Distributed Systems 2017/2018 - 2nd Semester
Project 1 – Distributed Backup Service

Diogo Almeida Cunha
up201405506@fc.up.pt

Miguel Sozinho Ramalho
up201403027@fe.up.pt

MIEIC, FEUP - APRIL 2018

1 Concurrency Design

In this section, the implementation of concurrency, considered fundamental for the current project, is described. Although we have completed the enhancements, this description was written for the Vanilla version (1.0) of the Service, as such, some remarks may not hold true after the enhancements.

In order to make achieve proper use of concurrency in the project, we started by extending the functionality of the java.net.MulticastSocket in a new class **MulticastSocketC** which also implements java.lang.Runnable. By doing so, we created a class that simultaneously represents a MulticastSocket and the corresponding Multicast Group, and also a class that can be executed asynchronously. Indeed, what the *@override void run()*{...} method does is to continuously listen for java.net.DatagramPackets on the corresponding Multicast Group. Whenever one of this packets arrives, it is parsed into a **Message** (see 1.1.1). Then a decision happens: if this message was one the current Peer sent, then it is discarded, otherwise a new task is created and added, wrapped in a **Dispatcher** (see 1.1.2), into the thread pool system as described bellow.

Each Peer, when started and after reading the command line arguments into an instance of **PeerConfig** (see 1.1.3), creates and executes three (3) threads - one per multicast channel it must subscribe (Control, Backup and Restore), in the form of a **MulticastGroupC**. These are left running and waiting for incoming packets until the process is killed. During this initialisation, a *java.util.concurrent.ExecutorService* (henceforth named ThreadPool) is created. This Threadpool will be responsible for handling, managing and executing the threads required for the tasks it is given, it works as a *fixed length window* for threads to run. One example of a task is precisely an instance of the **Dispatcher** class mentioned above, but the thread pool is used to manage the concurrency in the system at a higher level and can therefore be used for many ends, as it accessible through the **PeerConfig** object. As a side note to the concurrency, the internal state of the peer is also loaded from the non-volatile memory, from the file system. At this stage, the Remote Method Invocation (RMI) is also started and the stub for it is inserted into the RMI register.

The main thread, after the aforementioned initialisation process, does not stay idle, as an anonymous enhancement we designed it so that it performs some checks over the status of the internal storage of the peer, at every 10 seconds (other values could also be used). These checks currently include:

- Finding and removing empty folders from the internal storage (that could result from deleting chunks through the DELETE subprotocol or from space reclaiming/management operations);
- Looking for stored chunks that were marked as deleted but that were actually kept in the file system, due to some inability to delete them upon request and attempt to delete them from local storage.

This asynchronous tasks (to the rest of the service and subprotocols execution) can be seen as a high-level garbage collection system that can be expanded in the future for, for example, perform

some computations to better decide how to manage the current available memory i.e. which chunks to delete according to the current perceived and desired replication degrees.

The abstract use of concurrency is as described above. Nevertheless, some peculiarities of the implementation should be mentioned and explained in a more pragmatic manner.

Before the current implementation, a type of blocking queue was associated with each **MulticastSocketC** that would store the messages that did not need to trigger a task, but rather needed to be used for information by a task - an example would be the STORED message for file chunks that do not belong to the Peer that receives them, these would only need to be used by the task handling the incoming PUTCHUNK messages in the same Peer as this task needs to check if, after an increasing amount of time, enough STORED messages were already sent and, if so, abort the action of saving the chunk locally. This check was done inside the task that would process the PUTCHUNK message, by filtering and removing the STORED messages from this queue.

Although this worked fairly well, it increased the complexity off adding new messages to the Service and could also incur in concurrency problems, due to many tasks needing to iterate (even if through embedded methods) the same blocking queue - additionally it also would loose the received messages as this would be cleared from the queue, thus representing a potential limitation for protocols that needed the same information. Because of this, we replaced this mechanism by sending every message through the dispatcher that would, for the case of a STORED message, update the information about which Peers replied with a STORED to that same PUTCHUNK. This way, from the PUTCHUNK task, a simple read operation, from the hashmap that holds all the information for the chunks, would be required. It should also be mentioned that these hashmaps, accessible from the **PeerConfig**'s **InternalState**, are java.util.concurrent.ConcurrentHashMaps.

Figure 1 sums up the described concurrent architecture.

1.1 Relevant Classes (for the concurrency description)

The default package for the current project is *src*.

1.1.1 src.util.Message

A Java class that receives a DatagramPacket and extracts the header and body parts into an object form which is much easier to handle, query and update. It accommodates the values of:

```
<MessageType> <Version> <SenderId> <FileId> <ChunkNo>
<ReplicationDeg> <CRLF>
```

and can easily be extended or updated for new versions of the Protocol.

1.1.2 src.worker.Dispatcher

A Java class that implements a *java.lang.Runnable* interface and whose sole purpose is to determine what is the correct *src.worker.Protocol*'s child class, that represents a single message type handler, and run it in its own execution thread. Simultaneously, it holds all the information needed for any task handler to execute, namely pointers for the internal state of the Server and the message being processed.

1.1.3 src.main.PeerConfig

A Java class that parses the command line arguments into an object form which is much easier to handle, query and update. Simultaneously, it is responsible for holding a variable of type **InternalState**(1.1.4) for the current Peer.

1.1.4 src.localStorage.InternalState

A Java class responsible for holding, querying, saving and loading the data stored in non-volatile memory. It contains two *ConcurrentHashMaps*, one for storing information about the local chunks that other peers are backing up, and another for storing information about the chunks the Peer is

backing up for others. This class has the ability to serialise itself into a file and read itself again, corresponding to the a disk memory commit and retrieval, respectively.

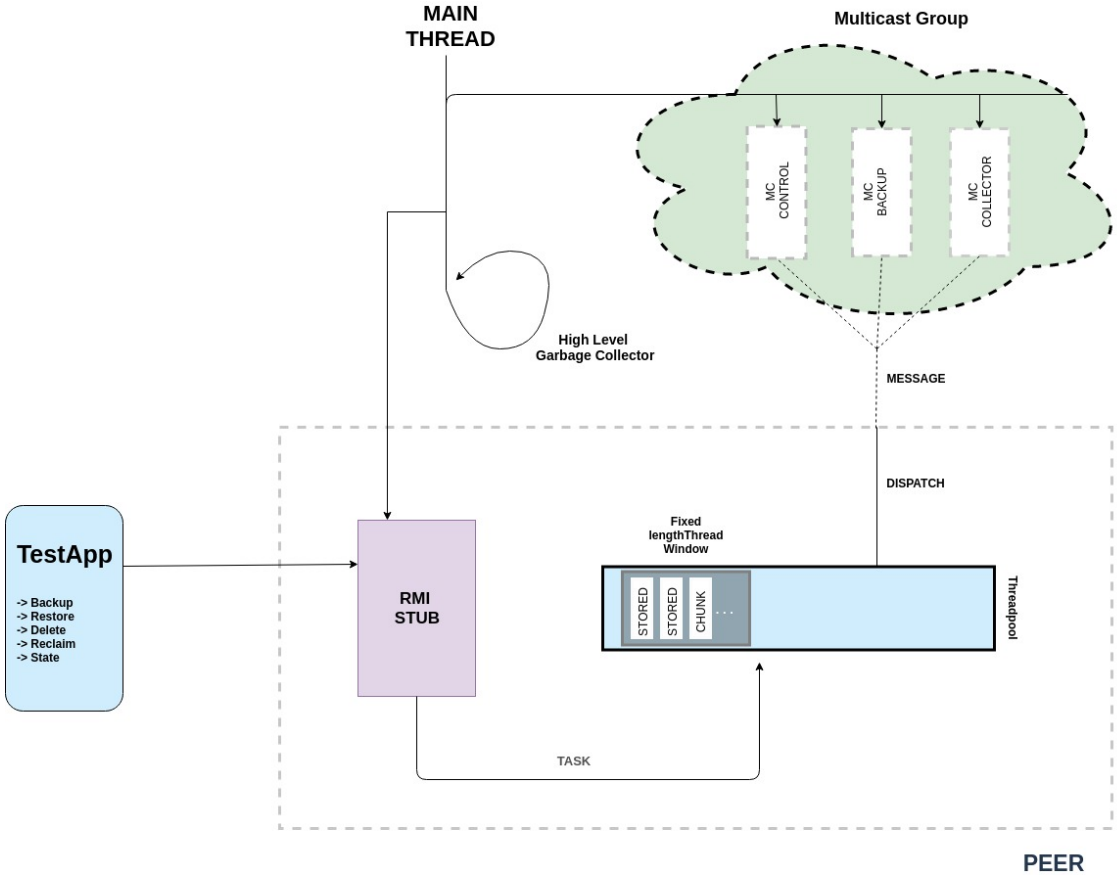

Figure 1: A Schematic visualisation of the concurrency implementation

2 Enhancements

The part of the code that contains the most important changes for the enhancements can be found by searching the project for strings matching **ENHANCEMENT_X** where **X** is the number of the enhancement in accordance with the following subsection numbering (1, 2 and 3), and also the project script.

2.1 Chunk backup subprotocol

Since the scheme for a Peer to manage insufficient available memory is expensive as it usually involves evicting chunks through the multicast group, what can we do about it (while ensuring the desired replication degree, avoiding these problems and, nevertheless, inter-operating with Peers that execute the base backup protocol)?

Our answer is as simple as it is efficient. Instead of a peer waiting for a random period of 0 to 400 ms, it shall wait from α to 400 ms, where α is a value calculated through equation 1:

$$\alpha = min(POC_i, 100) * 4 \tag{1}$$

Given POC_i as the percentage of occupied spaced in Peer i.

By doing so, we will ensure that the likelihood of a Peer, with little available space, saving a chunk is smaller than that of a Peer with a greater amount of available space. However, as this Peer with more available space accepts chunks, the likelihood of it accepting the next chunk grows closer to that of the first.

So as to continue permitting a Peer to detect others' STORED messages and stop its own chunk saving action, the highest value for α could be, say, 350 ms - for a peer with 100 or more percent this would be a random value from an interval $[350, 400]$ instead of $[400, 400]$. This change was not applied due to the fact that the behaviour upon a full memory can vary and both hypotheses remain valid.

The changes to the source code were made in the **src.worker.Protocol** abstract class.

2.2 Chunk restore subprotocol

This enhancement was designed to decrease the load impact of the original protocol on the restore multicast group. Whenever a CHUNK reply was sent, every Peer would be receiving the whole content of that chunk, in the message's body, when only a single Peer would be asking for it. To solve this, while keeping the versions interoperable and using the Transport Communication Protocol (TCP), we devised the changes to the subprotocol described bellow.

Whenever a Peer needs a chunk, the procedure remains the same: Send a GETCHUNK message on the control multicast group. However, the body of this message is no longer empty, but rather contains the IP address and Port number in the format *ip:port* of the server socket in the initiator Peer. Even though this changes the subprotocol, the Peers implementing the old protocol will still be able to process and handle this message in the same way as before, the only difference being a non-empty body, but since this extra information is discarded by default, no changes in behaviour will come from it.

Next, the first Peer to respond with the corresponding CHUNK message to the restore multicast channel will, in light of this enhancement, send an empty body in that message with a version different from *1.0* (Peers using the old protocol will send a CHUNK message with version *1.0* and so the initiator Peer will know the chunk contents are in the body). This simultaneously prevents large amounts of unnecessary data going through the restore multicast channel and keeps interoperability with the Vanilla version. Why? Because this message has the same effect for the non-initiator peers implementing the previous version, which is to realise they need not send their copy of the chunk as some other Peer already did so. However, we devised the changes to the protocol so that the initiator Peer can collect chunks from non-initiator Peers that do not implement the new protocol, as well. When a non-initiator Peer sends a CHUNK message with the new version in the header, the initiator Peer tries to receive the chunk, not through a multicast channel, but through the TCP socket it first sent on the GETCHUNK message. Thus avoiding the clutter of large messages in the multicast channel.

If the TCP connection fails, the chunk is assumed to be null so as not to be mistaken for a last chunk of length 0. In case of TCP failure, the restore protocol will keep sending GETCHUNK requests (that can be handle via TCP or not, depending on the responding peer) and so there is a fail safe system. Optionally, we could force the program to only attempt a TCP connection once and fall back to the previous version if needed, but we don't believe this is a deterministic matter and so we did not do it. The classes updated to accommodate this change were: *src.worker.P_Chunk*, *src.worker.P_GetChunk*, *src.worker.RestoreChunk* and a schematic representation of the flow can be seen in Figure 2, where Peer 2 is the first to send a CHUNK with an empty body, Peer 3 then holds back after waking up, and the chunk transaction is carried out through the TCP channel.

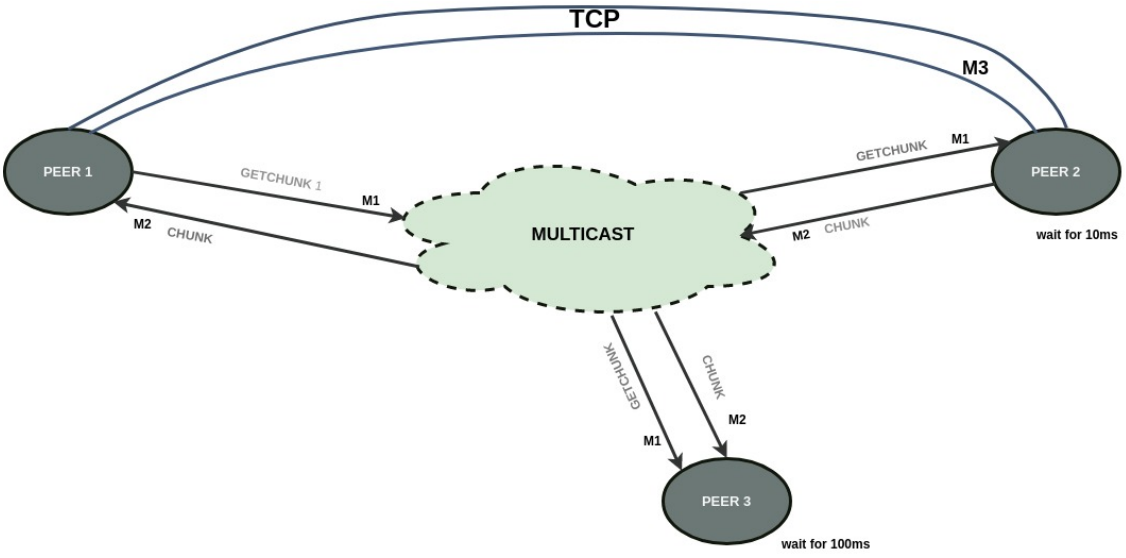

Figure 2: Visualisation of the restore subprotocol enhancement with TCP

2.3 File deletion subprotocol

If a Peer that backs up some chunks of the 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. Our solution for this problem consists of two changes to the base protocol.

The first is: each Peer shall reply to the DELETE messages (with a DELETED message), so that the initiator Peer can know which Peers that had previously replied with a STORED message (meaning they possess a local copy of this message) have actually deleted it, as intended. The initiator peer will update its internal state to know which Peers have not answered with a DELETED message.

The second change is to make the first action of every Peer be, after its initialisation, sending a message through the control multicast group simply saying HELLO (for sanity preservation we decided to be derisive and name this message ADELE instead of HELLO, hoping no one implementing our protocol enhancement dislikes the singer) thus identifying itself as having just woken up. The Peers in the network shall then look for the information on the deleted files and, if the new Peer is marked as one that did not reply with a DELETED message to the previous DELETE messages, than those Peers that identify this will resend the DELETE message. Once all the Peers that replied with a STORED message, have confirmed the deletion of the chunks of that file, the internal state is updated and no more DELETE messages will be sent for that file.

This way, without adding a great load to the network (as Peers are not expected to intermittently go on and off), we achieve a way of managing the removal of unneeded chunks.

As a matter of fact, this ADELE message can be useful for future improvements of the protocol, since it provides a way for the network as a whole to have an idea of when Peers enter the network, and a general idea of its size.

The changes for this enhancement were made in the form of two new classes that extend **src.worker.Protocol**, namely **src.worker.P_DELETED** and **src.worker.P_ADELE**

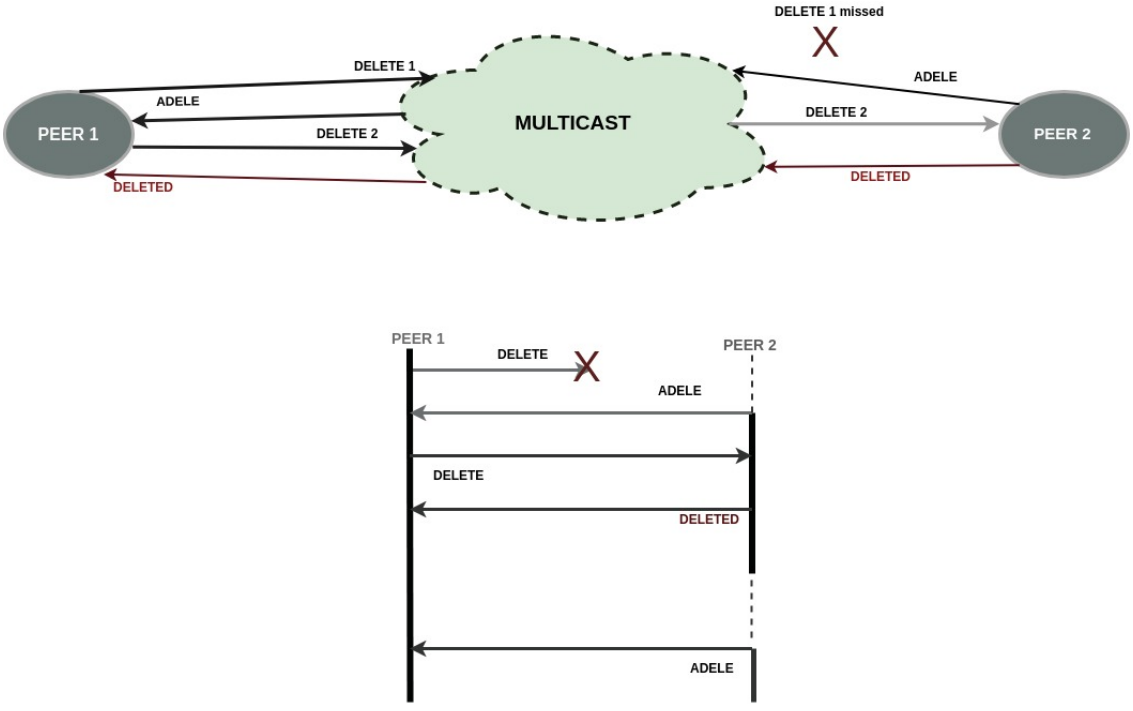

Figure 3: Visualisation of the deletion subprotocol enhancement

3 Suggestions for Service Enhancement

Some ideas that were not implemented, but that stood out in our conscience as interesting to implement include:

- Create a good internal memory management system that would take into consideration relative ratios of desired vs real replication degree and other inputs to better decide which chunks to keep and how to provide a sustainable limited memory backup service;
- Implement a fail safe that would make the chunk distribution (or at least file distribution), of a Peer's local files, be as widespread amongst different Peers as possible so that, in the event of server crashes or failures, no peer is, in average, more affected than others.

Besides these, many other ideas came up during the implementation of the distributed system in this project, as this seems to still be, metaphorically, a child that just started walking and not Francis Obikwelu himself.