SDIS Project 1 Distributed Backup Service

Nuno Oliveira – up201806525

Luís Miguel Pinto – up201806206

1. Enhancement specification and rationale

1.1. Backup Enhancement

For the backup it was required of us to guarantee the exact replication degree wanted for every chunk and not go over it to avoid the rapid depletion of backup space and too much activity on the peers once that space is full.

In order to do this, when a peer that is version 1.1 receives a PUTCHUNK message it waits for a random time between 0 and 800 milliseconds and if in that time frame it receives STORED messages of that specific chunk equal to the replication degree wanted by the backup protocol, it will not save a copy of the chunk and therefore not send the STORED answer. If all peers in the system are version 1.1 this implementation should guarantee that the real replication degree is always very close to the required. It is not always equal because it depends heavily on the number of peers, the delay the get for acting on a PUTCHUNK and travel and processing time of the messages.

Also, this protocol can interoperate with peers that implement only the 1.0 backup protocol because they will simply save the copy of the chunk indiscriminately and reply the same way the version 1.1 would if they would have a reply. If the number of 1.0 peers are lower than the desired replication degree, the effectiveness of the 1.1 protocol isn’t

affected, of course if the number is higher instead, the 1.1 peers will only save chunks when the others are full.

1.2. Delete Enhancement

This enhancement required that when the delete protocol is initiated and a peer with local copies of the file's chunks is unavailable (being offline) those chunks would still be deleted whenever the peer would be available again.

For this to work we added two new message types:

- AWAKE : <Version> AWAKE <SenderId>
- PURGED : <Version> PURGED <SenderId> <FileId>

Upon initiation every peer will send an AWAKE message to the multicast control channel.

Whenever a 1.1 peer starts the delete protocol it removes the file from his list of files and puts it in another list for files that are being deleted. When a peer with a local copy of the chunks receives the message DELETE and is also version 1.1, besides purging all info related to the file it will also send to the multicast control channel the PURGED message. When the initiator peer receives this message, it will update the peer list of every chunk on that file and when all chunks are not saved on any peer, the file is removed from the list. If a non-initiator peer is unavailable some chunks will survive the first DELETE and whenever it is available again it sends the AWAKE. Because the initiator peer still hasn't removed that file it shall send yet again the DELETE message to the multicast control channel.

This enhancement also works if the initiator peer becomes unavailable and available again before the non-initiator peer is back up and running since the file's folder will contain a new file "PURGING"

indicating it does not belong to the peer's file list, but its information is still needed.

Additionally, this implementation can interoperate with the 1.0 version either on the initiator or the non-initiator side of protocol.

1.3. Restore Enhancement

The enhancement for the restore protocol is to make connection between initiator peer and non-initiator peer happen only between themselves and not on a multicast channel (the use of TCP was also obligatory to be able to get full credit).

To implement the enhancement, we made the initiator peer and non-initiator peer communicate through a TCP connection where the former is the "client", and the latter is the "server". Because the connection is created on the peer that receives GETCHUNK message, there will be one TCP connection per chunk. If the non-initiator is version 1.1 it will start the connection and add to the CHUNK message the address and port for the other peer to connect to (it will also change the version of the message to 1.1).

- <Version> PUTCHUNK <SenderId> <FileId> <ChunkNo> <Address> <Port>

When the initiator peer receives the message and it is version 1.1 it will then join the TCP connection and read the chunk data through there, otherwise it will read the chunk data from the message itself thus guaranteeing the interoperability between peers with different versions.

Furthermore, both versions are interoperable except when the initiator peer is version 1.0. In this case it can only receive chunks from other 1.0 peers.

2. Concurrency Design

2.1. Threading

Each peer has 3 MulticastDispatcher objects that read/write to each of the multicast channels required. Each MulticastDispatcher has a thread pool (ExecutorService) of size 10 which it uses to read/write/process messages concurrently and in parallel,. The peer also has a scheduled thread pool (ScheduledExecutorService) of size 10 used to delay the execution of things.

Once a Dispatcher receives a UDP packet it uses the thread pool to execute a worker class that is responsible for the message processing.

Multiple protocols may be initiated at once on one peer, one thread is created by each “call” of the rmi protocol used. Every send to a multicast channel is delayed by a random time (depends on the specific message), that is achieved by scheduling functions using the thread pool.

The choice of the size of the pools was by empirical observation. More threads were found to lead to a poorer performance due to the changing overhead and also the threads responsible for reading the input were “starving” when used more than 20 threads per pool.

```
private ExecutorService pool = Executors.newFixedThreadPool(10);
private MulticastDispatcher mc;
private MulticastDispatcher mdb;
private MulticastDispatcher mdr;
private ScheduledExecutorService pool = Executors.newScheduledThreadPool(10);
```

2.2. Avoid Race conditions

To avoid race conditions, we use multiple instances of AtomicInteger, AtomicLong and ConcurrentHashMap. These objects allow for thread-safe access and edition of its values. Also in some cases,

the synchronized java mechanism was used to prevent race conditions in inside our classes.

2.3. Writing to files

To allow a better concurrency, and prevent threads from the pool to be used when writing to files, all writings to file are asynchronous (AsynchronousFileChannel) while readings are not. We chose to do this because reading only happens on peer startup and on the beginning of the backup protocol while writings are way more frequent.

This way the threads on the pool become available sooner.

```
private AtomicInteger agains = new AtomicInteger(0);
private ConcurrentHashMap<String, RestoreFile> fileRestoring = new ConcurrentHashMap<>();
private AtomicLong maxSize = new AtomicLong(-1);
private AtomicLong currentSize = new AtomicLong(0);
private ConcurrentHashMap<String, Boolean> chunkQueue = new ConcurrentHashMap<>();
private ConcurrentHashMap<String, RemoteFile> waitingForPutchunk = new ConcurrentHashMap<>();
private ConcurrentHashMap<String, File> waitingForPurge = new ConcurrentHashMap<>();
```

2.4. Schedule instead of Thread.sleep()

To avoid a larger number of co-existing threads and thus the need of way more resources we chose to only use the scheduled thread pool mentioned in the Threading section to delay any sort of operation. With this there is not a single use of Thread.sleep() in our application.

Even though while sleeping, threads are not using resources, they still are “using” one of the threads of the pool, leading to some cases were every thread on the pools were asleep(wainting for the sleep to end) and the application was not idle.(the same idea was behind the use of the AsynchronousFileChannel).