SDIS 2018/2019 - 2nd Semester Project 1 -- Distributed Backup Service (T3G10)

Amadeu Pereira
201605646

Nuno Lopes
20160533

I. Introduction

This report's objective is to describe the specification of the enhancements made (to the Backup and Restore subprotocols) and how the concurrent execution of instances of the protocols was achieved.

II. Enhancements

Backup Enhancement

This enhancement aims to reduce the depletion of the backup space and the consequent increase in a peer activity once that space is full.

Using the vanilla version of the Backup protocol, the backup space is used rather rapidly because, after receiving a *PUTCHUNK* message, the peer proceeds to the immediate store of the chunk received, replying with a *STORED* message after a random delay uniformly distributed between 0 and 400 ms.

To battle the problems regarding the default implementation of the protocol we opted to make a slight, yet really simple, change to the protocol. Instead of starting the protocol rapidly storing the chunk, after a *PUTCHUNK* message, we wait random delay (between 0 and 400 ms, as described before). If in that period of time enough *STORED* messages were sent to satisfy the current chunk's desired replication degree the peer ignores it, otherwise, it continues normally saving the chunk a replying with a *STORED* message.

Note that this implementation doesn't add additional overhead to the service, since all threads are responsible to keep track of the current replication degree of all chunks they are co-responsibles for.

Restore Enhancement

The Restore protocol enhancement objective is to combat the fact that the default protocol sends large chunks of data through a multicast channel, when those chunks are targeted at the initiator peer only.

As the project specification explicitly states, this enhancement must use a TCP connection for a direct and safe connection, so we did it.

When the Restore protocol is started (in enhanced mode) the initiator peer opens a TCP Server to receive the chunk data directly. Although, a problem occurred, how would the other peers know where to send the messages… The IP was fairly easy to solve, since we could get it from the DatagramPackets sent from the initiator peer by calling getAddress. The biggest complication was the Port. Since we would like it to be able to run several instances of the Restore protocol we couldn't give it a default port to listen to, so we needed to start the server the following way:

```
new ServerSocket(0);
```

A port number of 0 means that the port number is automatically allocated, typically from an ephemeral port range. This port number can then be retrieved by calling getLocalPort (as stated in the Java Documentation [here]).

The best way we found was to create a new type of message. The message is *GETCHUNKENH* and is the same as the normal *GETCHUNK* but it adds one argument that is the Port where the TCP Server is listening from. This way we have all the information needed to send data between two peers.

To interoperate with non-initiator peers that implement the protocol, we opted to keep sending the *CHUNK* message to the multicast, like in the vanilla Restore protocol, the only difference is that this messages doesn't contain a body. They act like control messages to the remaining peers.

III. Concurrency Design

For an efficient concurrent design we split the application workload in several smaller ones. Each Multicast Channel has its own thread, and their only purpose is to listen to all the messages sent through each one. After receiving a DatagramPacket it creates a new thread that runs the MessageHandler class. This class only job is to interpret the message received and start the protocol related to it. Although, for a more scalable solution, we needed to cut the overhead of creating and deleting threads, which is incurred once per message, so we use a thread pool (java.util.concurrent.ThreadPoolExecutor) avoiding the creation of a new thread to process each message.

This process of using thread pools is also used in the implementation of some protocols, such as the Backup Protocol, and the Restore Protocol, giving them an easy and efficient way to become multithreaded, where each thread is responsible for an chunk.

Other approach that we took was to use ScheduledThreadPoolExecutors. Since a lot of the protocol specification required the use of a given delay for an action the use of Thread.sleep() "for timeouts can lead to a large number of co-existing threads, each of which requires some resources, therefore limiting the scalability of the design.” ([Hints for Concurrency (6)]).

Finally, with so many threads running simultaneously, the use of appropriate data structures was imperative for the good performance of the application. These data structures needed to be able to support full concurrency of retrievals and high expected concurrency for updates, with all operation being thread-safe. The java.util.concurrent.ConcurrentHashMaps class was the solution. It has all the requirements that we needed, and obeys the same functional specification as Hashtable. Thus we keep the per protocol information stored in this class.