchronized is also used for concurrency reasons, taking advantage of the synchronization in Java in order to let only one thread at a time have control over that method, of which is an example the function processPUTCHUNK used in the Backup subprotocol.

Regarding the communication channels, each of the multicast channels (Data, Control and Data Recovery) runs in a separate thread, receiving each message and then scheduling its processing to a different thread depending on the type (PUTCHUNK, DELETE,...).