Faculdade de Engenharia da
Universidade do Porto

Project 1 - Distributed Backup Service

Final Report

SDIS 2020/2021 - 2nd Semester

3MIEIC05 - Grupo 06

Ivo Saavedra	**up201707093@edu.fe.up.pt**
Tiago Duarte da Silva	**up201806516@edu.fe.up.pt**

1. Concurrency Design

1.1. Thread safety and concurrent access

To safeguard against concurrent modification exceptions, we used data structures that are designed to be compatible with thread based applications.

One of these data structures is the ConcurrentHashMap, which as specified by Java Platform SE 8 Documentation “obeys the same functional specification as HashTable, and includes versions of methods corresponding to each method of Hashtable”. Therefore, this class allows us to maintain the integrity of the peer’s data and prevents exceptions thrown due to multiple accesses on an object by various threads at the same time.

While simply locking access to the entire data structure was a valid approach, the ConcurrentHashMap only locks certain partitions of the keys, allowing for much better performance when multiple threads are accessing different parts of the map, for example the backup protocol accessing one file’s details while a different thread returns the chunk data to a *GETCHUNK* request.

1.2. Multicast channel, message dispatcher & threads

Our implementation has one thread per multicast channel, many protocol instances at once and every message is handled concurrently. To achieve this, every multicast channel (backup, control and restore) have their respective instance of the MulticastChannel class. Each instance runs in a separate thread and has a ThreadPoolExecutor. The channel’s main thread is continuously listening for incoming messages and creates a new MessageThread with the received packet and submits it to the ThreadPoolExecutor.

The MessageThread class extends the Thread class and is responsible for parsing the packet’s message (handled by the Message class) and ignoring messages coming from the same peer or messages whose version is higher than that of the peer. This class is also responsible for dispatching the adequate response for a given message, as it uses the HandlerBuilder class to return an implementation of the Handler interface.

The HandlerBuilder class establishes a connection between every known message type and returns their respective handler for the MessageThread to run the command.

We have also ensured the replacement of all Thread.sleep() instances with more appropriate alternatives. In the cases where execution speed is crucial (ex.: backup protocol waiting for the replication degree), it was replaced by monitors (discussed below) as we wanted to be able to interrupt the thread and resume processing as quickly as possible. In other cases, such as handling the reply to *PUTCHUNK* messages needing to wait in idle for a predetermined duration, we created a DelayedThreadPoolExecutor in each peer to schedule these threads to avoid consuming system resources unnecessarily.

To maximize the performance of the ThreadPoolExecutors, we have tailored the core and maximum pool size, the keepAliveTime and the BlockingQueue implementation used. The backup and restore channels share the same ThreadPoolExecutor and, as their activity is not prone to bursts, we have chosen a SynchronousQueue. As the control channel is prone to burst activity from *STORED, DELETE,* and others, we have opted for a LinkedBlockingQueue which helps smooth out burst activity.

1.3. Protocol monitors

In order to allow the execution of multiple protocols and handlers concurrently and allow for a more effective thread management, we have employed the use of monitors to handle the waiting period between various messages (ex.: *GETCHUNK* awaiting the *CHUNK*).

The class ChunkMonitor handles both the waiting process (taking spurious wake ups into consideration) and the notification process.

On the backup protocol, the monitor is used in order to assert whether a given chunk's replication degree is already at the desired level. Whenever a chunk is sent via the multicast backup channel, a new monitor is assigned to it. Afterwards, when the *STORED* message related to that chunk is received the *resolveInitiatedChunk* function is called. In this function the peer checks if the replication degree for that chunk is fulfilled and, if so, the monitor is marked as resolved and notifies the backup protocol so that it can advance to the next chunk. Otherwise, the peer resends the chunk and awaits again for the confirmations. The waiting period for receiving the necessary *STORED* messages to resolve a chunk starts at a second and is duplicated for every failed attempt for that chunk.

```
 public void resolveInitiatedChunk(String fileHash, int chunkNo) {
     FileDetails file = this.initiatedFiles.get(fileHash);
         if (file == null) // only used on initiator peer
             return;

         Chunk chunk = file.getChunk(chunkNo);

  if(chunk.getPerceivedReplication()>=
file.getDesiredReplication()){
        ChunkMonitor monitor = file.getMonitor(chunkNo);
        if (monitor != null)
            monitor.markSolved();
  }
 }
```

Snippet 1 - resolveInitiatedChunk() called whenever a *STORED*message is received in order to keep track of the chunk's replication

Regarding the space reclaiming protocol, upon receiving a *REMOVED* message, the peer checks if the removed chunk still has the desired replication level or not. If so, then no response is triggered. Otherwise, a new chunk monitor is created in order to wait for a random delay of 0 to 400 milliseconds before sending a backup message for that chunk. If another peer has already detected the same problem and has already sent a *PUTCHUNK* message, then the chunk is marked as resolved and the current peer aborts before resending the same message.

```
public void resolveRemovedChunk(String fileHash, int chunkNo) {
    FileDetails file = this.storedFiles.get(fileHash);
    if (file != null) {
            ChunkMonitor monitor = file.getMonitor(chunkNo);
            if (monitor != null)
                monitor.markSolved();
        }
    }
```
Snippet 2 - resolveRemovedChunk() called every time a putchunk message is received

The last main use of the *ChunkMonitor* class is in the restore protocol, where the initiator peer awaits for the arrival of a *CHUNK*. For this case, the initiator will wait up to two seconds before timing out and resending the *GETCHUNK* request, repeating this process up to three times for each chunk. When receiving a *CHUNK* message, it will resolve the monitor associating its body to it (the data or TCP details of the enhancement).

1.4. Implementation limitations

As our monitor based system is unique for every file/chunk, our implementation allows for the execution of multiple protocols and handlers concurrently, both in the same peer or across multiple peers.

The only restriction applied is that we support just one operation at a time over the same file, for example, doing two backups/restores of the same file in the same initiator peer at once. Another example can be whenever one client app calls a backup protocol on a given file and another client concurrently calls the recovery protocol on the same file, then the restore operation won't be able to recover every file chunk and will fail.

The maximum supported file size is 64 GByte, limited by the chosen message system, whose chunk number must not go over 6 digits.

2. Enhancements

To achieve maximum compatibility, the peers will always ignore incoming messages whose version is higher than that of the peer. When a higher version peer receives an older version message, it will always use it and collaborate using the older established protocols.

2.1. Chunk Backup

For this subprotocol it was suggested that we try and find a way of reducing the inefficiency of the vanilla backup protocol, by only storing the necessary chunks to achieve the desired chunk replication. For this, we only allow peers to store a given chunk if its desired replication degree hasn't been reached yet.

Upon receiving a *PUTCHUNK* message, a peer checks if there is enough disk space for storing the chunk and only then adds it to the respective file. Afterwards, before sending the *STORED* confirmation message to the other peers, indicating that it has in fact stored this chunk, the peer checks if the perceived replication degree of the current chunk is already at the desired level. If so, the storage operation is cancelled and the chunk is removed from the file. Otherwise, the confirmation message is sent and the chunk is written to the disk.

This implementation is possible due to the random delay between 0 to 400 milliseconds before sending a *STORED* message to the other peers, which allows some peers to finish earlier than others, and thus send the confirmation message sooner. The peers who, by chance, had to wait longer until sending the confirmation will have received the *STORAGE* messages from those who finished first and updated their perceived replication degree in time to prevent sending the confirmation message and abort the storage operation.

The only downside to this implementation is that in rare cases the peer doesn’t receive the confirmation message sent by other peers in time, and, as such, is unable to stop the unnecessary storage of the chunk, leading to some chunks ending up with a greater replication degree than the desired one.

Having this in mind, we consider that our approach greatly reduces the amount of disk usage and, because of this, achieves the goal of the proposed enhancement.

2.2. Restore

The enhancement suggested for this protocol was to avoid flooding the multicast channel with the file data itself, as only one of the peers requires said data. As such, the chunk should be sent exclusively to the initiator peer, using the TCP protocol, while interoperating with non-initiator peers that implement the original protocol. This enhancement can be seen in the message.handlers.GetchunkHandler class and the Peer.restore method.

To implement this enhancement, we changed the original CHUNK message so that when replying to v2.0 peers, it will instead open a ServerSocket and send its IP and port and send this information in place of the chunk data itself.

The ServerSockets opened will timeout the acceptance of a connection after 2 seconds. This implies that threads will be waiting for a possible TCP connection from the initiator peer. As such, minimizing the number of threads in the peers waiting for a connection is an important issue. However, as the default protocol already cancels replies to GETCHUNK after it detects a CHUNK message for the same hash and chunk number, the threads will abort before opening the ServerSocket, reducing the idle time in the threads.

To guarantee the protocol cooperates with older peers, it responds with the correct version. Upon receiving a 1.0 version GETCHUNK it replies with a 1.0 CHUNK with the chunk’s data as the body. When replying to a 2.0 version GETCHUNK it will open the server socket and reply with a 2.0 CHUNK with the details required for the TCP connection.

To guarantee maximum compatibility, if a 2.0 peer somehow receives a 1.0 CHUNK, it will immediately store its data and advance to the next chunk.

2.3. Delete

The objective of this enhancement was to find a way of removing undeleted chunks of a file in a given peer in the event of a sudden crash before a delete operation. As such, the peer that crashed must reclaim the space used by the unremoved chunks upon recovery.

Since we receive a *STORED* message from the peer that has backed up a copy of a chunk, instead of only saving the perceived replication, we opted on saving the ID of the peer that had stored said chunk. Thanks to this implementation, we are able to track which peers contain each chunk.

In order to achieve this we decided that the best approach would be to create two new message types:

- <Version> DELETED <SenderId> <FileId> <CRLF><CRLF> - message sent by a peer who has deleted a file to inform the initiator peer confirming the file deletion (see message.handlers.DeletedHandler)
- <Version> INIT <SenderId> - message sent by a peer when it boots up. Used by the initiator peer to assert if there are any unremoved files in this peer caused by a crash (see message.handlers.InitHandler)

To keep track of the unremoved files, we created a new map which maps the IDs of the other peers to the hashes of the files they haven't deleted yet (Peer.undeletedFiles). The latter are represented internally as a set.

When the deletion protocol is called, the initiator peer fetches the FileDetails of the target file and adds its hash to every peer who has stored at least one chunk, on the undeleted files map.

Whenever a peer removes its local copy of a file, it sends a *DELETED* message back to the initiator peer. Upon receiving this confirmation, the initiator peer then removes the file hash from the unremoved files of the peer. If a *DELETED* message is never received, we assume that the peer was offline for any reason and didn't delete its local copy of the file.

When a peer comes online, it sends the aforementioned INIT message to inform the other peers that it is online again. Every other peer is then responsible for verifying if there are any entries in the unremoved files map concerning that peer. If so, they resend a *DELETE* message for every file in that peer's entry.

This system left a vulnerability however, backing up a file, then one of the peers that stored it goes offline, and finally deleting and backing up the same file again (the hash does not consider *when* a file was backed up). Whenever the peer that was offline reboots, the initiator peer would detect that it had files to delete and send a *DELETE* message to the control channel. This would cause all other peers to delete the newly backed up file (same hash).

To avoid this issue, when the backup protocol is initiated, any entries of that file's hash are deleted. This ensures that the *DELETE* message is not wrongfully sent.