Distributed Backup Service

1. Concurrency

To achieve maximum operability our implementation of the protocol makes use of the multiple cores of a system so that multiple operations may be executed at the same time.

A representation of the various entities of the implementation is below. Furthermore it illustrates how each of these communicate with each other.

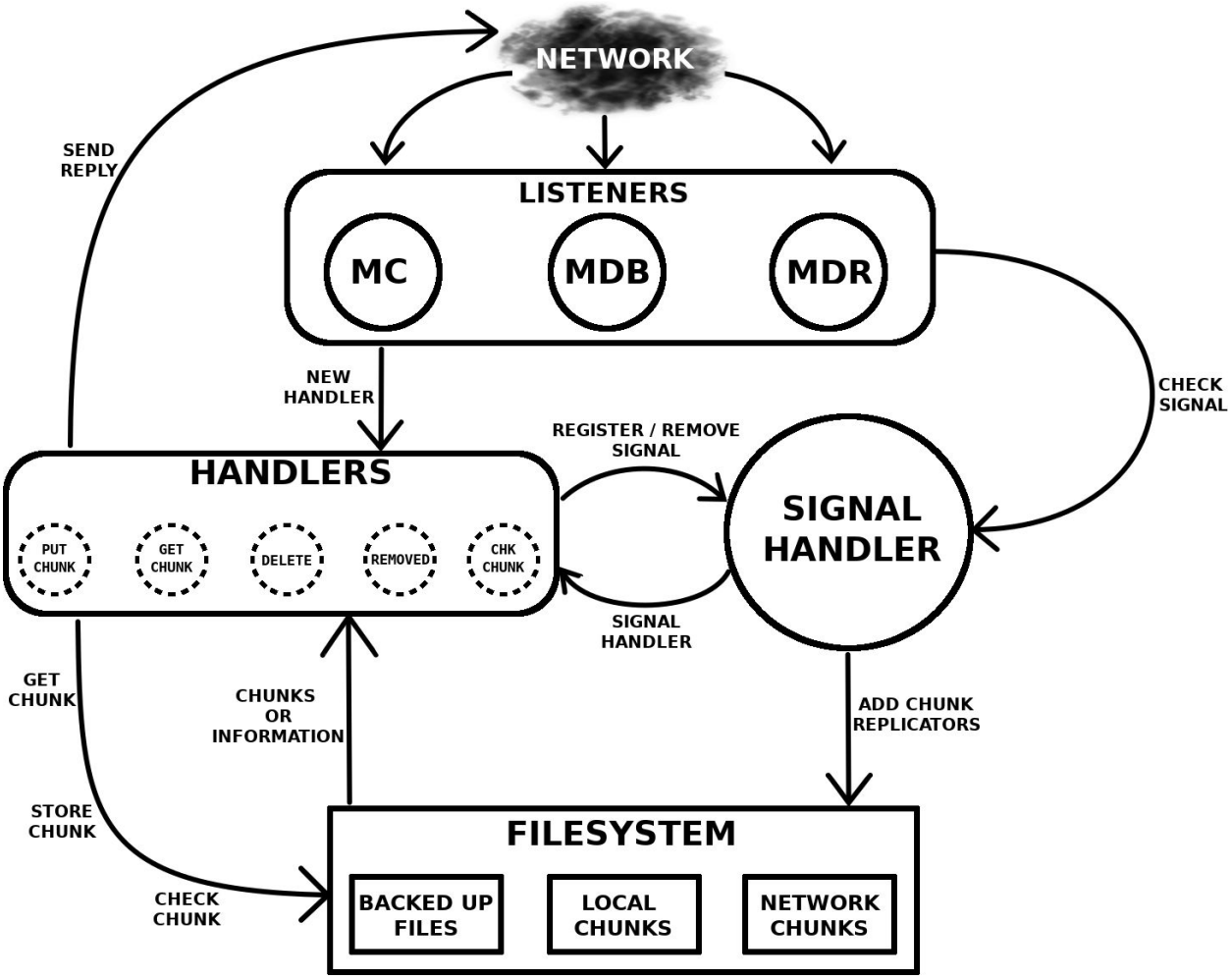

Fig. 1 - Representation of the concurrency and communication between modules

In the representation the rounded rectangles represent a group of entities, namely the *Listeners* which are used to listen to the network communications, and the *Handlers* which process and handle messages when required.

The *Filesystem* represents the underlying file-system of the peer. It stores 3 types of information, the files that were sent for backup to the network, the files that were received from the network to be stored in the peer, and the files which are not stored locally but where sent into the network to be stored.

Circles with full-border represent a permanent thread, so the program has 4 constantly running threads, *MClistener*, *MDBlistener*, *MDRlistener* and *SignalHandler*. Circles with dashed border represent threads which are launched on demand, such threads are used to handle the received messages from the network.

1.1 - Listeners

The *Listeners* sole purpose is to listen to the communications between the peers on the network. They do not process the messages received, they simply send them to *SignalHandler* and then if needed create a new *Handler* to process packets.

The three *Listeners* share a single *ThreadPoolExecutor* which is used to allow multiple *Handlers* to run at the same time. The maximum number of tasks is 255.

1.2 - Handlers

The *Handlers* is a group of threads which are only executed on demand. So if there is a message from the network that elicits some kind of response a new *Handler* is started to process that message and possibly respond to it.

Some *Handlers* require the knowledge of whether some kind of message has arrived or not, as is the case with *PUTCHUNK*. In these cases the *Handler* needs to know whether that message has been received or not. For those cases the *Handlers* register a type of packet, file ID and chunk number to the *Signal Handler*, which in turn stores the information that the *Handler* wants to be notified if a message matching the requirements is received. Similarly, after the *Handler* has registered the message for signal, it can also remove it from signal.

In the case that the *Handler* needs to have access to the *Filesystem* it can do so without worrying about the synchronization of the operations since the *Filesystem* module takes care of it.

1.3 - Signal Handler

The *Signal Handler* plays a major role in most of the protocols of the network. Its implementation is quite simple, it has a *queue* of received messages, and is constantly taking messages from that queue and processing them. *Listeners* put messages received in that queue to be processed and then proceed to keep listening.

Furthermore *Signal Handler* has a 'registry' of what *Handlers* want to be notified when a certain message matching a pattern is received. The registry stores the type of message such as *PUTCHUNK or STORED*, then it stores the file ID and the chunk number of these messages. These 3 arguments point to a single *Handler* that wants to be notified when these messages are received. There can only be 1 *Handler* at a time that is registered to receive a signal for a given message, though that is enough for the protocol in question. This registry is implemented in the following data structure:

```
ConcurrentHashMap<String, ConcurrentHashMap<String, Handler> > signals;
```

The first key of the *HashMap* is the type of the message. The key of the inner *HashMap* is a combination of file ID and chunk_number, which we refer as *chunk_id*, it is composed of <fileID>#<chunk_number>. The value of the inner *HashMap* is the *Handler* that registered to be signalled of a message matching these elements. Should a message match a registered message pattern, then the *Handler* is signalled by the following function that all *Handlers* must implement:

```
public abstract void signal(PacketInfo packet);
```

Through this the *Handler* can be executing other instructions and at the same time be signalled that a certain message was received since this function is called by *Signal Handler* thread and not a *Handler* thread.

Another important aspect of *Signal Handler* is that it keeps track of *STORED* and *PUTCHUNK* messages in order to keep *Filesystem.NetworkChunks* and *Filesystem.LocalChunks* up to date this is important to be able to achieve maximum reliability in a chunk replication degree and to keep track of which peers stored a given chunk. This also allows a peer to store information about chunks it does not store, which is useful for the implementation of the Delete enhancement.

As a final note, we decided to separate the *Listeners* and *Signal Handler* because *Signal Handler* might have to do some operations which are time consuming and so in order to maximize the time that a *Listener* is actively listening to the network we separated these in 2 threads.

1.4 - File System

Although this module does not represent any running thread it is important for the concurrency of the project because of the multitude of operations that can be done on the *File System*. *Handlers* communicate with *File System* through the class *files.FileHandler* which is a

static class able to be accessed from anywhere, so multiple threads can execute the same function at the same time. Synchronization is guaranteed by *files.FileHandler* so *Handlers* do not need to worry.

When restoring a file *File System* launches multiple threads to write to a file, using *AsynchronousFileChannel* which minimizes the time a thread is stopped in I/O operations. All the other I/O operations are blocking, since they mainly consist of either writing or reading a single file, this means that the thread cannot continue operating before it read or written the file, so it will have to wait for the I/O to complete.

Although the *File System* distinguishes a *Local Chunk* from a *Network Chunk*, the remaining modules do not, so for example, when a peer is storing a chunk because it got a *PUTCHUNK* message, it need not keep track of how many peers stored the chunks, it simply requests that information to the *File System*. That information might already be in a *Network Chunk* that keeps track of which peers replicated that chunk. Should no *Network Chunk* exist in the *File System* that matches the chunk *PUTCHUNK* wants to store, then it means that the peer is the first replicator of that chunk and *File System* creates a *Local Chunk* when the chunk is stored, and further *STORED* messages received for that chunk will add a replicator ID to that *Local Chunk.*

If there was a *Network Chunk* that matched the chunk *PUTCHUNK* wants to store and the actual replication degree is smaller than the desired, then *File System* will store the chunk in both disk-memory and metadata, and convert the *Network Chunk* to a *Local Chunk*. All posterior signals from *Signal Handler* to *File System*, such as a *STORED* message to that chunk, will add a new replicator ID to the newly created *Local Chunk*.

2 - Enhancements

Our implementation has 2 enhancements, the Backup enhancement and a Delete enhancement.

2.1 - Backup

As explained above, *File System* keeps track of which and how many peers stored every packet exchanged in the network, so before the *PUTCHUNK Handler* stores the chunk received (after waiting for a random interval), it checks what is the actual replication degree number, if that number is smaller than the desired replication degree then it stores the chunk in the *File System* and proceeds to send a *STORED* message. Further *PUTCHUNK* messages for the same chunk also elicit the *STORED* response, should the message be lost.

2.2 - Delete

To achieve this enhancement 2 messages where added that allow a peer to know if a given chunk is still being stored in the network. The messages are the following:

CHKCHUNK <version> <senderID> <fileID> <chunk_number> CRLFCRLF

CHUNKCHKS <version> <senderID> <fileID> <chunk_number> <replicators_n> CRLF
<replicator1> <replicator2> <replicator3> ... CRLFCRLF

The first message *CHKCHUNK*, meaning Check Chunk, is the message the initiator peer sends to the network to check if a given chunk is still being stored in the network. A peer which has information about the chunk, either in a *Local Chunk* or *Network Chunk*, sends the reply *CHUNKCHKS*, meaning Chunk Checks, that utilizes the original messages replication degree field to instead send the number of replicators of the chunk, and in the next line of the header sends the ID of the peer that replicated the chunks. The number of ID's sent is equal to *replicators_n* and each replicator ID is separated by at least one space character.

Such messages allow the initiator peer to update its local information regarding the replication status of the requested chunk, so that every peer in the network has up to date information. These messages also allow a further enhancement explained below.

3 - Further Enhancements

The base enhancements which were recommended by the teacher that we implement were explained in section 2. However we thought there was a need for some further enhancements to the original protocol that we implemented and explain below.

3.1 - Restore Concurrency and Resend

In the original restore protocol, the initiator peer only sends *GETCHUNK* messages once for every chunk. What we observed through testing was that for files that are split into many chunks (>100 chunks) more often than not, the non-initiator peers did not receive some *GETCHUNK* messages, this is due to UDP not being reliable, so some messages might be lost. To fight this, the initiator-peer of the restore sub-protocol, keeps track of what chunks were received and what chunks are missing. Similarly to the backup sub-protocol, if after a given delay not all chunks were received, the initiator-peer resends the *GETCHUNK* message of only the files that are missing. This action is repeated at most 5 times before the initiator-peer stops the restore protocol and fails the restore of the selected file.

Furthermore, instead of the initiator peer sending a *GETCHUNK* for a given chunk and only after getting a *CHUNK* does it send the next *GETCHUNK*. We decided that the peer would send all *GETCHUNK* messages at once, scheduling a waiter thread to check if all the chunks were received, if not then it resends the corresponding *GETCHUNK* messages. Meanwhile the *SignalHandler* would signal the *Restore* thread of received *CHUNK* messages, and creates a new *Future* that will write the contents of the received chunk to the file system using *AsynchronousFileChannel*.

3.2 - Reusing previously stored chunks

Although in the protocol specifications it was stated that we can assume that ‘*any files with metadata stored on a server are never lost on a server's crash*’, after implementing the Delete enhancement we realized that the enhancement could easily be applied to reusing the peer previously stored files on disk, in case of a server crash. So if requested, a non-initiator peer when starting checks if there is remnants of a previous execution stored on the disk. If there is then it initiates the Delete enhancement, sending *CHKCHUNK* for every chunk it finds in its designated folder in disk. Peers that did not crash will reply with *CHUNKCHKS* which will enable the newly started peer to quickly stay up to date with the network current status, and delete any unnecessary chunk.