Universidade do Porto
FEUP Faculdade de Engenharia

Distributed Systems

Project Report

Mestrado Integrado em Engenharia Informática e Computação

Francisco Friande up201508213@fe.up.pt
Luís Mendes up201605769@fe.up.pt

Introduction

This report aims to expose the strategies used in the project to manage the protocol concurrency issues, as well as to explain the only enhancement implemented: backup replication degree ensuring enhancement.

Backup Protocol

Given that the *"scheme [backup] can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full.",* we figured this could be a major problem in storage managing and overall performance of the project. To solve this problem, we had to come up with a solution that involves individual peer's storage system handling and some asseveration on synchronization matters.

Thus, we implemented the following architecture: each Peer will manage its own chunks storage (class Storage), using the following data structures.

```
private ArrayList<FileContent> files;
private ArrayList<Chunk> storedChunks;
private ArrayList<Chunk> restoredChunks;
private HashMap<String, Integer> blackListedChunks;
private ConcurrentHashMap<String, Integer> chunkOccurences;
private int space;
```

In it, we will explain specifically the *chunkOccurences* ConcurrentHashMap. The aforementioned java native data structure manages concurrency of different threads trying to access the same HashMap, allowing only one thread at a time to do so. Because it provides thread-safety and memory-consistent atomic operations, it's ideal for our final purpose.

To achieve it, every time a PUTCHUNK message results in an actual chunk save, Peer's local *chunkOccurences* int value for the key (file id name + "/" + chunk id) is increased; following a STORED message being sent, which will tell the other peers to update their *chunkOccurences* too. With this mechanism, all Peers have the information they need not to store chunks beyond the desired number.

With a simple verification in the PUTCHUNK handling class, of how many times has the chunk been saved and if it has achieved the desired replication degree or not, the likelihood of storing more chunks than the degree is very low, also due to the random delay [0-400ms] to the actual storing of the chunk (scheduling the PUTCHUNK handling class thread).

Protocol Thread Concurency

There were many facts that we took into account to implement a reliable architecture that allows simultaneously running threads.

As mentioned before, ConcurrentHashMap is a way to ensure safe thread concurrency, revealing a remarkable performance in access and modification management for multiple threads.

We also made use of java synchronized methods to ensure that each thread would execute each method in the best form.

In this case, each peer selected as the initiator peer by the TestApp will execute a synchronized method of the class Peer that corresponds with the protocol specified in the TestApp call via the interface provided by RMI, be it BACKUP, RESTORE, DELETE, RECLAIM or STATE.

Each Peer class holds a Scheduled Thread Pool of about 200 threads, the reason for using a scheduled pool was because we were required to initiate a series of actions with a given delay over the course of the project.

Each method of class Peer will end up sending messages over one of the three multicast channels MDB, MC or MDR as specified in the project guide, and these channels are, themselves, threads as well that are part of the global thread pool.

As soon as one of these threads receives a datagram via multicast (or, in other words, as soon as a message passes through the channel, having originated from the initiator peer's side) a call will be made to a worker thread known as HandleMessage.

This thread represents the actions of all other peers besides the initiator and will schedule the necessary tasks (possibly calling other threads) to handle the received message by type (be it a PUTCHUNK, GETCHUNK, REMOVED, etc..).

Note

It is possible to detect in the HandleMessage class the presence of a given message type not specified in the guide.

This type is PUTCHUNKREMOVED. Essentially, in order to guarantee that once we reclaimed space on a peer that it wouldn't restore the chunks that were just removed (as in this protocol, once those chunks are removed, other peers that own those chunks must decrement their amount of occurrences, and if that amount drops bellow the desired replication degree, it will send a PUTCHUNK, or in this case a PUTCHUNKREMOVED), we created this different type of message for this situation in particular.

Once a PUTCHUNKREMOVED is received, the peer will check its list of Black Listed Chunks, these are chunks that were reclaimed a moment ago by the client. If it detects that is accepting those chunks back to itself in this particular fashion, it will simply ignore the message and do nothing.