U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Serverless Distributed Backup Service

SDIS: First Project

MIEIC - T7G08:

João Castro Pinto up201806667@edu.up.pt

José Miguel Maçães up201806622@edu.up.pt

Concurrency

For this project, in order to deal with concurrency, our group developed a strategy that can be subdivided into three main sub-strategies.

Firstly, we made use of the ScheduledThreadPoolExecutor class as mentioned in the “Hints for Concurrency in Project 1” document provided as a course material. Our Peer object has an attribute of this type called threadExecutor, used to deal with thread execution, as the name may suggest. In our project, the threads we execute are mainly related to listening to the multicast channels, message sending handlers and message reception handlers (in addition to these, we have shutdown and cleanup threads, meant for when a peer is terminated). We implemented Runnable objects to be run by the mentioned threads. Some of these threads are directly executed using the *execute()* method of our threadExecutor, but for others, such as the message sending handlers, we schedule their execution, using the *schedule()* method with a random delay (between 0 and 400 ms) to prevent channel flooding.

Secondly, we made methods *synchronized* to make their execution thread-safe and avoid multiple accesses to information that needs to remain valid at the same time.

Finally, we used the ConcurrentHashMap class, as it is designed to deal with concurrent accesses. Sometimes we use it just as a set, making the value Boolean and interpreting True as the key being part of the set and False as it not being part of the set. This, combined with the ScheduledThreadPoolExecutor class, also allows us to cancel thread execution, as the *schedule()* method of that class returns an object of type Future, which we store as the value in a ConcurrentHashMap, making it possible to then use the method *cancel()* over the Future object. For example, when we wish to cancel the sending of a message, having scheduled it before, we can use the *cancel()* method. We use this in our project in the backup protocol. When we send a PUTCHUNK message, we schedule successive messages, as mentioned in the project handout, until enough STORED messages are received. If we achieve the necessary number of STORED messages, we cancel all messages that have been scheduled.

In addition to this, it is relevant to mention we avoid sending two or more equal CHUNK messages from different storing peers to the initiator peer when a restore protocol is in place. Each peer has two ConcurrentHashMap attributes, chunksAskedFor and chunksReceived, each working as a set. When a GETCHUNK message is sent, the initiator peer adds the chunk it refers to to chunksAskedFor. A storing peer replies with the CHUNK message and the initiator peer, upon the reception of that message, updates both chunksAskedFor, removing the entry relative to the chunk it just received, and chunksReceived, adding an entry for the same chunk. However, every peer that receives this message updates these attributes and so, before sending a CHUNK message, a peer verifies if that chunk has not been received yet. If it has, no message is sent. We use chunksAskedFor to prevent the processing of CHUNK messages by peers who have not requested them, as that verification is made before every CHUNK message is processed.

A similar strategy is in place for preventing multiple PUTCHUNK responses to REMOVED messages.

It is also worth mentioning that we do not use *Thread.sleep()* anywhere in the project.

Enhancements

We implemented every enhancement suggested.

Backup Enhancement

For this enhancement, the only change made was a verification before storing a chunk. While in version 1.0 the storing peer stores every chunk, in version 2.0 the storing peer checks if the actual replication degree for that chunk is lower than the desired replication degree and only in that case does it store the chunk. This simple modification works, because there are other threads listening and processing the STORED messages.

Delete Enhancement

For the delete enhancement, two new messages were created: BOOT and DEGREES. The BOOT message is sent by a newly booted peer to the MC Channel, requesting a copy of the replication degrees stored by an active peer. Its format is as follows:

<version> BOOT <sender_id> <CRLF> <CRLF>

Only one active peer should reply (the first one to do so) and it sends back the DEGREES message, with the id of the newly booted peer (booted_id) as a second line of the header and the replication degrees in the body of the message. It has the following format:

<version> DEGREES <sender_id> <CRLF>

<booted_id> <CRLF>

<CRLF><CRLF>

<body>

The body is a sequence of lines containing chunk ID (file ID and chunk number), actual replication degree and desired replication degree, following the format:

<file_id> <chunk_no> - <actual_rep_degree> / <desired_rep_degree>

The newly booted peer then compares it to the replication degrees it has stored, deleting chunks if their desired replication degree is 0.

The strategy to try to guarantee that only one peer replies to the BOOT message is similar to what is described in the Concurrency section, in the third to last paragraph.