Project 2 - SDIS - Group 21

Members:

João Mascarenhas, up201806389
João Matos, up201703884
Luís Pinto, up201806206
Nuno Oliveira, up201806525

Index

Overview

The project supports all the operations/features that were supported in the first project, this includes backing up a file, restoring a file, deleting a file, setting a limit for the amount of storage used by a peer and getting information about a peer's state. Additionally, our project is a decentralized P2P solution that uses threads to achieve the concurrent execution of requests and makes use of JSSE (with SSLSockets) to improve security. Finally, we make use of Chord which allows us to improve fault tolerance and scalability (for this last one we also use thread pools and Java NIO for writing asynchronously).

Protocols

For the communication between the peers and clients RMI is used, for everything else TCP is used. If a peer receives a message with which it can't do anything then it sends it to its successor to see if that peer can do something about it. If no peer can do anything then an error message is sent to the initiator peer. The error message has the following format: PUTERROR <initiator id> <file id> <replication degree> where the first argument is the ID of the peer which initiated the failing process, the second argument is the ID of the file being processed, and the last one is the replication degree of the file. A large portion of the code related to the following protocols is in the Handler.java file (the portion related to processing received messages between peers).

Backup

When a peer receives a backup request it checks if the file has already been backed up and if it has then it checks if the file has been changed, if the file hasn't been backed up yet or has been changed then the back proceeds normally (if the file had been backed up before then it deletes the previous version from all peers). The file is read and its ID and data are stored in a hash table. Locally, it stores in the peer's directory, in a "stored" subdirectory a file with the ID of the file which

contains the file's data, also it stores metadata (its name and replication degree). When a peer stores a copy of the file (the file is only stored if doing so doesn't lead to exceeding the maximum amount of space imposed by a RECLAIM message), it places the data of the file in a different hash table used only for stored files. To do perform the backup a peer sends a message with the following format:
PUTFILE <sender id> <initiator id> <file id> <replication degree> <address to receive file> <port to receive file> <message id>
where the first argument is the ID of the peer that is sending the message, the second is the ID of the peer that initiated the backup process, the third is the ID of the file being replicated, the fourth is the replication degree, the fifth is the address which the message recipient should use to receive a copy of the file, the sixth is the port which the message recipient should use to receive a copy of the file and the last one is the ID of the message itself. After receiving this message a peer connects to the address and port it received and receives a copy of the file, saves it, decrements the replication degree by 1 (if it's greater than 0 then it creates a new similar message to send to the next peer, this next peer will connect to the current peer in order to receive a copy of the file). The backup method can be found in the Peer.java file in lines 165-203.

Restore

A peer that receives the request checks if the file has been backed up, if it hasn't then nothing is done, otherwise it checks which peer the file was initially sent to and sends a message to that peer. To do this it uses a message with the following format:
GETFILE <sender id> <file id> <address to receive file> <port to receive file> <message id>
where the first argument is the ID of the sender, the second is the ID of the desired file, the third is the address which the message recipient should use to receive the file, the fourth is the port which the message recipient should use to receive the file and the last one is the ID of the message itself. The recipient checks if it has the file, if it does then it sends the file data to the address and port it received, otherwise it passes the message along to its successor. The restore method can be found in the Peer.java file in lines 239-291.

Delete

A peer that receives the request checks if the file has been backed up, if it hasn't then it does nothing, otherwise it deletes all copies from all peers by sending a message with the following format:
DELETE <sender id> <file id> <replication degree> <message id>
where the first argument is the ID of the peer sending the message, the second argument is the ID of the file being deleted, the third argument is the replication degree and the last argument is the ID of the message itself. A peer which receives this message deletes its copy of the file (if it has one) and passes the message along to its successor if the replication degree is greater than 0 (to avoid passing by the whole circle). The code related to this protocol is in the Peer.java file in lines 293-321.

Space management

A peer which receives this request checks the amount of data currently stored locally (which is updated when a file is stored), if the limit provided is higher than the amount of space currently being used then it only updates the maximum amount of space that can be used, otherwise it also checks the hash table with the data about stored files and sorts them in descending order by size, and starts deleting them until the space being used drops below the limit, when a file is deleted it sends a message to its successor in order to make another copy of the file to maintain the replication degree (the peer which ends up storing a copy of the file will receive the file's data from the peer which needs to clear space). The code related to this protocol is in the Peer.java file in lines 323-345.

Status retrieval

A peer which receives this request returns a string with the following information: maximum storage size (-1 if there is no limit); current storage in use; data of files processed (backed up), for each one of these it has the name, replication degree and file ID; data of files stored, for each one of these it only has the file ID and the file size. The code related to this protocol can be found in the Peer.java file in lines 476-500.

RMI

The RMI interface is the same that was used for the first project, which looks like this:

```java
public interface RemoteInterface extends Remote {
    String Backup(String filename, int replicationDegree) throws RemoteException;

    boolean Restore(String filename) throws RemoteException;

    boolean Delete(String filename) throws RemoteException;

    void Reclaim(long spaceLeft) throws RemoteException;

    String State() throws RemoteException;
}
```

The interface contains a Backup method which receives as arguments the name of the file that should be backed up and the desired replication degree, this method returns a String indicating what went wrong in case the backup fails. The Restore method receives as an argument the name of the file that should be restored. The Delete method receives as an argument the name of the file that should be deleted. This method and the previous one return booleans indicating if something went wrong or not. The Reclaim method receives as an argument the amount of space the peer (Chord node) is allowed to use (in kilobytes). The State method returns information about the state of a peer, in the form of a String.

Chord

All Chord related messages have the following format:

```
CHORD <action> <args>*
```

where <action> is string that describes the purpose of the message and <args>* is a set of one or more arguments required to fulfill the purpose of the message. Furthermore all the code related to chord is located in the Chord.java and ChordHelper.java source files. Now, all Chord related functions and protocols will be explained.

Lookup

The most relevant functionality of the Chord protocol and the only one that is used by other parts of the project, allows a node (peer) to look for the node that contains an element with a certain ID. To do this, a node that belongs to the Chord circle checks if its successor is the one that contains the desired element, if it isn't then it looks in its finger table for the closest predecessor available and sends a message to that node with the following format:

```
CHORD LOOKUP
<original_sender_address>:<original_sender_port>:<original_sender_id>
<id_to_lookup> <index_waiting>
```

where the first argument contains the address, port and ID of the node that sent the message (so that the node receiving it can reply later); the second argument is the ID that will be looked up and the last one is the index where the reply should be stored in the response queue of the sender. In order to reply to this request, a node must send a message with the following format:

```
CHORD NODE <address>:<port>:<id> <index_waiting>
```

where the first argument contains the address, port and ID of the node where the element with the requested ID is located, the second argument is the index where the node info should be placed in the response queue.

Requesting the predecessor and successor of a node

This is an internal functionality that is necessary in order to ensure that Chord functions properly. Periodically, a node checks who the successor and predecessor of its current successor are: the predecessor of its successor could be its new successor, on the other hand the successor of its successor will become its new successor if the current one dies. In short, the predecessor is needed so that new nodes may join the circle and the successor is required for fault tolerance. The messages used for this functionality are the following:

CHORD REQ_PRED/REQ_SUCC <sender_address>:<sender_port>:<sender_id>
CHORD GET_PRED/GET_SUCC
<pred/succ_address>:<pred/succ_port>:<pred/succ_id>

The “REQ” messages are used to request the successor/predecessor of a node and the argument is the data of the sender just like in the lookup functionality. The “GET” messages are the responses to the requests and the argument is the data of the desired node.

Notify

The last type of messages exchanged as part of the Chord protocol are “notify” messages which are periodically sent to a node’s successor to check if that node’s predecessor is still the same, this is needed to ensure that new nodes can join the circle by alerting their successors of their presence. A node that wishes to “notify” another one sends the following message:

CHORD NOTIFY <sender_address>:<sender_port>:<sender_id>

where the argument is the data of the sender that the recipient needs in order to check if the sender is its new predecessor.

For more information about the chord protocols, check the Chord paper located in the “doc” directory of our project.

Concurrency design

To allow for the concurrent execution of service requests, the project makes use of a pool containing 10 threads. When a request arrives a thread is assigned to process said request. This is done by the UnicastDispatcher class (the source code is in the .java file of the same name).

```java
@Override
public void run() {
    while (true) {
        try {
            SSLSocket socket;
            socket = (SSLSocket) this.serverSocket.accept();
            this.pool.execute(new PreHandler(socket, this.peerId, this.chord, this.localFiles, this.localC
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Furthermore, Java NIO is used for writing asynchronously. This can be seen in the Peer.java file.

```java
public static void write(String path, ByteBuffer byteBuffer, long offset, boolean truncate, int size) throws IOException {
    AsynchronousFileChannel output;
    if (truncate) {
        output = AsynchronousFileChannel.open(Path.of(path), WRITE, TRUNCATE_EXISTING, CREATE);
    } else {
        output = AsynchronousFileChannel.open(Path.of(path), WRITE, CREATE);
    }
    byteBuffer = byteBuffer.limit(size);
    up201806206, 20 hours ago | 1 author (up201806206)
    output.write(byteBuffer, offset, output, new CompletionHandler<Integer, AsynchronousFileChannel>() {
        @Override
        public void completed(Integer result, AsynchronousFileChannel attachment) {
            try {
                attachment.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void failed(Throwable exc, AsynchronousFileChannel attachment) {
            try {
                attachment.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
}
```

JSSE

The project makes use of JSSE for all TCP connections, this is done to ensure security when exchanging data and messages between peers. All JSSE connections are done through SSLSockets. Additionally, we're using truststores and keystores, along with the cipher suites that are supported. To receive data and/or messages the UnicastDispatcher class is used (a separate thread is used to run the code of this class), when the instance is created the socket is setup and then an infinite loop is used for creating the sockets used for communication like is shown below:

```
this.notStoredFiles = notStoredFiles;
try {
    this.serverSocket = (SSLServerSocket) SSLServerSocketFactory.getDefault().createServerSocket(port);
    this.serverSocket.setNeedClientAuth(true);
```

```
@Override
public void run() {
    while (true) {
        try {
            SSLSocket socket;
            socket = (SSLSocket) this.serverSocket.accept();
            this.pool.execute(new PreHandler(socket, this.peerId, this.chord, this.localFiles, this.localC
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

To send data and/or messages the TCPWriter class is used (the source code is in the .java file of the same name), when an instance is created the socket used for communication is setup like shown below:

```
public TCPWriter(String address, int port) {
    this.address = address;
    this.port = port;

    try {
        this.socket = (SSLSocket) SSLSocketFactory.getDefault().createSocket(address, port);
    } catch (IOException e) {
        e.printStackTrace();
    }

}

public TCPWriter(String address, int port, boolean shouldNotCatchExceptions) throws IOException {
    this.address = address;
    this.port = port;

    this.socket = (SSLSocket) SSLSocketFactory.getDefault().createSocket(address, port);
}
```

After that, the "write" methods are used to send the messages/data like shown below:

```java
public void write(byte[] message) {
    try {
        PrintWriter out; // output stream
        BufferedReader in; // input stream
        out = new PrintWriter(this.socket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));

        out.println(new String(message));

        try {
            in.readLine();
        } catch (IOException e) {
        }

        this.socket.shutdownOutput();
        while (in.readLine() != null) ;
        out.close();
        in.close();

        this.socket.close();
    } catch (Exception e) {
    }
}

public void write(byte[] message, boolean shouldThrow) throws IOException {
    this.write(message);
}
```

Scalability

The project offers high scalability through the use of thread pools which were mentioned in the concurrency section. Additionally, we also use Java NIO for asynchronous writes and Chord is also implemented. References to the thread pool and NIO code can be found in the concurrency section, for Chord the source code can be found in the Chord.java and ChordHelper.java files.

Fault tolerance

To improve the fault tolerance of the project we added an extension to the Chord protocol which allows a node to store information about the successor of its current successor in the Chord ring. This way, if the current successor dies, then the successor of the successor becomes the new successor (later, in the stabilize function that runs periodically the new successor will be notified about the change in its predecessor). The way the information is obtained is explained in the

protocols section. Furthermore, with the use of higher replication degrees, file data is stored in more than one peer (Chord node), which means that even if a peer dies, the data is available in other peers. Also, if a peer dies unexpectedly, then the information isn't lost because the metadata is stored locally. If a peer is shut down correctly then it will try to transfer its data to the network (the Chord circle). If it succeeds, its data is deleted, otherwise a local copy is maintained. If a peer recovers then its information is restored from local storage. For the Chord extension the relevant pieces of code are shown below:

```
public void requestSuccessorSuccessor() {
    try {
        TCPWriter writer = new TCPWriter(this.successor.address.address, this.successor.address.port, true);
        String message = "CHORD REQ_SUCC " + this.n.toString();
        writer.write(message.getBytes());
        writer.close();
    } catch (IOException e) {
        if (this.sucessorSuccessor != null) {
            this.successor = this.sucessorSuccessor;
            this.fingerTable[0] = this.successor;
        }
    }

}

public void setSuccessorSuccessor(byte[] message) {
    Node successorSuccessor = this.parseMessage(message, "GET_SUCC");
    if (successorSuccessor != null) this.sucessorSuccessor = successorSuccessor;
}

public void getSuccessor(byte[] message) {
    Node messageSender = this.parseMessage(message, "REQ_SUCC");
    if (messageSender != null) {
        TCPWriter writer = new TCPWriter(messageSender.address.address, messageSender.address.port);
        String response = "CHORD GET_SUCC " + this.successor.toString();
        writer.write(response.getBytes());
        writer.close();
    }
}
```

The first function runs periodically to get the information about the successor of a successor, if it fails to establish connection with its current successor, then the data currently available about the successor of the successor is used to set the new successor. The second function processes the reply to the request made in the first function by storing the data about the successor of the successor. The third function processes the reception of the request about successor data and replies with the requested information. For the replication degree the relevant

code is shown below:

```java
@Override
public String Backup(String filename, int replicationDegree) throws RemoteException, IOException {
    BigInteger fileId = null;
    Path newFilePath = Paths.get(filename);
    BasicFileAttributes attr = null;
    if (Files.exists(newFilePath)) {
        long size = 0;
        try {
            size = Files.size(newFilePath);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            attr = Files.readAttributes(newFilePath, BasicFileAttributes.class);
            fileId = File.getHashedString(filename + "" + attr.lastModifiedTime().toMillis());
            File f = new File(fileId.toString(), String.valueOf(this.peerId), filename, attr.size(), replicationDegree);

            f.saveMetadata();
            if (this.localFiles.containsKey(f.getFileId())) {
                return "File " + filename + " already backed up";
            }
            for (File file : this.localFiles.values()) {
                if (filename.compareTo(file.getFileName()) == 0) {
                    Delete(new BigInteger(file.getFileId()));
                    break;
                }
            }
            this.localFiles.put(f.getFileId(), f);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
    sendFile(filename, replicationDegree, fileId);

    System.out.println("File " + filename + " backed up successfully");
    return "File " + filename + " backed up successfully";
}
```

This is the implementation of the Backup method shown in the RMI interface, which makes sure the file is backed up with the desired replication degree.