Project 1 - Distributed Backup Service

Concurrency scheme:

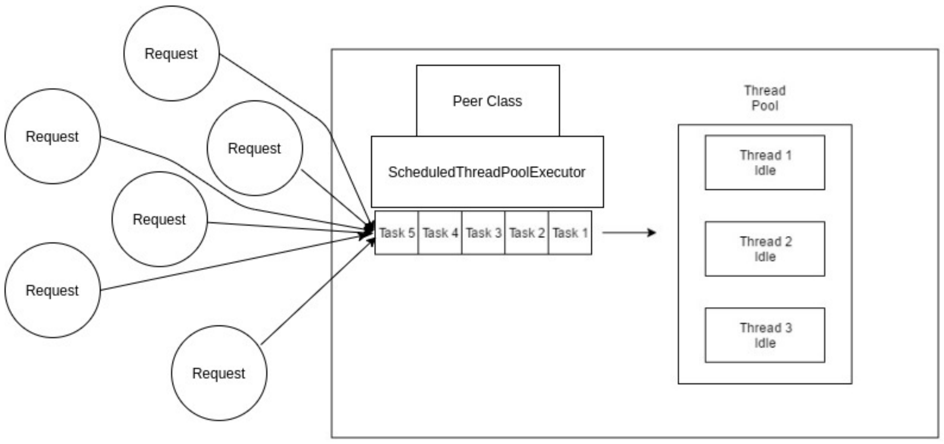
In our solution we followed the third option presented in the document “Hints for Concurrency in Project 1”, **One Thread per Multicast Channel, Many Protocol Instances at a Time.**

This option grants the maximum level of concurrency from the three options that were available to choose. This fact was the reason behind the adoption of this particular strategy in our project.

The concurrency was achieved making use of three major elements of the Java concurrency package:

- **Runnable Workers**: The task of receiving and dispatching the multicast requests is done using a worker that implements the runnable interface of Java. There is one worker per multicast channel running in separate threads. Using the OOP mechanism of dependency injection, we inject in these workers a proper strategy to handle and respond in the correct way. Furthermore, since they implement runnable, they are being executed parallelly in different threads. This way we achieve a scenario where we are allowed to deal with many protocol instances at the same time.

- **Thread Pool:** Some of the subprotocols of this project require waiting a random time, before sending a certain request. Without the introduction of concurrency in our project the side effect of this requirement would be completely disastrous, demanding us to use sleep in a single thread of execution. In our concurrency scheme, the easiest way to handle it in a simple and “clean” way was making use of Thread Pool Java mechanism, more specifically the **ScheduledThreadPoolExecutor**



Peer Class is the starting point of any request.
At any time in the project there is a reference to it.
This makes the access to the thread pool always available

- **Synchronized Blocks:** Operations regarding I/O in a multithreaded scenario required us to introduce synchronized blocks to ensure data integrity. These functionalities of Java use monitors to make sure that a single thread runs that block of code at a given time, avoiding race conditions. This decision was taken since we didn’t find any solution completely thread safe to deal with this segment of the project.

Enhancements:

In addition to the **1.0** vanilla solution we introduced enhanced solutions to all the suggested additional tasks of this project.
This solution composes the version **1.1.** This new version was built with a philosophy of **appending as much as possible, changing the few as possible**, in order to reduce compatibility issues with the use of the **1.0** version.

The vanilla version itself has different problems that affect the performance in different stages, but they all have a common ground. It induces too much activity in the nodes, activity that could be avoided much of the time, if less brute force solutions were followed and the requirement of using multicast for moving huge amounts of data could be relaxed.

Backup:

The problem the backup sub protocol shows in the vanilla version is a fast depletion of the disk space in the non initiator peers. The dummy version demands their peers a philosophy of storing as much as possible in the largest possible place This has a good consequence, that is we always get the best replication level for each chunk.

Although in practice the effects are not that convenient since the requirement is to achieve a prefixed level of replication, not maximizate it.

The solution found to handle this problem is divided in three steps:

-Before processing a PUTCHUNK request, now, the peer receiving the request, listens to the **environment for a random time between 0 and 2 seconds**. This fact gives the peer time enough to update his awareness of the replication level in the environment.

-After listening to the environment the peer just stores the chunk if the replication level it perceives for that given chunk is below the desired level + a WELL_DEFINED_THRESHOLD constant. **With this precondition we reduce the replication above the desired level** and the number of stores flowing in the network.

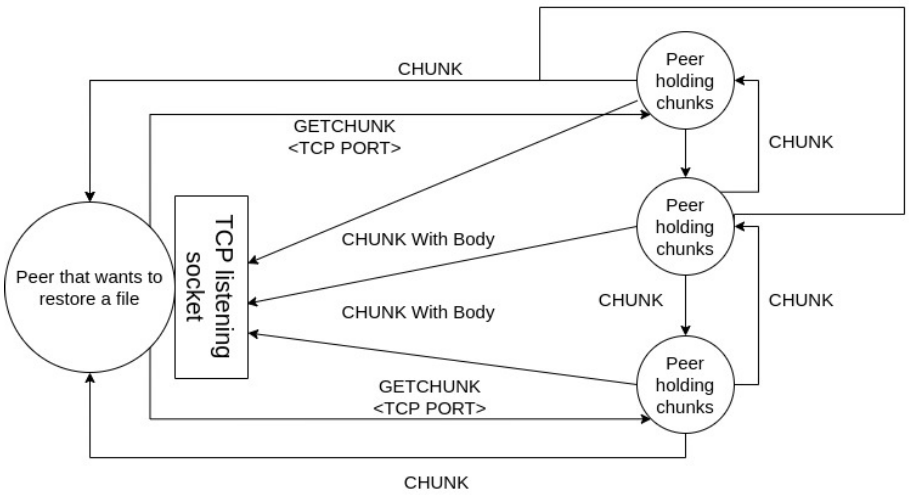
-The two methodologies presented above only reduce the depletion of space in the disk. There is another problem that the vanilla version of the protocol doesn't take into account. **We can easily run in a scenario where it is not possible to store no more chunks, because all peers run out of free space.** In this case, flooding the multicast channel with data packets that will be discarded is not a good design. Because of that we introduced a new message, the HEARTBEAT, that publishes the amount of free space periodically, this allows the introduction of a new terminology the "alive/dead peer", the ones that have/haven't, respectably, free space available. **This way before each backup we could only test if the number of alive peers are enough to satisfact our needs for a given backup request.**

```
<Version> HEARTBEAT <SenderId> <CRLF>
<AVAIABLE_DISK_SPACE><CRLF><CRLF>
```

Restore:

The restore subprotocol's biggest problem in the vanilla solution is the amount of information the CHUNK reply sends in a multicast way. Furthermore, many of these replies get completely discarded since they are related to the same chunk and only are intended to be received by a single peer.

In order to avoid sending big amounts of duplicate information in a multicast channel, we should use a one to many approach, rather than a many to many one.
In order to do this, we need to use TCP. Furthermore we needed to edit the GETCHUNK message to include information to allow the use of this protocol communication.



This schematic shows the flow of the new enhanced restore version.

Now the GETCHUNK message has an additional line that publishes the TCP port the initiator peer wants to receive the chunks.

```
<Version> GETCHUNK <SenderId> <FileId> <ChunkNo><CRLF>
<TCP_PORT><CRLF><CRLF>
```

Now there are two types of CHUNK messages:

-The one shared in the multicast channel, that don't have body, and they just serve the purpose of coordinating who is sending what in order to keep the mechanism that comes from the vanilla version of aborting the sending of a chunk if someone else do it first, making for that use of a random delay before testing that scenario.

-The same Chunk structure as in the vanilla version, with body to reply using the TCP connection.

Delete:

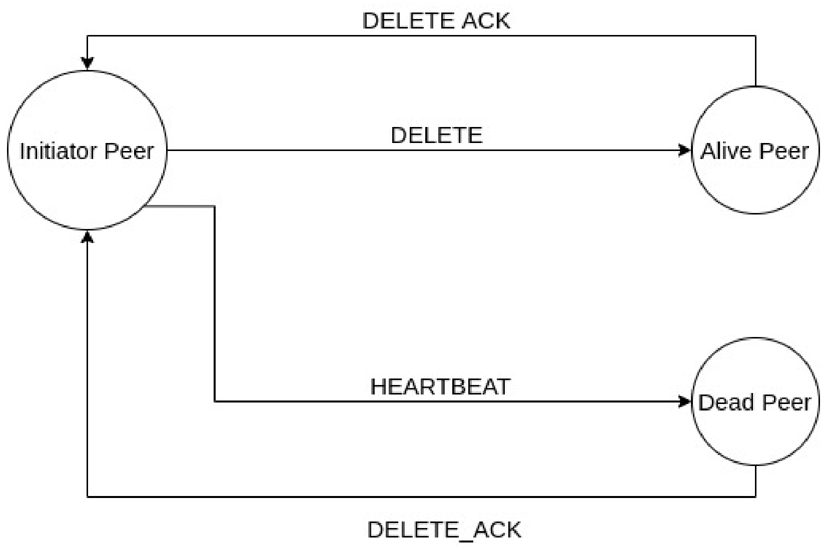
The biggest problem with the delete sub protocol in the vanilla version is that it doesn't introduce a reply to a DELETE request. The only approach we could use until now to ensure the request arrived at the destination was to repeat the same delete request a fixed number of times.

This was just a hotfix rather than a real solution for a bigger problem. If a peer is offline at a given time it won't process the delete request.
The consequence of this scenario is that some chunks become untracked and are simply wasting space in the peer. Phantom chunks that would never be reclaimed.

To fix this problem we were required to introduce two new elements:

-The new DELETE_ACK message that is just a confirmation reply that the DELETE message was received by the destination. These ACK replies are stored as they were metadata, this way we ensure that this record is persistent between executions.

```
<Version> DELETE_ACK <SenderId> <FileId> <CRLF><CRLF>
```



-Then making use of the HEARTBEAT message already described in the backup enhancement section of this document, we are able to check if a peer that misses a DELETE_ACK heartbeats, if it does, we try again the sending of the DELETE request. **This pooling mechanism only stops when all alive peers acknowledge all their deletes.** Furthermore it doesn't send requests when the faulty peer doesn't give proof of life, reducing the flooding of useless requests as possible.

FEUP MIEIC
SDIS 2020/2021
Class 06 Group 13
Klara Banić up202010658@edu.fe.up.pt
Rúben Almeida up201704618@fe.up.pt