Mestrado Integrado em Engenharia Informática e Computação

Sistemas Distribuídos

Distributed Backup Service

(April 14, 2020)

T4G06

Gustavo Torres, up201706473

Joaquim Rodrigues, up201704844

1. Enhancements

1.1. Backup Enhancement

The purpose of the Backup Enhancement is to avoid storing the chunks as much as the system allows, thus depleting the backup space rather rapidly, while maintaining interoperability with other versions of the protocol.

The solution adopted consists of keeping track of how many peers have stored a chunk that has been received by a PUTCHUNK message, by counting the number of STORED messages received of that chunk by other peers, and after the random delay between 0 and 400 milliseconds, if the perceived replication degree is greater or equals to the desired replication degree of that chunk, the peer deletes its own copy of that chunk because the chunk as already achieved it desired replication degree, otherwise it sends a STORED message. This way, the peer with this enhancement does not contribute to the over storing of a chunks. To note: when a peer receives a PUTCHUNK message, it stores the chunk immediately, and if needed the peer deletes it.

If all peers have this enhancement, the chunk replication degree is never above the desired replication degree, guaranteeing that the minimum space is used to achieve the desired replication degree, and the system occupied space is keep at its minimum, avoiding high activity in nodes.

This enhancement interoperates with other peers that execute the previous version of the protocol because no additional neither modified messages are sent.

1.2. Restore Enhancement

The purpose of the Restore Enhancement is to increase the efficiency of the Restore Protocol, by using TCP connection while interoperating with the base version of the protocol.

In order to do so, we implemented a **ServerSocket** (ServerThread class) associated with each peer. This server socket will allow, when a peer has the requested chunk and the enhancement version, the connection to a **Socket**, where a CHUNK message that contains the

chunk data will be written. This message is processed by the ClientHandler class and the chunk data will be added to the requesting peer's FileInfo.

The socket information, address and port, is sent in the body of the GETCHUNK through the MCC so that the peer who has the chunk data can establish TCP connection.

The interoperability of the two Restore Subprotocols is assured because the Control Channel, when receiving a GETCHUNK message, will evaluate the presence of the chunk in that peer sending the message containing the chunk (ControlChannelMessageReceived class) and the first peer to send the CHUNK message (no matter the version) will block the sending of this message to all other peers (RestoreChannelMessageReceived class). By doing this, we are both reducing the load of both the Channels and, if the peer version corresponds to the enhancement one, directly sending the chunk data to the requesting peer through the TCP connection, reducing the amount of the attempts to find it.

To create the server socket, the port used is equivalent to 1025 + (peerId + offset) % (65536 - 1025), this expression allows the system to avoid ports below 1025 and above 65536, where peerId is the id of the peer, which is unique, however, only using this does not avoid collisions on ports, depending on the ids, so we added an offset, which, if a collision is detected, it increases by one until a port is available, however, if the offset is greater than (65536 - 1025), we conclude that no port is available.

1.3. Delete Enhancement

The purpose of the Delete Enhancement is to recover a peer's disk space filled with data that was deleted by the initiator peer when it was not running.

To achieve this enhancement we found important that the initiator peer knows the maximum chunk perceived replication degree of the files it has initiated a backup protocol, as it is the one that initiates the backup of that file and the only one that can delete it from the system. This number is important because it represents the number of peers that have stored any part of the file. Given this number the initiator peer can know how many peers still have some chunk of the file.

With this enhancement, when the initiator peer starts the DELETE protocol, it stores an entry for that file with this maximum perceived number and every peers that has a copy of the file will send a new message, a DELETED message, through the Control Channel, saying that the file is no longer stored in that peer, so that the initiator peer can know how many peers have deleted the file. However this alone does not overcome the problem. To do this, when a peer starts running, it starts a DELETE protocol for each file it has deleted and was not deleted from every peer, and besides that sends a message by the Control Channel, a new message, TODELETE, that essentially asks the peers in the system what files they have deleted but were not fully deleted from the system. The peers that receive this TODELETE message, wait after a random delay between 0 and 400 milliseconds and then start a DELETE protocol for each file in this conditions.

This system is not perfect because if the peers that has a part of the chunk and the initiator peer are not running at the same time, no DELETE message is not sent, so the peer that has the chunk can not know that the file was deleted.

The interoperability is assured because the base messages, of the base protocol, are still sent with no modifications, and any new messages that are not recognized by the older version are ignored. Thus, if there is any peer with older versions the enhancement might not work.

2. Concurrent Execution of Protocols

The concurrent execution of protocols was achieved by the design in the implementation of the different subprotocols. Our implementation consists of a one thread per multicast channel, where different messages can be processed at the same time, by creating a thread by each one of them. The implementation with RMI is also used. In order to do so, it was pivotal to choose the right data structures, like ConcurrentHashMap, as well as AtomicInteger and AtomicBoolean types, allowing to improve its reliability and avoid race-condition in a multithreaded environment.

Each multicast channel runs on a thread, and, upon receiving a message, creates a threads that processes it, and does the appropriate action. Each message sent is also sent using a thread, a with the use of the ThreadPoolExecutor, it is possible to send a message with a given delay without flooding the program with threads in idle when they use Thread.sleep(). So our

implementations tries to minimize the number of co-existing threads running by avoiding the Thread.sleep() function.

Furthermore, the Backup and Restore protocol can send several messages using different threads without necessarily receiving the previous one because of the multiple message processing described above.

To avoid several threads accessing the same data and modifying it, creating race-conditions and unreliable information, the use of ConcurrentHashMap and AtomicInteger in important data accessible to multiple threads at the same time, is imperative.

To conclude, our program by the use of RMI, one thread per channel and a thread processing/sending each message we achieve multiple protocols at a time while guaranteeing the reliability of the program data.

3. Annexes

```
scheduler_executer = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(64);
```

Image 1 - ScheduledThreadPoolExecutor initialization

```
private ConcurrentHashMap<String,Integer> chunkReplication = new ConcurrentHashMap<String,Integer>();
private ConcurrentHashMap<String,FileInfo> restoreFile = new ConcurrentHashMap<String,FileInfo>();
private ConcurrentHashMap<String,Integer> requestedChunks = new ConcurrentHashMap<String,Integer>();
```

Image 2 - Initialization of different ConcurrentHashMaps, needed to identify file and chunks data

```
private AtomicInteger numChunksWritten = new AtomicInteger(0);
private AtomicInteger numChunksBackedup = new AtomicInteger(0);
private AtomicInteger backupState = new AtomicInteger(0);
private AtomicInteger restoreState = new AtomicInteger(0);
```

Image 3 - Initialization of different AtomicIntegers, pivotal to the processes of backup and restore

```
MC = new ControlChannel(mcAddr,mcPort,this);
MDB = new BackupChannel(mdbAddr,mdbPort,this);
MDR = new RestoreChannel(mdrAddr,mdrPort,this);
```

Figura 4 - Channels initialization. Each channel will be an independent thread