Distributed Backup Service
Distributed Systems

Gonçalo Alves[up201806451] and Pedro Seixas[up201806227]

Faculdade de Engenharia da Universidade do Porto
feup@fe.up.pt
https://sigarra.up.pt/feup/pt/web_page.Inicial

1 Concurrency

Our implementation of the Distributed Backup Service processes different messages received on the same channel, at the same time, **without resorting to sleep functions**. We have also eliminated all blocking, through the use of **AsyncronousFileChannel**. As a common basis, each protocol implemented resorts to two **Thread Pools**, one for sending messages and the other for receiving messages.

When a peer initializes, it stars three main threads, one for each Multicast Channel (MC, MDR, MDB), **MulticastListener**, that will be waiting to get messages. As soon as a message arrives, the peer creates a Thread, **MessageReceiver**, which will parse and create the specific message type handler to handle that message.

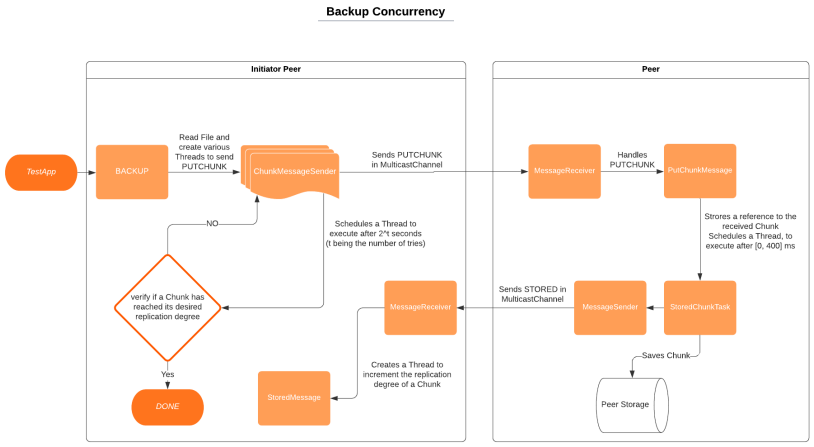
Below, we will detail more accurately how we dealt with concurrency in each protocol.

1.1 Backup Concurrency

This protocol starts with the reading of the file that the peer initiator wants to backup. When we read a chunk of the file, we request a thread from our thread pool of sending messages. This thread, **ChunkMessageSender**, will be responsible for sending the **PUTCHUNK** message to the **MDB** and for scheduling another thread. After a fixed interval, this thread either terminates or creates a new **ChunkMessageSender** with double the timeout, depending if the **Chunk** has already reached its desired replication degree.

To avoid having almost the full file content in volatile memory, a semaphore is used alongside the thread pool. Although the thread pool has a maximum size, this size only applies to the active threads, not limiting the number of threads in the waiting queue. Using a semaphore with a capacity, $2C$, double the thread pool size, C, we are able to **reduce the volatile memory usage without decreasing the performance of the backup protocol**. This is achieved by reading $2C$ chunks of the file and creating C chunk sending threads, being left with C chunks in the queue. When the first C chunks are sent and the threads are finished, the execution doesn't need to wait, as it has another C chunks to be sent in the queue. With the semaphore now unlocked, the time spent sending the chunks that were in the queue is used to read another C chunks of the file. With this implementation, the execution does not stop and the amount of memory used is significantly lower.

When a peer receives a message, it will create a Thread to handle it, as mentioned in the introduction, in this case, an object of type **PutChunkMessage**. **PutChunkMessage** is responsible for saving a registry of the Chunk it has received and for scheduling another thread. After a random interval, this second thread, **StoredChunkTask**, saves the contents of the chunk in non-volatile memory and creates a new Thread, **MessageSender**, for sending the **STORED** message to the **MC**.

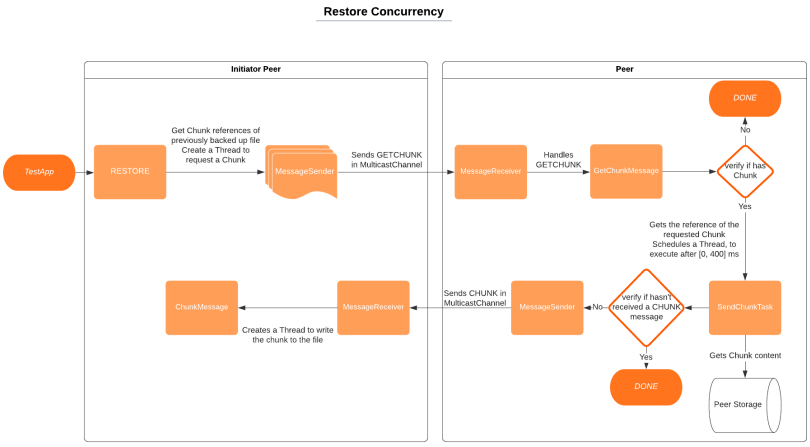


1.2 Restore Concurrency

This protocol starts with getting all the chunks of the previously backed-up file. For each chunk we get, we request a thread from our thread pool of sending messages. This thread, **MessageSender** will be responsible for sending the **GETCHUNK** message to the **MC**.

When a peer receives a message, it will create a Thread to handle it, as mentioned in the introduction, in this case, an object of type **GetChunkMessage**. **GetChunkMessage** checks if the peer has the requested chunk and schedules a second thread. After a random interval, this second thread, **SendChunkTask**, checks if the peer has received a **CHUNK** message, ending this process if it has. If not, it loads the chunk from non-volatile memory and creates a new Thread, **MessageSender**, for sending the **CHUNK** message to the **MDR**.

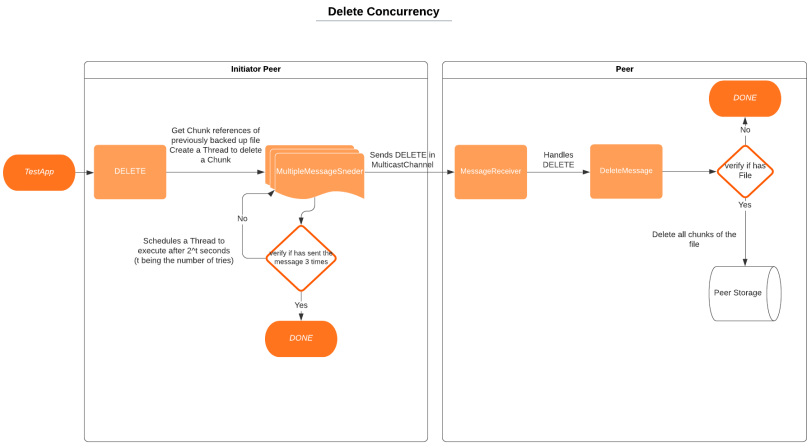
When receiving this **CHUNK** message, the initiator peer then uses an **AsyncronousFileChannel** to save the chunk in the position it belongs. With this, we assure that the execution doesn't stop by saving each peer in its position as soon as they arrive and not having to wait for them to arrive in the correct order. This also helped us to remove all blocking calls that were left.



1.3 Delete Concurrency

This protocol starts with getting the file identifier of the file which the initiator peer wants to delete and we request a thread from our thread pool of sending messages. This thread, **MultipleMessageSender**, is responsible for sending the **DELETE** message to the **MC** and for scheduling another thread of the same type. After a fixed interval, this second thread either terminates or creates a new **MultipleMessageSender**, depending on if it has sent the message with the desired tries.

When a peer receives a message, it will create a Thread to handle it, as mentioned in the introduction, in this case, an object of type **DeleteMessage**.. **DeleteMessage** is responsible for deleting all the chunks of the requested file, in non-volatile memory, and delete all registries of that file and its chunks.



2 Enhancements

2.1 Backup Enhancement

"This scheme can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full. Can you think of an alternative scheme that ensures the desired replication degree, avoids these problems, and, nevertheless, can interoperate with peers that execute the chunk backup protocol described above?"

The original backup protocol had all the peers counting the number of **STORED** messages they received. For the initiator peer, this allowed him to check if a chunk had reached the desired **replication degree**. For the other peers, this count served to update the local registry of the chunk. With this implementation, every peer would store a chunk, after receiving a **PUTCHUNK**, and respond to the initiator peer, after a time interval, leading to a quick depletion of storage space and an unnecessary high replication degree.

To solve this problem, we took advantage of the time a peer has to wait before sending a **STORED** message. This small interval, between 0 and 400 ms, is used to reduce the collisions of messages in the channels and during this interval where a peer is "dormant", it receives several messages, such as **STORED**. Therefore, a peer is constantly updating a chunk's replication degree, before responding to the initiator peer.

Because of this, we did two things to enhance the original protocol. The first one was to not immediately store the chunk in memory. Instead, we only kept a reference to the chunk, so that we could update its replication degree as **STORED** messages came. The second change was to verify if the chunk we wanted to store before the peer was "dormant", had already reached its desired replication degree. If this was the case, we would simply delete the reference we had previously created and end the processing of the **PUTCHUNK** message there, without saving the chunk, as it already had the desired replication degree.

This way, the desired replication degree doesn't grow substantially past that of the necessary and we make sure we aren't depleting the storage of every peer.

In terms of interoperability, this enhancement does not change in any way the way the peers function. The only difference is that non-enhanced peers will always store a chunk, even if it already has reached or surpassed the necessary replication degree.

2.2 Restore Enhancement

"If the files are large, this protocol may not be desirable: only one peer needs to receive the chunks, but we are using a multicast channel for sending all the file's chunks. Can you think of a change to the protocol that would eliminate this problem, and yet interoperate with non-initiator peers that implement the protocol described in this section? Your enhancement must use TCP to get full credit."

In the original restore protocol, the initiator peer sends a **GETCHUNK** message to request for a chunk. If a peer has that specific chunk and no other peer already sent that chunk, it will send a **CHUNK** message with the content of the chunk in the message body. This causes every other non-initiator peer to get a message with the body of that chunk even though only the initiator peer needs it, which causes a lot of unnecessary traffic for the other peers.

Our approach to solve this problem was to establish a **TCP** connection between the initiator peer and the peer that has the chunk, so it could be the only one to receive the content of the chunk.

To achieve this, the body of the **CHUNK** message was modified to carry the **IP** address and the **port** number, instead of the chunk content. Before sending the **CHUNK** message, the peer opens a TCP socket and then waits for the initiator peer to establish the connection. If, for any chance, the initiator peer doesn't connect to the TCP socket in 5 seconds, the peer will close the connection and assume that the initiator peer already received the chunk from another peer.

In terms of interoperability, the initiator peer will always know if the content of the **CHUNK** message is either the chunk content or the TCP connection information when parsing the message header, as the version number in the message header is different. The other peers that receive the message only need its header, so they won't have any issues with this new implementation.

With this enhancement, the initiator peer is the only one to receive the full content of the chunk file, significantly decreasing the unnecessary traffic in the other peers.

2.3 Delete Enhancement

"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. Can you think of a change to the protocol, possibly including additional messages, that would allow to reclaim storage space even in that event?"

With the original implementation of the delete protocol, if a peer is down at the moment that another peer sends the **DELETE** message, it will not receive the message and therefore it will not delete the chunks that are part of that file, keeping them forever, occupying unnecessary storage.

To solve this we need to make sure that when a peer reconnects to the network after a crash, it can find out which files were deleted while it was down. To achieve this, we implemented a new message, **CONNECTED**, which is sent every time a peer is initiated. This message's body has the **IP** and **port** of the TCP server socket, opened by the peer on startup. This socket has a timeout of 1 second, so that it doesn't wait for connections indefinitely. Upon receiving this message, a peer checks if it has any deleted files and, if so, sends a new **DELETE** message for every one of those files using the TCP details received in the message body, to make sure that the new peer deletes every file that isn't needed anymore.

In terms of interoperability, this new message won't interfere with the non-enhanced peers, as it will simply be ignored because they don't know how to parse this kind of message.

With this solution, it is guaranteed, assuming that every **DELETE** message is received by the peer that was down, that the peer will delete the chunks that correspond to files already deleted from the network.