FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 1 - Report

Turma 7 - Grupo 5

Nuno Oliveira - up201506487
José Martins - up201504761

Sistemas Distribuídos
Mestrado Integrado em Engenharia Informática e Computação

1. Concurrency

This project supports full concurrency of the protocols specified by the use of multiple Threads and ConcurrentHashMaps for the volatile data.

Each peer has a dedicated “listener” thread for each of the multicast channels: MCListener.java, MDBListener.java and MDRListener.java

These listeners are responsible for parsing every message and start the requested protocols.

The peer’s main thread is used to interface with the client application.

For maximum concurrency, every instance of a protocol(task) is run on its own thread as well, this way we free the listeners to parse and start any other tasks that maybe be requested. Every message type has its own Task thread: PutchunkTask.java, StoredTask.java, GetchunkTask.java, ChunkTask.java, DeleteTask.java & RemovedTask.java

All data that needs to be accessed by multiple threads is kept in the peer’s object in a ConcurrentHashMap. Data about the chunks and files being backed up and the chunks and files being restored is stored in these HashMaps. For the enhanced protocols we required 2 more HashMaps to be created: one for perceived replication degrees during the BACKUP protocol, and one to keep logs of the DELETE protocols.

2. Enhancements

2.1. Backup

We chose to enhance the BACKUP protocol by controlling the way the chunks are stored. The issue that we observed was that no matter how many peers were online in the network every single one would try to store a chunk when requested, even if the desired replication degree was already achieved. This obviously leads to unnecessary storage of chunks.

In order to enhance this behaviour, each peer now checks every STORED message that crosses the MCListener and keeps track of its perceived replication degree. During the delay of [0,400] ms in the PUTCHUNK protocol, the peer will be updating its perceived replication degree of the chunk to be stored. When this delay is over, it will check if the perceived replication degree has reached the desired value, and avoid saving the chunk if true.

2.2. Restore

In order to stop the excessive traffic in the MDR channel and obey the restriction of using TCP, we decided to use the existing RMI interface to send the chunks between peers.

To do so, we simply added a method to our Remote interface that effectively does the CHUNK protocol, but in a one-way connection. This way, every time a GETCHUNK message is received by a peer with the requested chunk, this same peer creates a stub connected to the sender's endpoint. He then sends the chunk and relevant data using the new method and avoids using the heavily populated MDR channel at all, while ensuring the chunk arrives safely due to the RMI's usage of TCP.

Due to the properties of RMI, this enhancement does not work if the peers are located in different computers. However, it works if the peers are in the same computer.

2.3. Delete

The issue with the DELETE protocol was that any peer that was offline at the time of sending the message would never get a chance at deleting said chunk from its system. The best solution we came up with was to keep logs of all the DELETE messages that came across each peer during its lifetime and send them to a peer that comes online later.

To do this, our idea was to create a new message type called STATUS, that requests on the MC channel every peer to send its recorded logs so far. After waiting for a brief delay to ensure all peers sent their logs, the peer parses all the information together and executes all the DELETE protocols that weren't registered.

In the end, the peer updates his own logs to reflect these changes and stay on track with the rest of the service. However, due to time constraints, this feature was not implemented.