Universidade do Porto
Faculdade de Engenharia da Universidade do Porto - FEUP

Master in Informatics and Computing Engineering - MIEIC
2020/21

Distributed Systems - SDIS

Report on Project 1 - Distributed Backup Service

Class 3 Group 08

Catarina Justo dos Santos Fernandes - up201806610
Flávia Carvalho Gavinha Pereira Carvalhido - up201806857

Index

Concurrency design

Each Peer creates 3 Multicast Sockets (one in each PeerMultiThread class: PeerMultiThreadBackup, PeerMultiThreadControl, PeerMultiThreadRestore). They are responsible for receiving messages from other Peers and delegating the messages to the MessageHandler (each PeerMultiThread class has an instance of a MessageHandler). The process of delegating messages consists of following the "Processing of Different Messages Received on the Same Channel at the Same Time" methodology using an instance of the ThreadPoolExecutor class to achieve a more scalable solution.

```
public class PeerMultiThreadBackup implements Runnable {
    private String multicastAddress;
    private int multicastPort;
    private MulticastSocket multicastBackupSocket;
    private ExecutorService workerService;
    private MessageHandler messageHandler;
    private final int BUFFER_SIZE = 64000;

    public PeerMultiThreadBackup(Peer peer, String version, String multicastAddress, int multicastPort, int nThreads) throws IOException {
        this.multicastAddress = multicastAddress;
        this.multicastPort = multicastPort;
        this.messageHandler = new MessageHandler(peer);

        // join multicast socket
        InetAddress group = InetAddress.getByName(multicastAddress);
        this.multicastBackupSocket = new MulticastSocket(multicastPort);
        this.multicastBackupSocket.joinGroup(group);

        // start worker service
        this.workerService = Executors.newFixedThreadPool(nThreads);
    }
```

Figure 1 - Implementation of an Executor service in the *PeerMultiThreadBackup* class

Each PeerMultiThread class has an instance of the ExecutorService class *workerService,* which uses a thread pool to achieve concurrency by dispatching each received message to a different thread, who then takes care of processing the message, by calling the *handle* method from the *MessageHandler* class.

```
public void handleMessage(byte[] packet, String packetAddress, int packetPort) {

    Runnable processMessage = () -> this.messageHandler.handle(packet, packetAddress, packetPort);

    this.workerService.execute(processMessage);
}
```

Figure 2 - *workerService* execute() call in the PeerMultiThreadBackup class

The Peer class also uses the same methodology by creating an ExecutorService instance - *service* - that uses a thread pool in order to run different protocol operations at the same time. When the TestApp calls a function from the RemoteInterface, it will be dispatched to another thread that performs the TestApp request.

As this design implies there will be many operation instances occurring in multiple threads at the same time, there is a very high probability that the program will need to access the same data simultaneously. Our solution for that problem was to use concurrent data structures such as the

ConcurrentHashMap and the ConcurrentSkipListSet as they provide a thread-safe way to access shared data stored in volatile memory.

```java
// Other's chunks that this peer stored
public ConcurrentHashMap<String, BackupChunk> backedUpChunks;
// Owned files that initiator peer asked to be backed up
public ConcurrentHashMap<String, BackupFile> files;
// Saves the set of peers where backed up chunks are/chunks belonging to a file are
public ConcurrentHashMap<String, ConcurrentSkipListSet<Integer>> chunksLocation;
// Keeps all the deleted files and their peer location
public ConcurrentHashMap<String, ConcurrentSkipListSet<Integer>> deletedFilesLocation;
// This saves the body of the chunks that will be used to restore a file
public transient ConcurrentHashMap<String, byte[]> toBeRestoredChunks;
```

Figure 3 - Concurrent data structures used in the *DiskState* class

In order to not lose data between different runs of the program, these data structures are saved in their respective directory as they implement the Serializable interface. Every time a Peer is killed by a SIGINT signal, an instance of the TerminatorThread class catches said event and runs some cleanup code which saves all the data structures in the DiskState class to a *.ser* file.

To avoid using Thread.sleep() calls for the delays in some protocols or for sending repeated messages (as they can lead to a large number of co-existing threads, each of which requiring some resources and therefore limiting the scalability of the design) we opted to use a ScheduledThreadPoolExecutor, using schedule() calls instead. This way blocking time in the threads is eliminated, freeing up resources and guaranteeing better performance.

```java
public void run(){
    try {
        byte[] messageInBytes = this.message.convertToBytes();
        String chunkId = message.header.fileId + message.header.chunkNo;

        if (!this.peer.storage.chunksLocation.containsKey(chunkId) ||
                this.peer.storage.chunksLocation.get(chunkId).size() < message.header.replicationDegree){

            MulticastSocket socket = new MulticastSocket(this.message.port);
            socket.setTimeToLive(1);
            socket.joinGroup(InetAddress.getByName(this.message.address));

            //sending request
            DatagramPacket putChunkPacket = new DatagramPacket(messageInBytes,
                    messageInBytes.length,
                    InetAddress.getByName(this.message.address),
                    this.message.port);

            socket.send(putChunkPacket);
            socket.close();

            if (this.tries < 4){
                scheduler.schedule( command: this, (long) Math.pow(2, this.tries), TimeUnit.SECONDS);
            }
            this.tries++;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 4 - Implementation of a ScheduledThreadPoolExecutor schedule() call in the *PutchunkTask* class

Another mechanism to implement a concurrent design was using the *Runnable* interface. Upon receiving a message, the MessageHandler in our program will parse it and create a *Runnable Task,* according to the message received. This will create a new thread to run the created task, freeing MessageHandler up and enabling it to spend most of its time receiving and dispatching new messages.

```
public abstract class Task implements Runnable{
    protected Peer peer;
    protected Message message;
    protected Header header;
    protected BackupChunk chunk;
    protected ScheduledThreadPoolExecutor scheduler;
    protected final int NUMBER_OF_WORKERS = 5;

    public Task(Message message) { this.message = message; }

    public Task(Peer peer, Header header, BackupChunk chunk) {
        this.peer = peer;
        this.header = header;
        this.chunk = chunk;
    }
```

Figure 5 - Abstract *Task* class implementing the Runnable interface

```
switch (newHeader.messageType) {

case "PUTCHUNK":
    if (newHeader.senderId != this.peer.id
            && this.peer.storage.occupiedSpace + body.length <= this.peer.storage.maxCapacityAllowed
            && !this.peer.storage.files.containsKey(newHeader.fileId)) {
        String chunkId = newHeader.fileId + newHeader.chunkNo;
        BackupChunk newChunk = new BackupChunk(chunkId, newHeader.fileId, newHeader.chunkNo, body.length,
                newHeader.replicationDegree, body);

        // CREATES TASK THAT SENDS STORED MESSAGES
        StoreTask newTask = new StoreTask(this.peer, newHeader, newChunk);
        newTask.run();
    }
    break;
```

Figure 6 - Creation of a new Runnable *StoreTask* in the *MessageHandler* class

Enhancements

Backup enhancement

The Backup protocol can cause a rapid depletion of the backup space in the peers and also cause too much activity on the nodes once the available space of a peer is full. To avoid this, we thought of a scheme that can interoperate with the simple version of the protocol while also diminishing the number of chunks stored while ensuring the desired replication degree.

Instead of immediately storing the received chunk, the *StoreTask* waits for a random backoff time between 0 and 600 ms (using a ScheduledThreadPoolExecutor schedule() call), to collect STORED messages from other peers. After that, it checks if it already received a STORED message for that chunk, and if it has it doesn't save the chunk in the corresponding ConcurrentHashMap.

This way there is a decrease of STORED messages in the network, achieving the goal of lessening the node activity once the replication degree is reached. It is also a way to save storage memory in the peers, making the program able to provide the backup service to more files.

However, as this enhancement relies heavily on the chances of the random backoff time being bigger than the random delay used while sending the STORED messages, there are still some occurrences of the replication degree being higher than the desired one, thus not being able to achieve maximum efficiency.

Delete enhancement

The Delete Protocol ensures the complete deletion of a file from the backup service. However, if a Peer was to go to sleep after backing up a file and missed out on a delete message for said file, it would end up having unnecessary space occupied with said file's chunks that no longer need to be backed up in the system.

As to ensure the complete deletion of the chunks, the enhanced delete protocol must keep track of the deleted files at all times and where the chunks to said files are located. When the asleep peer wakes up, it must check if those deleted files were stored in said peer and must resend that message. That poses two problems: how to store the deleted files and their chunk locations to later send another delete and how to know when the peer wakes up.

As for the first problem, the deleted files are stored in a *ConcurrentHashMap*, each entry containing the file ID and its respective list of Peers where this file has stored chunks. This map is updated every time a peer starts a *DeleteTask* and each peer has their map of deleted files. If a backup of that same file happens, the initial entry is deleted and the map is updated, thus ensuring that only the no longer needed files are deleted and no redundant backup location is deleted needlessly.

Regarding the second problem, the solution implemented is to have a new type of message *HELLO* that the enhanced peer should send when waking up. Upon reception of this message, the other enhanced peers that have files in their *deletedFilesLocation* map, whose locations include the peer that was previously asleep, must start the delete protocol for that file. The peer will then update its maps and delete the no longer needed chunks, making up space for other needed backups.

```
case "HELLO":
    if (this.peer.version.equals(ENHANCED) && newHeader.senderId != this.peer.id) {
        for (String fileId : this.peer.storage.deletedFilesLocation.keySet()) {
            if (this.peer.storage.deletedFilesLocation.get(fileId).contains(newHeader.senderId)) {
                Header header = new Header(this.peer.version,  messageType: "DELETE", this.peer.id, fileId);
                this.peer.sendDelete(header);
            }
        }
    }
    break;
```

Figure 7 - Handling a *HELLO* message in the *MessageHandler* class

```
public void run() {
    try {
        byte[] messageInBytes = this.message.convertToBytes();

        MulticastSocket socket = new MulticastSocket(this.message.port);
        socket.setTimeToLive(1);
        socket.joinGroup(InetAddress.getByName(this.message.address));

        // sending request
        DatagramPacket deletePacket = new DatagramPacket(messageInBytes, messageInBytes.length, InetAddress.getByName(this.message.address), this.message.port);
        socket.send(deletePacket);

        socket.close();

        if (this.tries < 4) {
            Random rand = new Random();
            int upperbound = 401;
            int randomDelay = rand.nextInt(upperbound);   //generate random values from 0-400

            scheduler.schedule( command: this, randomDelay, TimeUnit.MILLISECONDS);
        }
        this.tries++;
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 8 - Implementation of the HelloTask, which sends a HELLO message a maximum of 5 times