U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems 18/19

Distributed Backup Service

3MIEIC06, Group 03

Eduardo Luís Pinheiro da Silva up201603135@fe.up.pt
Francisco Teixeira Ferreira up201605660@fe.up.pt

Introduction

The aim of this report is to describe the enhancements made for the Distributed Backup Service, implemented over the standard protocol, as well as to explain the Concurrency Implementation.

Concurrency Implementation

The group implemented a solid concurrent application. The application's workload was divided in smaller units, with the purpose of being run in separate threads.

The Peer class extends Java's Thread and serves as parent to all other subclasses, therefore making them all thread like objects. In addition to this, the Peer class contains the main data structures used by the other threads as protected static members as well as other utility functions, therefore making them all available to the extending classes. Here is important to note the data members that can be modified by the threads as well. These are represented by structures of the java.util.concurrent package, more specifically by 3 ConcurrentHashMaps and 4 atomic variables (3 boolean and 1 integer):

- ConcurrentHashMap<String, int[]> backedUpChunks - This hashmap contains certain data regarding the chunks currently stored locally, more specifically their expected replication and size.
- ConcurrentHashMap<String, ArrayList<Integer>> chunksStorage - This one contains a list per chunk of the ID's of the peers that currently have that chunk stored in their disk. The keys to this hashmap and the previous one are represented by the String <fileID>-<ChunkNo>.
- ConcurrentHashMap<String, String[]> backUpRecordsTable - This table stores the data regarding each backup sub-protocol that was initiated in that peer, including the file ID, the expected replication and the number of chunks that resulted from the splitting of the file.
- AtomicBoolean changedBackedUpChunks, changedChunksStorage, changedRecordsTable - These atomic booleans are used to determine which tables (if any) are needed to be saved to the disk (i.e, the tables that were changed since the last check). This check is performed at regular intervals of 30 seconds. As such, there is only one thread at a time accessing the database. The mechanism we used to do this is Java Object Serialization since it's often quicker than reading and analyzing a text file or similar.
- AtomicInteger diskSpace - This variable is changed by the reclaim subprotocol and represents the peer's current disk space available for the backup subprotocol.

The Peer also implements the RMI service methods, such as backup, restore, delete and space reclaim, therefore the listening for client requests is handled automatically in that manner. Upon receiving such a request, the peer launches one of the threads from the package protocol.initiator, depending on the type of the request. These threads, as the name suggests, are responsible for initiating one of the subprotocols.

Another function of the Peer is to initiate the listener threads from the package protocol.listener. These threads are responsible for receiving the messages that are sent in each multicast channel, analyze part of their header and call the respective handlers. These handlers are themselves separate threads from the package protocol.handler and perform a more in-depth analysis of the message and act accordingly.

Protocol Enhancements:

- ### Backup (version 2.0)

In order to enhance the backup sub-protocol and reduce the total amount of disk space used by the service, several changes were made to the original, more specifically when handling PUTCHUNK messages.
In the newer version, a peer that receives a PUTCHUNK message will, before storing the chunk, wait a random amount of time between 0 and 400 ms. If in that period the number of STORED messages picked up by the peer that referenced that particular chunk of that particular file already complied with the replication degree indicated, the peer will refrain from storing the chunk.
Note that the capture and analysis of these STORED messages doesn't add any additional overhead since there are already other threads dedicated to these functions that update the respective information accessed by the PUTCHUNK handler threads.
In addition to this, we added a mechanism in which if a peer's disk space can't handle the storage of another chunk, it will try to evict stored chunks with a replication degree greater than the one desired. If this happens, it will send a REMOVED message to the control multicast channel.