2º Pratical Assignment - Report

Report – MIEIC

Course: Sistemas Distribuidos

Authors:

- Ana Rita Fonseca Santos – up201605240@fe.up.pt
- André Filipe da Silva Moutinho – up201704889@fe.up.pt
- Eduardo João Santana Macedo – up201703658@fe.up.pt
- Sérgio Bruno Rodrigues Dias – up201704889@fe.up.pt

May 2020

Academic Year 2019 / 2020

Index

1. Introduction

This report serves as a complement for the second practical assignment, having the purpose of specifying the design and implementation of the backup service. It also describes the extra features that were implemented to raise the project's grade's ceiling.

2. Overview

The implemented solution allows for the backup, restore and deletion of files. There is also a state operation, which shows the client the storage status and the files that a give peer has initiated backup for.

JSSE was used for secure communication. Although the project members planned and attempted to use the more complex interface **SSLEngine**, the difficulties encountered in the process and shortage of time led to the decision of changing the communication mechanism halfway through the project development. Therefore, the final solution uses **SSLSocket** instead**.**

As for the scalability and fault-tolerance, those are ensured at the design level. The implemented design is decentralized and conforms with the Chord protocol/algorithm, which was incorporated to satisfy both requirements.

This means that the ceiling of our project's grade should be 19 out of 20.

3. Protocols

3.1. Backup

The protocol is initialized with the following command:

$ java src.service.TestApp <*peer access point*> **BACKUP** <*filename*> <*replication degree*>

The file to backup should be placed in the folder ***build/files_to_backup.***

The backup process involves the following steps:

1. Initially, the class TestApp connects to the right peer via RMI, and then executes the following function *backup(file_path, replication_degree)*.
2. A new thread is initialized in the correct ChordNode, which creates the key file, stores the key, the file path and its desired replication degree.
3. Secondly, a FIND_BACKUP_NODE message is created with the necessary information, and the thread calls *find_successor_addr*(key_file, msg). This last function checks if the current node successor is the correct node to backup the file and from here two things can happen:
   - If the previous is true in the first iteration, it will return the successor's IP and from there, since we still are on the node that started the request, we send directly the file to the correct node with a BACKUP_FILE message. If it is false, the node will forward the query (with the FIND_BACKUP_NODE message) using full advantage of the chord implementation and each node's finger table;
   - On the following iterations when a node receives a FIND_BACKUP_NODE message, it once again checks if its successor is the correct node. If this is the case, this time it will send a message to the peer that first originated the query with the IP address of the node to back up the file. Once the peer receives this message it will then send the file to the node with the IP address it received.

4. Once a Chord Node receives a BACKUP_FILE message containing the file it will store the file in the correct place and update its *files_backed_up* hashmap so that the information is up to date. Secondly, it will handle the replication degree:

- When the Replication Degree received in the message is higher than one, the node will subtract this value by one and ask its successor to backup the file. This will go on until the value of the replication degree is one, meaning that we have achieved the replication degree requested by the user. There are two cases in which the file is not stored in a peer and is just redirected to its successor: the peer does not have space for the file or it's the backup protocol's initiator. In both these cases the file key is stored in the *cancelled_backups* HashMap so that this information can be used in other protocols. In case there are not enough peers in the network, or there are not enough peers with space to store the file that is being backed up, once the message that is being looped through consecutive successors reaches the peer that started this message chain, then it stops and displays on the console that the desired replication degree could not be achieved.

3.2. Restore

The protocol to restore a file is initialized with the following command:

$ java src.service.TestApp <peer access point> ***RESTORE*** *<filename>*

The restore process involves the following steps:

1. Once the TestApp is connected to the specific peer via RMI, the restore function is called and executed in this ChordNode.
2. In the ChordNode a new *Restore* thread is created, with the *node* who invoked the restore protocol and the *file_name* as arguments.
3. This thread is responsible for making three important tasks, depending of the constructor:
    a. **SEND_RESTORE_MSG:** The first task, handles the process of finding the key for the file. With this key, it stores the path of the file in the node's *files_restored* HashMap.

       Following that, it creates a message of type FIND_RESTORE_FILE and tries to find in the chord circle the node that has that specific key file.

       To reach the node as quickly as possible, the first approach is to find the node that is supposed to have the file (the successor of key file) always using the search algorithm O(log N).

       The search starts in the node itself, and the message created is to communicate between the nodes within the circle when needed. When the message is received, they continue the search.

       When a peer finds the successor of the key, a message of type FOUND_RESTORE_FILE is sent to the peer responsible for said key. When the key's successor receives the message, a new Restore thread is created and assigned the task EXECUTE_RESTORE.

    b. **EXECUTE_RESTORE:** If a node creates this type of *Restore* thread, that means he is supposed to have a file that he needs to send to the node who requested restore (the FIND_RESTORE_FILE message sent in the constructor contains in the header the IP address of the node who started the restore request in the first place).

       In case the file exists in the peer's file system, it is converted to bytes and stored in the body of a new message of type RETRIEVE_FILE and sent to the specific peer who requested the restore. Otherwise, the message received is forwarded to the peer's successor until a peer that has the file receives it. The query stops when it goes through the whole circle, meaning that the restore cannot be executed.

Once this message reaches the destination node, a new Restore thread is created with the node and the message received as arguments, performing the third and the last task, SAVE_FILE.

c. **SAVE_FILE:** When a node starts this thread, the message sent in the constructor is parsed, and the file is created in the *peer_key/restore* directory, with the name corresponded to the key file in the files_restored HashMap. Once the file is created, the key is removed of the files_restored HashMap.

3.3. Delete

The protocol to delete a file is initialized with the following command:

$ java src.service.TestApp <peer access point> ***DELETE*** *<filename>*

The delete process involves the following steps:

1. Once the TestApp connects to the specific peer via RMI, the delete function (implemented in ChordNode) is called.
2. Here a new *Delete* thread is created with the *node* that is initiating the delete protocol and the *file_name* as arguments.
3. This thread is responsible for making two important tasks, depending on the constructor:
   a. **SEND_DELETE_MSG:** Handles the process of finding the key for the file and checks if the node is responsible for that key file, in other words if it is the successor of the key (the first node that tried to backed up the file).

      If the query returns true it tries to delete the file automatically, calling *delete_file()*.

      Otherwise, if the node isn't the successor key, it creates a message of type FIND_DELETE_FILE_NODE, and starts a search for that specific node in the circle using the Chord's optimized search algorithm to jump between nodes and

      the message referenced before to communicate between them. When the message is received, the node verifies if its responsible for that key as described above, if not, they continue the search. The search ends when the message reaches the successor of the key file. When that happens, a new *Delete* thread is created with the information of the node and the key file as arguments. The second task EXECUTE_DELETE is initiated.

Since the file no longer exists, in the end of this task the *Delete* thread removes the key file from the node's *files_backed_up* HashMap, responsible for storing information about the files the node backed up.

b. **EXECUTE_DELETE:** This thread executes the function *delete_file()*, and as the name indicates, the function is responsible for deleting the file.

Similarly to the backup protocol, the peer deleting a file sends a message to its successor so that all of the copies of the file can be deleted. A peer sends a message to its successor only when it has the file that is being the deleted, it is the responsible for that file, it was the protocol initiator or the file is contained in the *cancelled_backups* HashMap. Otherwise, no more messages are sent. This ensures some optimization, since it is more efficient than looping through all the nodes in the network.

3.4. Reclaim

The protocol to reclaim a peer's disk space is initialized with the following command:

$ java src.service.TestApp <peer access point> ***RECLAIM*** *<disk_space>*

Initially, the reclaim protocol was planned to be done on the server side, but with current backup implementation the actual replication degree will never be higher than the desired one so when trying to backup a file in a peer that does not have enough space, the file is forwarded to the peer's successor instead of initiating the reclaim protocol. Calling the reclaim protocol on the server side would just affect the protocol's efficiency. Instead, the reclaim protocol is called in the client side.

Once the protocol is initiated in the correct peer, a new Reclaim thread will be called to execute the protocol and the peer will start by setting its maximum disk space as the one desired by the client. In case new maximum storage is lower than the space previously used, files will be deleted until there is enough space for all of the files in the peer. For each of the deleted files, the backup protocol will be initiated and their key will be stored in the *cancelled_backups* HashMap.

3.5. State

The protocol to request a peer to show its state is initialized with the following command:

$ java src.service.TestApp <peer access point> ***STATE***

The state protocol is rather simple. It was mostly designed for debug purposes, displaying some relevant information that helps ensure that the peer-to-peer system is behaving as expected. This information consist of:

- List of files whose backup was initiated in that peer;
- Key value and desired replication degree for each of those files;
- Disk status: total, available and used storage.

4. Concurrency Implementation

We developed a multi-thread oriented application, thus ensuring better efficiency in comparison to a single-threaded implementation. That was possible since the workload can be spread across multiple threads.

Whenever a peer is requested to execute a protocol, a new thread is created to attend that request, allowing the same peer to immediately start responding to another possible request. It's worth noting that we are using a thread pool, since each of those threads is initialized through an ExecutorService.

Going deeper into the implementation details, we have a class for each operation (be it backup, restore, delete, reclaim or state). They all implement the Runnable interface, so we just instantiate and use them as the argument when we call the execute or submit functions from ExecutorService.

It is also important to note that some functionalities include different stages or behaviors that need to take place in each circumstance. That said, the run() method will perform the action specified in the constructor. This specification is not always done by a parameter, but rather by calling different constructors for different actions. Given the fact that each of them requires different information in most scenarios, we found overloading to be the obvious and best solution. The same reasoning was followed for the rest of functionalities, when applicable.

The MessageReceiver class handles any received messages. The handling may lead to the call of another thread, which will execute an instance of functionality protocol, or just forward a message to another peer.

As a final note, it's worth mentioning we used the ***ConcurrentHashMap*** class to protect data that is frequently accessed and updated by threads that run concurrently.

For testing purposes, we created a file named "script.bat" which can be run from the build folder and tests the concurrent execution of the protocols (stress-test).

5. JSSE

As mentioned earlier, this project does make use of JSSE for a safer communication. The security is ensured through the usage of Secure Sockets by the classes **SSLServerSocket**, **SSLServerSocketFactory**, **SSLSocket** and **SSLSocketFactory**, all from the javax.net.ssl package. Usages of this implementation can be seen in the file **Server.java**, which initializes the server and has loop which runs in the background waiting for new connections. Also, the file **MessageSender.java** has the functions used to connect and send messages to a peer. This secure communication is always used between the peers in the network, i.e. in each message sent, thus being involved in every protocol mentioned above.

The cipher-suites used for message encrypting are included in the files **client.keys**, **server.keys** and **truststore** (directory **keys**), provided in lab5, which serve the purpose of authenticating a client when the latter tries to establish a connection (authentication is enabled with the command *server_socket.setNeedClientAuth(true).*

6. Scalability

Scalability of the application was ensured by implementing the Chord protocol/algorithm. This protocol revolves around a distributed hash table (DHT) to help in the resolution of unstructured names in the network. A DHT stores pairs (key, value) by assigning keys to different peers/computers in the network. A node only stores the values of the keys it is responsible for. Both values and peers are assigned an *m* bits identifier using consistent hashing, where 2^m = maximum number of peers in the network.

Through the usage of this protocol, each peer only needs to keep track of around $\mathbf{\log_2 m}$ other peers, being **m** the amount of peers currently in the network. These peers' information is stored in what's called a **finger table**. The finger table helps in a way that querying for a key that is stored in the network is fast, since the number of nodes that must be contacted to find a successor in an *N*-node network is O(log N).

As mentioned earlier, implementing this protocol ensures scalability, since adding nodes to the network will have a low impact on the memory usage and the number of nodes required to query a key from the network.

Reference to important functions that implement chord (all in ChordNode.java):

- **join(InetSocketAddress)** : line 184;
- **notify_successor()** : line 215;
- **notified(InetSocketAddress)** : line 223;
- **find_successor_addr(long, Message)** : line 239.

7. Fault-tolerance

The design chosen was decentralized, which means that fault-tolerance was ensured by Chord's fault-tolerance features. There are three threads running in the background that ensure the stability of the network:

- **FixFingersThread**: Every three seconds checks whether the peers in the finger table are all alive and updates the finger table in case of departure and join of nodes;

- **PredecessorThread**: Pings the predecessor and in case it doesn't reply, sets it as null so that it can be updated later through the usage of a notify;

- **StabilizeThread**: Asks the successor for its predecessor and decides whether that predecessor should be the new successor, then notifies its successor, so that it can update its predecessor.

These threads are respectively implemented in the files: **FixFingersThread.java**, **PredecessorThread.java**, **StabilizeThread.java**, and only start running when the peer joins the network.