U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Backup Service for the Internet

DISTRIBUTED SYSTEMS

T7G22

Bernardo Ferreira – up201806581@fe.up.pt

Caio Nogueira – up201806218@fe.up.pt

Diogo Almeida – up201806630@fe.up.pt

Miguel Silva – up201806388@fe.up.pt

Table of Contents

Overview

The main goal of this project is to develop a **distributed peer to peer service** that operates over the internet. The communications between the different nodes (peers) needs to provide security for both ends, assuring fault tolerance and encryption features.

To provide security in the communications, our implementation uses the class *SSLSocket* from *java.net.ssl.* Since we are not dividing the files into *chunks,* like we did for the first project, there is a limit regarding the transferred file size (500 MB).

We also addressed the scalability of the program at the implementation level, using Chord. Its implementation was based on a paper. By using Chord, it was easy to add fault tolerance to the service, by adding the *check_predecessor* protocol.

Protocols

Like the first project, the application's goal was to provide a service that allows users to backup, restore, and delete their files. Furthermore, the peers should be able to manipulate their storage available space, using the *RECLAIM* protocol.

The following table contains information about the arguments that should be passed to each protocol.

Protocol	Arguments	Output
BACKUP	<filepath> <rep_degree>	Operation's success
RESTORE	<filepath>	Operation's success
DELETE	<filepath>	Operation's success
RECLAIM	<size:bytes>	Operation's output
STATE	-	Peer State

Communication with the test application

To call the protocols, the *TestApp* communicates with the peers via RMI. The remote interface contains 5 methods (one for each protocol), overridden by the *Peer* class. Therefore, the *TestApp* can call the desired methods from the *Peer* class.

Backup

The implementation of the backup protocol allows users to backup and replicate their files over the internet once the *initiator peer* receives the path of the file to be backed up and the desired replication degree. After reading the file, it passes a *messages.BackupMessage* object through the *SSLSocket*, that connects the initiator peer with its successor in the chord network. Upon receiving this message, the successor decrements the desired replication degree of the file, adds its address to the message *header* and, if it has not been achieved (*replication_degree* field has not reached 1), issues another backup of its own to its successor.

Once the *replication_degree* reaches 1, the current *peer* sends a response to its predecessor, that will reach the initiator peer, thus concluding the protocol. All there is left to do is store the information about the peers responsible for the file in the *chord.ChordStorage* class.

Restore

The restore protocol is relatively easy, since we store in each peer's storage a *distributed hash table* that maps, for each file, a list of peers that contain a backup of this file.

Therefore, upon calling the restore protocol, the initiator peer establishes a connection with the first (online) peer that contains a backup of the desired file. In response, this second peer returns a message containing the body of the file. This protocol is then completed when the restored file is created on the initiator peer's filesystem.

Delete

The delete protocol is also simple, given that the updated information about the peers that have a copy of the files are available for the owner of those files. This is possible, due to the existence of the *distributed hash table* data structure.

Once this protocol is called, all the peers that have a copy of the desired file are notified with a *messages.DeleteMessage* object. This message does not require any response.

Reclaim

As previously mentioned, the *peers* must be able to change their available storage space. This is assured with the *RECLAIM* protocol.

Upon calling this method, the initiator peer loops through its files from larger to smaller, deleting them until the space occupied by the peer's storage is greater than the argument passed to the reclaim protocol.

For each deleted file, the initiator peer sends a messages.ReclaimMessage object to that file's owner. The owner now must handle the deletion of the file, by sending a *messages.RestoreMessage* to a node that contains that file, in order to retrieve its body. After having the file's body, all there to do is issue a backup protocol to reestablish the desired replication degree, if possible.

Concurrency design

In this project, we followed an approach like the one followed in the first project: using *thread pools* and *appropriate data structures*.

Thread pools

In the transport layer, we are using a *FixedThreadPool* with 16 *threads*. These threads are responsible for the communications with other peers in the network: accepting incoming messages and handling them.

In the application level, we are using a *ScheduledThreadPoolExecutor* that executes the chord's periodic functions: *stabilize*, *check_predecessor* and *fix_fingers.*

Data structures

In the application implementation, we used data structures appropriate for a multithreaded environment.

- The *distributed hash table* is a *ConcurrentHashMap,* which is better than a normal *HashMap* in these conditions, allowing faster and more efficient reads.
- The information about the files owned and backed up by a peer is contained in a *synchronizedList* that works similarly to an ArrayList.

These data structures make the environment of the program **thread safe**. This way, no information is lost during simultaneous executions of protocols.

Scalability

To assure that the service is scalable, we chose to implement a **Chord network**.

The Chord network implemented by the classes in the Chord package allow for a stable decentralized network with **logarithmic lookup time**. For the purposes of this project, the peers have a finger table with 10 entries, which means a maximum number of 1024 peers in the network.

Each peer in the network is identified by its unique **guid**. Once the network is initiated by its *boot peer,* it may take a few seconds before the network and all its peers are completely stable.

Furthermore, using Chord at the implementation level lowered the impact of node joins and leaves:

- Once a node joins the system, it is given a guid and a successor. The stabilize calls will then assure that the network remains stable.
- Once a node leaves the system, it passes into the other peers in the network all its files backed up, so that these files do not get lost.

Fault tolerance

To address the fault tolerance issue, we implemented chord's **check_predecessor** feature.

The peers send a periodic message to their predecessor to detect possible crashes. If the predecessor does not respond it means it has crashed. To avoid possibly losing files, we implemented the **replication degree** feature, which ensures there are multiples copies of the same file in more than one peer (if desired).