Serverless Distributed Backup Service

First SDIS Project

Sofia Cardoso Martins (up201606033)
Bruno Dias da Costa Carvalho (up201606517)

Faculty of Engineering
University of Porto

2018-2019

1 Concurrent Protocol Execution Model

Our concurrency model involves several independent threads per Peer, for each socket, and also several thread pools with configurable sizes, one for each task inciting a network response from the Peer.

1.1 Sockets

The Peer's UDP socket runs independently in one dedicated thread, extracting from a *public and concurrent thread-safe* message queue the datagram packets constructed by and added from other threads running in the same process. These packets are *not* grouped by destination channel, and are sent in order as without delay.

Similarly, the Peer runs each of the three *Multicasters* – one for each channel – independently in one dedicated thread. So we have a total of four independent threads running socket I/O. This enables the Peer to read from the channels as quickly as possible. The read DatagramPackets are passed to a dedicated Runnable – one for each channel – and submitted to the general peer thread pool, to be parsed and handled.

1.2 Received Message handlers

Now, each received *Message* is picked up by a thread from the pool, parsed, and handled if understood – we call this the *parsing thread*. The messages received can be divided in two groups as follows.

The first group of messages will incite the Peer to schedule a *Transmitter* thread that *might* answer the message after a short waiting period – this is the case for **PUTCHUNK**, **GETCHUNK** and **REMOVED** messages. The first one will trigger the creation of a *StoredTransmitter*; the second one will trigger the creation of a *ChunkTransmitter*; and the third one will trigger the creation of a *RemovedWaiter*.

Each of these transmitters has a *dedicated scheduled thread pool*, so three pools total, and they all follow a similar execution model: before creation, if the peer finds that an instance of these agents already exists *for the same chunk*, another one will *not* be created; otherwise, a new instance is constructed and scheduled to be run after the required short random delay, and will answer promptly if appropriate.

The second group of messages do not incite a response from the Peer: this is the case for **STORED**, **CHUNK** and **DELETE** messages. These messages are handled from start to finish by the *parsing thread*.

1.3 Subprotocols initiated by the TestApp

1.3.1 Backup and restore subprotocols

The *backup* subprotocol, when initiated by the RMI API, will prompt the creation of N instances of a *PutchunkTransmitter*, one for each chunk of the file being backed up. Similarly, the *restore* subprotocol will prompt the creation of N instances of a *GetchunkTransmitter*, one for each chunk being restored.

Unlike the previous transmitters, these transmitters have a more complex task that might require retransmitting their respective messages, as dictated by the specification. So each of them has its own *dedicated thread pool*, but their scheduling mechanism is a little different: to prevent overuse of the shared multicast channels, these transmitters are submitted to the thread pool once created, to be run as soon as a slot opens up; but instead of being rescheduled between retransmissions they **sleep**, not allowing other pending transmitters to be run while they wait. This is a design choice that creates a sliding window mechanism running through the file being backed up or restored (though the chunks might not be in order). The sleep period doubles every retransmission, as mandated by the specification.

The *PutchunkTransmitters* run independently, regardless of whether they succeed in reaching the desired replication degree, fall short of it, or get no backup peers at all. They stop retransmitting their chunk whenever the perceived replication degree is sufficiently high or they run out of retransmissions.

On the other hand, the *GetchunkTransmitters* are controlled by an instance of a *Restorer*, which coordinates the multiple transmitters for one file. If any of the transmitters fail to retrieve the desired chunk after the prescribed number of retransmissions, the parent *Restorer* cancels all pending and running *GetchunkTransmitters* it controls, as it will not be able to reconstruct the desired file from the collected chunks. This allows other *GetchunkTransmitter* instances (for other files) to proceed sooner.[1]

1.3.2 Reclaim subprotocol

The *reclaim* subprotocol, when initiated, will select the files to be deleted (if necessary, of course) based on a simple heuristic: chunks with a higher *surplus* or backup peers are elected for removal first. A chunk's surplus is merely its perceived replication degree minus the desired replication degree.

[1]The *Restorers* themselves also have a dedicated thread pool, though this is only a minor implementation detail to submit the various *GetchunkTransmitter* instances. There can be an arbitrary number of active *Restorers*.

A chunk with, say, +5 surplus has five more peers keeping it in their backup systems than was originally requested, so they can be removed without a major fuss. To avoid flooding the MC channel, the **REMOVED** messages are scheduled for transmission after the protocol-prescribed short random delay.

1.3.3 Delete subprotocol

This is the only subprotocol we have enhanced. The peer will create an instance of a *DeleteTransmitter*, which is scheduled on the peer's own thread pool, to transmit the **DELETE** message a certain number of times.

2 File Deletion Subprotocol Enhancement

In the *vanilla* version of the file deletion subprotocol, each peer would send a predefined number of DELETE messages, doubling the time interval between the transmission of consecutive messages, as would happen in the Chunk Backup Subprotocol. This solution is, however, inefficient as a peer that could have stored chunks of a deleted file would never free the space occupied by those chunks in the event of the peer not being up and running by the time the DELETE message was sent.
A better solution that would simultaneously take advantage of the same data stored for the Chunk Backup Subprotocol would be to wait for a backup peer to notify to owner of the file that was deleted whenever it deleted the chunks that it had stored for those files. As the owner of the file keeps track of which peers have stored the chunks of its files, it could simply send the DELETE messages until all those peers had responded with a certain type of message. This was the adopted solution. This enhancement implies, therefore, the implementation of a new type of message, namely the DELETED message When a owner peer receives a DELETED message it updates the data related to the ids of the peers which have a backup of any of the chunks of said file, and, if it detects that not all the file's chunks have a replication degree of zero, it reschedules the transmission of the DELETE message, doubling the time interval between the transmission of those messages. This process ends when all the backup peers of that file have sent the DELETED message.