Distributed Systems - First Project

Concurrency Design and Enhancements Report

Gonçalo Teixeira and André Gomes

BSc + MSc on Informatics and Computing Engineering

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

April 10, 2021

Porto

Concurrency Design

This section will cover the principles used to implement a stable and reliable concurrency design so that many operations could run simultaneously inside a Peer. We will break it down to every sub protocol to explain how and why we designed this model, but we will emphasize the design on the backup sub protocol. While we explain this design we will also take in consideration some enhancements principles to help us explain why we did what we did.

First thing we have to mention is that we have four separate executors (Thread Pools) with a fixed number of workers, defined on the `peer.Constants` class. Each task will be associated with an appropriate executor to prevent bottle-necking the Peer. We will demonstrate why this approach was necessary as we explain the sub protocols. Another thing worth mentioning is that by having three Multicast Channels, we came very early to the conclusion that we needed to start a Thread for each channel so the three could be running simultaneously and constantly receiving messages, so that's what we did. At the start of the Peer, three Threads are created and started: one for MC, one for MDB and another for MDR, these Threads do not take a worker from any executor, and they will be running until the Peer comes to a stop.

Our design, to put it simple, is made out of one triage executor and three specific executors. When a message is received, we want the channel to be quick and return to the receiving state, so we pass the packet to a triage, a `peer.Dispatcher` executable, then the dispatcher will parse the packet, retrieve the appropriate executor for that task and start said task on said executor.

Backup Sub Protocol

We will start this subsection with the client itself. The client sends a backup command to an Initiator Peer through RMI, the command takes a pathname and a desired replication degree for a file. The Initiator Peer will right away create the File ID for that requested file (we will not cover here how the ID was created but we will just say it takes the path and the modification date, so we can assume the ID is unique). After that ID step is done, it will loop through blocks (chunks) of 64000 bytes, as we don't want to read the whole file into memory (the maximum size for a file can be 64GB). For each of those blocks a `jobs.BackupChunk` job will be started on the `IOExecutor`, that job's task is to send a PUTCHUNK message, and after a delay of 1 second initially, it will start the second part of this job, which is to check if the desired replication degree was reached, otherwise it will start another `jobs.BackupChunk` job with twice the timeout, this procedure can be repeated up to 5 times.

But how does the initiator peer know if the replication degree was reached? Our design makes this very easy, using `ConcurrentHashMap` to store the chunks sent and received, we can access them across the whole program. We will not get deep inside the `files.Chunk` class as it is not the purpose of this report, the chunks have a `Set` containing the Peer's ID of the peers who sent a STORED message for that said chunk.

As we can conclude from here, while the second part of the `jobs.BackupChunk` job is being scheduled, the peer will be receiving STORED messages.

Now that we have the Initiator Peer covered we will talk about the other peers, the ones receiving the chunks. These peers start by receiving a packet, which is sent for triage, and a PUTCHUNK task is initiated. At the start of the task the peer will check if it's already storing that chunk, if that proves to be true it will wait between 0 and 400ms and send a STORED message. Otherwise it will store the Chunk received on a `savedChunksMap ConcurrentHashMap` to be accessible by other threads, then it will schedule the second part of this task with a delay from 0 to 400ms. While the second part does not start, the peer may receive STORED messages, some of which related to that chunk, thus updating the count of received confirmations. When the second part starts, the number of confirmations for that chunk will (probably) be changed, if that number is higher or equal to the desired replication degree the peer will remove the chunk from the mentioned map and exit the task, otherwise it will store the chunk locally and send a STORED message for this chunk. This prevents unnecessary storage

occupation, while also maintaining the main functionalities and interoperability, this behaviour is not expected on the vanilla version but it helps us describe the concurrency model (the enhancement is described on the next section).

Note: The sleep time is also enhanced - if the peer is enhanced the lower bound for the sleep depends on its occupation rate, a trigonometric function was used:

$$lower = 400 * sin(Occupation * 1.5), 0 \leq Occupation \leq 1.0$$

Now, in the case where we have only one general executor and one triage executor, and that general executor has 64 workers available. If the requested file can be split into 20 chunks, there will be no problem at all, the chunks will be received and the STORED messages will be somewhat handled without trouble, leaving some chunks with higher replication degree than desired but nothing too extreme. Now let's assume the file has 128 chunks, the initiator peer will occupy the 64 workers sending 64 chunks simultaneously, leaving no workers available to process incoming messages, the other peers will receive 64 chunks leaving no more workers left to handle STORED messages. Some of the STORED messages would get through, but not as much as we wanted, and the result would be that a very high percentage of the chunks would have higher replication degree than desired, but not even the peers would know about that.

So our solution was to create 3 general purpose executors: one to handle I/O operations, one to handle request operations and another one to handle responses/acknowledgements operations. As in a system with 5 peers a PUTCHUNK message would result in receiving something around 4 STORED messages by each peer, the number of workers for the responses executor is much higher than the number of workers for the requests and I/O executors.

At this point, the model for concurrency has been explored as the next sub protocols follow this design. To make the model even clearer the details for the remaining sub protocols will be also be explored in terms of concurrency.

Restore Sub Protocol

The key concepts for our concurrent design, described above, can be applied here as well. Starting by the client interface once more and sending a restore command through RMI to the Initiator Peer. The Initiator Peer will then start a restoration for the requested path. We are keeping a map between the file's path and its ID, as the hash is not reversible. A `files.FutureFile` will be created for this file, the purpose of this class is to retrieve all the chunks which have the same ID as the file. These chunks will be retrieved by Multicast or, if the peers (receiver and sender) are enhanced, by TCP, the way this enhancement is designed will be explained on the appropriate section, further on this report.

Again, this protocol follows a similar behaviour as the backup protocol: a peer receives a GETCHUNK message, waits for a period of time, and then sends back a message. Every sleep in our program (except for the one in the `jobs.RestoreChunk` class) were removed, adding instead a scheduler to the second part of the task, this allows the peer to work on other threads while waiting for the scheduled task to begin. Taking advantage of that, the protocol defines that if the peer receives a CHUNK message for a chunk which is being prepared to be sent (i.e. waiting for the scheduled task) it should cancel that task as another peer is already providing the requested chunk, this avoids flooding the network with large datagram packets.

The peers' information on sent chunks is accessible throughout the concurrent threads, so a flag was the only thing we needed to make this requirement work: when a peer receives a CHUNK message, if they have the chunk stored, the peer will set the chunk as `already_provided`, when the scheduled second part for the GETCHUNK message is triggered to start, it will verify the chunk is already provided by another peer so it will not provided it again. We have also added a `being_handled` flag to ensure a GETCHUNK message is not processed a second time while the first one is still being processed.

The chunks are sent to the initiator peer in a random order, so we had two options. The first one was to receive the chunks, store them on the disk, sort the received chunks, read them from the disk and write them to a file in append mode while also deleting them in the process, we can already spot some problems with this method: one problem is that we were making too much I/O operations, and this was very costly in terms of I/O management, and it would also impact the concurrency model in a negative way; another problem is that for a big file (number of chunks on the house of the thousands), those operations would be very inefficient in terms of time complexity.

The solution was a more natural implementation, instead of receiving, storing, reading and writing a chunk, we used a `RandomAccessFile` to write the chunks as they are received. We can calculate the offset using the chunk number and the body length making the implementation of this method effortless compared to the first option.

Delete Sub Protocol

We did not made any implementation worth mentioning regarding the Delete sub protocol, it's very straightforward, we just send the DELETE message 4 times and hope the message goes through to every peer listening. The enhancement regarding this sub protocol will be explored on the next section.

Reclaim Sub Protocol

We think this sub protocol is very much worth mentioning as the implementation was left for us to implement. One key aspect of this protocol is that when the peer starts a reclaiming sub protocol it will prevent PUTCHUNK associated tasks to start for 60 seconds, this is due to the fact that other peers may start a new backup for a chunk whose replication degree dropped below the desired, and the peer may still be freeing space on the reclaim sub protocol. We believe 60 seconds is enough time to free the necessary amount of disk space. Concurrency here was designed to keep the peer's state to block PUTCHUNK tasks for 60 seconds, using a `Timer` class.

Notes regarding Sub Protocols

All sub protocols implementations and notes on concurrency design are deeper explored on the javadoc pages available here.

Relevant Classes and Methods

In order to achieve the "perfect" implementation, many thread safe classes were used, with a special mention to the `ConcurrentHashMap` class, this class was very helpful as we needed to access some maps concurrently, those maps were used especially on the `peer.PeerInternalState` class. We also needed to access a specific chunk's peer list, containing the IDs of the peers who have stored that chunk, so that structure needed to be thread safe, we achieved that by a method of `ConcurrentHashMap`, as no `ConcurrentHashSet` exists as of Java 11, the mentioned set was obtained by invoking `ConcurrentHashMap.newKeySet()`.

To handle I/O operations on the chunks (reading and writing), Java's Non-blocking I/O was used, so there's no need for the typical `FileOutputBuffer` which can lead to some problems. To delete empty folders (caused by DELETE messages) and to calculate the occupation for the peer we used a Walker, present on the Java's NIO package.

In sum, we had no errors whatsoever testing the program thoroughly regarding the concurrency on accessing resources.

Enhancements

Enhancement 1 - Backup Sub Protocol

In sum, our implementation takes in consideration the STORED messages received from other peers to determinate if the chunk needs to be stored or not.

In more detail, when the initiator peer sends a chunk to backup, the target peers start the associated task to backup the chunk. They first verify if the chunk is already saved, and if not it will add the chunk to the saved chunks map temporarily to be accessed by other threads, and schedules a second part for that backup chunk task with a delay between 0 and 400 milliseconds (the sleep was also enhanced, and we will describe it further on). Meanwhile, other peers may be storing the same chunk and sending back STORED messages, the peer will save that information on the chunk previously stored on the sent chunks map. When the mentioned second part starts, it will verify if the number of STORED messages received for that chunk is higher or equal to the desired replication degree, and if that proves to be true, it will exit, as no action is need since the chunk is already being replicated as many times as required. If that condition proves to be false, it will proceed with the default behaviour.

Regarding the sleep enhancement mentioned before (and also mentioned on the previous section), we thought it would be a good feature to dynamically change the sleep lower bound on the PUTCHUNK associated task according to the peer's occupation rate. This will prove useful to maintain a (probably) uniform chunk distribution among the peers. In the worst case scenario, for a requested backup with a replication degree equal to 1, one peer will store every chunk requested, of course this is an highly improbable scenario, but nevertheless possible, so this small "enhancement" helps with that.

Regarding interoperability, this sub protocol enhancement will not, in any way, interfere with the implementation from non-enhanced peer versions, as the behaviour scope is local and does not depend on the other peers to maintain it's functionalities.

Enhancement 2 - Restore Sub Protocol

This enhancement came very naturally, given the TCP usage requisite, on a wider scope, the peers saving the chunks are the servers and the initiator peer is the client making connections to receive the chunks.

Only the initiator peer needs to read the chunk's data, hence sending the chunk's data through a multicast connection is not ideal. To prevent flooding the network, this enhancement was implemented using the Transmission Control Protocol.

The principle is very simple: the initiator peer sends a GETCHUNK message for a chunk, the peers who are saving that chunk enter the associated task and after some confirmations (such as verifying if the chunk is already being provided by another peer or handled by this peer) it will open a server socket with the next port available on the system, then it will send through the MDR channel an enhanced CHUNK message, not containing the body of the chunk but the local IP address and the newly created socket's port. The initiator peer will then receive that CHUNK message and will try to connect to the server socket, then it will start receiving the chunk's body over TCP.

Regarding interoperability, unlike the previous enhancement, this one depends on other peers to maintain its functionality, thus if any of the peers is not enhanced, it will not transmit the data over TCP, rolling back to the vanilla version. This behaviour can be determinated by the peer's version, defined on peer's execution arguments, or by the message's protocol version.

Enhancement 3 - Delete Sub Protocol

The problem with the Delete Sub Protocol was that when a DELETE message for a chunk is issued, some peers who are storing that chunk may be offline, and when they come online once again the information they have is incorrect, and they shouldn't be storing that chunk anymore.

To prevent this behaviour a GENERALKENOBI message and associated task was created (the meaning of this message will be explained further on). The purpose of this message is to, when an enhanced peer comes online, it will send this message to any listening peers, if an enhanced peer receives a GENERALKENOBI message it will check if it has deleted files (requested by the client) and if that proves to be true it will send a DELETE message (3 times, as UDP is unreliable). This prevents misinformation when any peer comes online.

GENERALKENOBI's message name origin: the natural representation for this enhancement would be an "Hello There" to every peer listening so it was only natural to name this enhancement after Obi-Wan Kenobi, one of the main Star Wars characters known by his signature catch phrase "Hello There" among fans.

State "Enhancement"

This was not requested, but we will describe it here as it can lead to a misunderstanding regarding the file system. When the client requests a STATE, the output on the standard output can be a bit hard to read, to make this easier for the user, when the peer receives a STATE request it will also assemble a webpage containing every detail regarding the peer's state. This HTML file will be stored under the output root directory with a SHA-256 like name.