Distributed Backup Service

João Diogo Martins Romão[1] and Tiago Ferreia Alves[2]

SDIS, 3MIEIC06, G08, Distributed Backup Service

1 Enhancements

1.1 Backup Enhancement

The initial version of the backup protocol was not very efficient since when a peer backed up a file, all the other peers would save it, independently of the replication degree. In order to solve this problem, we made a simple modification where instead of the executing the algorithm1, where the program stores the chunk, waits a random time and then send the stored messaged, we executed the Algorithm 2.

Algorithm 1 Backing Up File 1.0

```
1: function SaveChunk
2:     saveChunkFile()
3:     waitRandomDelay()
4:     sendStoredMessages()
```

In the Enhanced algorithm, the backup waits the random delay first, during which it saves the stored messages sent by the other peers, of that specific chunk. After the delay, it verifies if the number of stored messages received, which correspond to it's perceived replication degree of that chunk, is greater or equal to the desired replication degree, preventing it from backing up the file if so. The logic can be seen in the following algorithm

Algorithm 2 Backing Up File with Enhancement

```
1: function Save Chunk
2:     waitRandomDelay()
3:     numStoredMsg ← receiveStoredMsgDuringDelay()
4:     if numStoredMsg < chunk.getReplicationDegree() then
5:         saveChunkfile()
6:         sendStoredMessages()
```

This solution was found to be effective, since the random delay enables some peers with smaller delays to save the chunk and send the stored messages before other intervals have finished.

Whilst the algorithm is not perfect, since some files are still backed up more times than needed, we also wanted to implement a new type of message, that would delete the files that were not supposed to save the file, but did due to the delay of the UDP connection. On the other hand, because of the extra processing needed and given the fact that the enhancement considerably improved the project performance, we decided not to implement this feature.

In addition to this, we also verified that if two peers tried to backup the same file at the same time, this could bring some problems since they have the same id, but given the fact that we should assume that peers run in different machines, this problem should be solved since the Id has in consideration the file path, the modified date and the length, which is should be different in every computer.

1.2 Restore Enhancement

The enhancement to the restore protocol had as main goal to reduce the amount of peers that receive the chunks of the file to be restored to the initiator peer (the peer who requested the restore protocol). It was suggested to to establish a TCP connection to build the protocol enhancement.

To build this protocol we created a new CHUNK message that replaced the initial one established by the protocol.

Original message

<Version> CHUNK <SenderId> <FileId> <ChunkNo>
<CRLF><CRLF><Body>

New version

<Version> CHUNK <SenderId> <FileId> <ChunkNo>
<CRLF><CRLF><TcpPortNumber>

Firstly, the enhanced subprotocol sends, in the same way that the original one does, a GETCHUNK message via the Multicast Control Channel. Peers which have chunks of the file being restored respond with a CHUNK message send via the multicast data recovery channel, after a back off time that allows a peer to collect other CHUNK messages and realize if this one was already sent, in order to avoid flooding the host with messages.

The main difference is that the enhanced subprotocol message does not send the byte content of the chunk to the multicast channel. Instead it sends the port number of the TCP server it started. This way the hosting peer can establish a TCP connection and receive the chunk needed to restore the file, being the only one who receives the chunk.

To sum up, the enhanced protocol still uses UDP, so that the hosting peer can retrieve the port number of the TCP connection. Every peer that stores a chunk of the file being restored, starts a temporary TCP server by calling *new ServerSocket(0)*, being the hosting peer the client that requests information.

1.3 Delete Enhancement

The following Enhancement had the objective to correct a common problem in the initial version, the fact that if a peer backs up a chunk, and the initiator peer starts a delete protocol when the peer is turned off, the peer will keep forever a chunk that is useless. In order to correct this flaw, we created two types of messages:

- <Version> DELETED <SenderId> <FileId> <CRLF><CRLF>
- <Version> STARTING <SenderId> <CRLF><CRLF>

In order to add this feature, when the initiator peer that has the original file starts a DELETE Protocol of the file, instead of assuming all the peers have deleted their chunk, waits for a confirmation message, the DELETED, which is only sent if the peer has received the DELETE message and erased the respective chunks. This way, the initiator peer can eliminate one by one, the previously stored information of the peers that have stored it's chunks, remaining only with the ones that did not receive the message due to inactivity for example. It was also created a flag isDeleted, that is changed to true when the initiator peer starts the protocol, so that in the future knows which files have been deleted but have pending chunks. When the inactive peer that has the chunk starts, it sends the STARTING message to the Control Channel, so that when the initiator peer receives it, can confirm if it has any pending deleted file chunk associated to that peer and start a new Delete protocol if it has.

2 Concurrency design

2.1 Implementation structure

Each peer starts up by registering creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name received as a command line parameter in a Java RMI registry. This allows the peer to communicate with the client, the developed TestApp, and execute the requested protocol.

The peer also creates three receiver threads, one for each of the multicast channels (MC, MDR, MDB), that handle received Datagram Packets trough multicast and allow the peer to receive and process the messages.

2.2 Concurrent data structures

As it was referred we implemented one receiver thread per multicast channel and since we have used RMI, the multicaster thread is the thread created by the RMI run-time to handle any protocol operation invoked by the TestApp.

In order to keep consistent information of each peer we needed thread-safe data structures. The information relative to a peer is kept in a Serializable class called Metadata, that keeps information relative to protocol executions hosted

by the peer and information relative to chunks the peer has saved. Most of this information is kept in several ConcurrentHashMap's since it allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all.

Another thread-safe data structure used was the ConcurrentSkipListSet. For example, to keep information of the STORED messages received we kept in the peer metadata a Set of Integers containing the ids of the peers who sent the STORED messages, this way the initiator peer can keep track of the count of STORED messages as well as which peer has a certain chunk. To make sure that this information and others that make use of this data structure are consistent we opted to use this concurrent data structure.

2.3 Processing of Different Messages Received on the Same Channel at the Same Time

To support the concurrent processing of different messages received on the same channel more than one thread per channel is needed.

In an initial implementation each channel only allowed processing one message at a time. In order to avoid this problem and process multiple messages on the same channel we would need more than one thread per channel.

To do this we used the *ThreadPoolExecutor*, avoiding the overhead that comes with creating and terminating threads for every received message.

2.4 Elimination of blocking calls

In some implemented protocols, it was necessary to wait between 0 and 400 ms before sending a message to the control channel, in order to prevent the flooding of the channel, or for example in the enhanced backup, to decide whether to store the chunk or not depending on the stored messages received during that time.

In the beginning, we used the thread.sleep() calls to generate those waiting times. However, this calls can lead to a large number of threads running at the same time, consequently limiting the scalability of the service. In order to reach a better scalability, instead of the thread.sleep() we used the SheduledPoolExecutor.schedule(), a non blocking behaviour that is run in another thread.

The last step was to remove all the blocking calls that would be accessed by multiple threads, such as in File reading and writing, being possible to reach an even higher scalability. This was possible by using the AsynchronousFileChannel Class, which allows concurrent access to files and spawns a thread in different ways:

- Callback functions are used to execute a task as soon as the file operation was finished
- The operation returned a Future object, that allowed to execute other operations before having to use the respective value, taking maximum advantage of the asynchronous capabilities.