U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 1 - Distributed Backup Servers - Report

Distributed Systems

Regent Pedro Souto

Professor Rui Grandão

Class 6 - Group 6

Maria Alexandra Quintas Baía	up201704951	up201704951@fe.up.pt
Ricardo Amaral Nunes	up201706860	up201706860@fe.up.pt

Enhancement 1: Preventing unnecessary stored chunks

The backup protocol can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full. To fix this, when a peer receives a PUTCHUNK message, it waits 0 to 400ms, counting the number of STORED messages that it receives. After the waiting time, if that sum equals the desired replication degree, the peer refrains from storing the chunk.

Enhancement 2: Resolving unnecessary traffic

Large files that are sent to all peers due to a RESTORE request, when the target is always the peer initiator, can be inefficient and unnecessary. To overcome this problem, it was necessary to establish a TCP connection between the initiator peer and each peer that stores the requested chunks. In order for the peer to be able to send the content of the requested chunk to the peer initiator over the TCP connection, it needs to know its IP Address and its Local Port. The IP Address can be obtained from the received DatagramPacket. However, to obtain the local port, it needs to be sent in the GETCHUNKTCP message. This created message complements the GETCHUNK message by adding a parameter, the local port for the peers to be able to establish the connection. In order that all peers can continue to know which chunks have already been restored, they keep the logic of the default restore protocol, not sending the content of the chunk in the CHUNKTCP message.

The GETCHUNKTCP and CHUNKTCP messages have the following format:

<Version> GETCHUNKTCP <SenderId> <FileId> <ChunkNo> <LocalPort> <CRLF><CRLF>

<Version> CHUNKTCP <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>

Enhancement 3: Guaranteeing a file deletion

If a peer that stores some chunks of a file is not running at the time the initiator peer sends a DELETE message for that file, the space used by these chunks will never be reclaimed. To solve this problem, when a peer deletes a file, it stores in a map the fileID. Now, when a peer that missed the DELETED message goes online, it sends a ISALIVE message for all files that it has stored. When a peer receives a ISALIVE message, it checks if it has deleted the file, and if so, sends in response a DEAD message. By receiving the DEAD message, the peer deletes all chunks of that file.

The ISALIVE and DEAD messages have the following format:

<Version> ISALIVE <SenderId> <FileId> <CRLF><CRLF>

<Version> DEAD <SenderId> <FileId> <CRLF><CRLF>

Concurrency

To have the ability to process several messages in parallel we implemented a concurrent design. Each multicast channel has its own thread to listen to incoming messages. For each message received, the respective channel creates a thread (Task) to handle that message.
For a more scalable design we use a thread pool which allows us to avoid the creation of a new thread to process each message.

To maintain the information used for each protocol we use an appropriate data structure, the ConcurrentHashMap, so that multiple threads can operate on that data without any complications.