Distributed Backup Service
FEUP-SDIS 2020/21 – Project 1

Class 7, group 6
Breno Accioly de Barros Pimentel (up201800170@fe.up.pt)
Diogo Miguel Ferreira Rodrigues (up201806429@fe.up.pt)

12th of April, 2021

This report addresses the enhancements made to the **first project** of course SDIS (Distributed Systems) at the Faculty of Engineering of the University of Porto (FEUP). Enhancements are incremental; i.e., an enhancement implemented in a certain version is used in all posterior versions.

Parallelism enhancements

We defined a scheduled executor for each peer with 100 threads, which solves the issue of thread creation/deletion overhead. We picked a large number because most threads are not in use or are blocked waiting for other calls, so the actual number of running threads in CPU at each instant is relatively small.

One thread per multicast channel

By extending `Runnable` with generic class `SocketHandler` to handle incoming messages from a socket, and specializing `ControlSocketHandler`, `DataBroadcastSocketHandler` and `DataRecoverySocketHandler`, we can run the socket handlers in different threads. We could have avoided using an MDR handler as an initiator peer always knows it will receive packets on MDR only after sending a `GETCHUNK` message, but since there were many similarities with the way MDB works we decided to keep an uniform implementation.

Many protocol instances at a time

This is done using specialized `ProtocolRunnable`/`ProtocolSupplier` instances for each protocol instance. We extended `Runnable` with class `ProtocolRunnable` (which cannot throw exceptions) to represent a main protocol, which in turn we extended with classes `BackupFileRunnable`, `DeleteFileRunnable`, `RestoreFileRunnable`, `ReclaimRunnable` and `StateRunnable`. We extended `ProtocolSupplier` with `BackupChunkSupplier` and `RestoreChunkSupplier`, which are sub-protocols; this allows main protocols `BackupFileRunnable` and `RestoreFileRunnable` to run those sub-protocols in parallel. Each protocol instance has all the information it needs to run, and all protocol instances cooperate with the shared socket handlers to perform their jobs, and the shared sockets have appropriate synchronization mechanisms (mostly `synchronized` blocks) to avoid race conditions.

All protocol instances are executed concurrently and the `TestApp` that created those instances immediately returns, except for `StateRunnable` where `TestApp` only returns when it receives all state information and prints it, to allow state information to be printed on the same console `TestApp` is running from.

Processing different messages received on the same channel at the same time

`SocketHandler` is a wrapper which runs an infinite cycle that blocks while waiting for messages on `DatagramSocket#receive(DatagramPacket)`; when one arrives, the `SocketHandler` calls its abstract function `handle(Message)`, which each specialized socket handler must implement, usually just making some simple checks (like message type) and then calling `Message#process(Peer)`. Thus, all we had to do was to have the `SocketHandler` create a runnable on-the-fly where he calls `handle(Message)` and submit that to the peer executor, instead of the socket handler thread executing `handle(Message)` itself.

No `Thread.sleep()`

We do not use `Thread.sleep()` anywhere in our code; instead we used scheduled futures, the blocking call `Future#get()` and some future chaining with `then`-functions. We additionally invested efforts into reducing the number of running threads to the bare minimum (e.g., not having futures wait for other futures), particularly in avoiding the use of empty scheduled futures (which can be used as a valid and non-busy replacement for `Thread.sleep()`, but still take up thread resources).

Eliminate all blocking

We replaced all blocking calls for file-system access using non-blocking `AsynchronousFileChannel` and `CompletableFuture`, to use some `then`-calls and achieve a higher level of asynchronous execution.

Protocol enhancements

Backup enhancements

The issue is that, each time an initiator multicasts a `PUTCHUNK` message, all non-initiator peers will back up that chunk and rapidly exhaust their available memory. This is because a non-initiator peer only processes the `PUTCHUNK` message it receives, and doesn't bother with the rest of the network nor is there a node that assumes the role of coordinating the network.

We devised two complementary solutions for this problem.

Make wait time useful

Instead of saving the chunk and waiting for a random time period from 0 to 400ms before sending a `STORED` (which is currently only serving the purpose of avoiding packet collisions), a non-initiator peer can instead wait for that same random time period while listening to MC for arriving `STORED` messages, and by having the count of `STORED` messages that arrived by the end of the wait period it can either:

- Ignore the request, if the required replication degree was already met.
- Store the message and send its own `STORED` message, if the required replication degree was not yet met.

This method has the main advantage that it does not require the creation or modification of any messages, as it only changes the behavior of each peer, which will be aware of other peers' `STORED` messages instead of just ignoring them.

This method is expected to work the best in networks with least latency, as the guarantee that only as many peers with back-up the chunk as required is only valid if all peers receive messages instantly. This implies in high-latency networks the real replication degree will be quite larger than required.

This enhancement is implemented in version 1.2, and although we tested over a same-computer network the latency was appreciable (around 100ms), which meant that, in a network with 5 peers, one of the peers requesting to back-up a chunk with replication degree 2 would often lead to 3 peers backing-up the chunk.

Reclaim space from excessive backing-up

After the initiator peer waits for a certain time period (starting in the required 1000ms), it must count how many `STORED` messages it received relating to the chunk it just backed-up, so it can infer if the required replication degree has been met. However, from the `STORED` messages it receives, it can infer which peers backed-up the chunk, and if it notices there have been more peers backing-up than required, it can notify a certain number of them so they release the space they inadvertently used to store the chunk.

For that, we send over MC an `UNSTORE` message, with format:

`<Version> UNSTORE <SenderId> <FileId> <ChunkNo> <DestinationId> <CRLF><CRLF>`

which means the initiator peer with ID `<SenderId>` is notifying the peer with ID `<DestinationId>` that it should unstore a certain chunk. This enhancement is implemented in version 1.1.

Curiously, the previous enhancement improves the performance of this enhancement, as it first reduces the number of peers that backed-up the file without needing to, and this enhancement just solves those cases where latency is too high.

This enhancement could otherwise be made in a way that is less expensive in terms of disk usage, in which the initiator first multicasts a request for peers to tell if they are online (`GETONLINE`), the peers would answer they are online (`ISONLINE`) and indicate in the header if they can store a chunk or not, and the initiator would pick some peers to send the chunk to; the main disadvantage is that it would require two new messages, and either change `PUTCHUNK` to have a destination ID or make a TCP connection to each destination peer.

Restore enhancements

As per the enhancement statement, TCP is the solution:

- Initially, the initiator peer has to multicast its intention to restore a chunk, and then all peers compete over the multicast channel MDR to send the chunk
- With the enhancement, the peer creates a TCP socket, multicasts the intention to restore a chunk together with the TCP socket address, and then all peers compete to grab the other end of the TCP connection; the winner transfers the chunk, the others just ignore. If noone connects to the TCP socket, the initiator falls back to the original solution.

This requires a new message `GETCHUNKTCP` so we have a guarantee other implementations will ignore it:

`<Version> GETCHUNKTCP <SenderId> <FileId> <ChunkNo> <IP>:<Port> <CRLF><CRLF>`

which instructs all peers that have the requested chunk to try to connect to the socket with IP `<IP>` and port `<Port>`, and send the chunk.

The initiator will first multicast a `GETCHUNKTCP` message and try to get the chunk from the TCP connection. To maintain interoperability, the initiator must use the original protocol after it fails to find any peer that is able to connect to the TCP socket.

Deletion enhancements

The initiator peer of a certain file knows which peers stored chunks of that file, since it can remember which peers answered to its `PUTCHUNK` with `STORED`. Thus, when that initiator peer requests the service to delete a file, if the initiator peer can perceive who got the `DELETE` message it can then remember which peers still have chunks of the file but have not yet deleted them, and the next time the initiator realizes one of those peers is online it will resend the `DELETE`.

For that, we send over MC a `DELETED` message, with format:

`<Version> DELETED <SenderId> <FileId> <InitiatorId> <CRLF><CRLF>`

which means the peer with ID `<senderId>` is notifying the whole service (but particularly the initiator peer of the referred file) that it deleted all chunks of a certain file. This message is to be sent by a peer if it received a `DELETE` message and successfully deleted the chunks of the corresponding file.

This enhancement was implemented in version 1.1.

An additional possibility is for a peer that just went online to multicast an `ONLINE` message, thus allowing initiator peers that know that peer has not yet deleted a file they already requested to be deleted to realise said peer is online, and take action sooner. This was deemed unnecessary, as that peer will be noticed sooner or latter as most communications are multicast.