FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

SDIS - Distributed Backup Service

Alexandre Almeida de Abreu Filho
up201800168@fe.up.pt
João Castro Pinto
up201806667@fe.up.pt
Iohan Xavier Sardinha Dutra Soares
up201801011@fe.up.pt
José Miguel Afonso Maçães
up201806622@fe.up.pt

Master in Informatics and Computing Engineering

June 2, 2021

Overview

This project implements a peer-to-peer distributed backup system, it supports the following operations

- **Backup** makes a copy of the received file and ensures that a minimum replication degree is met
- **Restore** restores a previously backed up file.
- **Delete** deletes all the backups peers made from a specific file.
- **Reclaim** sets a maximum disk space to be used by the peer and can lead to removing backed up files to not surpass this space.
- **State** retrieves storage information of a peer, including initiated and made backups and disk usage.

The main features of the project are:

- Concurrency Each message handler runs on an independent thread, as well as other functions, which will be described onward.
- SSLEngine Peers communicate using the SSLEngine class from the JSSE module.
- Scalability The backup service was implemented using a CHORD peer to peer schema.
- Fault Tolerance With the help of CHORD, the program handles a crashes the same way as shutdowns.
- **Storage Save** When a peer process ends, the relevant peer information is stored in non-volatile memory, so that the information can be loaded when the peer reboots.

1 Protocols

All information send using this service is transported via SSL/TCP using SSLEngine where each connection is established and closed with a handshake, until it's closed the two sides can send and receive messages. The connections with clients are also made using the same protocols.

There are three types of messages and a total of seventeen different messages:

- CHORD messages
  - GET_POSITION «peer_key»;
  - GET_PREDECESSOR;
  - GET_SUCCESSOR;
  - PEER_ADDRESS «key» «address»;
  - SET_PREDECESSOR «key» «address»;
  - SET_SUCCESSOR «key» «address»;
- Client requests
  - BACKUP «file_path» «replication_degree>;

  - DELETE «file_path»;
  - RECLAIM «max_space»;
  - RESTORE «file_path»;
  - STATE;

- Peer requests
  - CONTENT «sender_address» «file_id» «file_content»;
  - DELETE «sender_address» «file_id»;
  - GET_CONTENT «sender_address» «file_id»;
  - REMOVED «sender_address» «file_id»;
  - STORE «sender_address» «file_id» «replication_degree» «file_content»;
  - STORED_FILE «sender_address» «file_id»;

The storage of the peers is in the folder ./data/«port_of_the_peer_access», it's used to save the saved files, logs and storage saves.

1.1 CHORD Protocols

When a peer receives one of the messages GET_POSITION, GET_PREDECESSOR or GET_SUCCESSOR it responds with a PEER_ADDRESS message, with the corresponding address. The response of get position is the address of the peer with the key or the successor of it if it does not exist.

The SET_PREDECESSOR and SET_SUCCESSOR messages do not have responses, and they exist to notify other peers of the sender existence.

1.2 Backup Protocol

As it was previously described, the backup sub-protocol makes a copy of the received file and ensures that a minimum replication degrees met. The following code handles the initiator peer's running of the backup sub-protocol. It sends the message "STORE" to the other peers asking to store a backup from a file. If the replication is not met, it sends the message again, and after the processes ends, it sends a message to the client.

```
public void run() {
    if (messageToSend == null) {
        // Create File object based on the message
        // Create new StoreMessage
    } else {
        // Get file from Storage
    }

    // Create message

    for (int i = 0; !stop && i < 10; i++) { // Try a maximum of 10 times
        try {
            // Send message
            Thread.sleep(100);
        } catch (Exception exception) {
            // Create exception message
            stop();
        }
```

```
    }

    if (stop) {
        // Add file to Storage
    } else {
        // Create new message "Could not backup file with desired replication degree";
    }

    if (messageToClient == null)
            messageToClient = new SSLMessage("Success");

        try {
            if (connection != null) {
                Server.getInstance().write(connection, messageToClient);
            }
        } catch (IOException exception) {
            exception.printStackTrace();
        }
}
```

From class BackupAction from line 28 to line 69 (skipping prints)

1.3 Restore Protocol

As it was previously described, the restore sub-protocol restores a previously backed up file. The following code handles the initiator peer's running of the restore sub-protocol. It sends the message "RESTORE" to the other peers asking to restore a file from a backup. It sends a message to every peer it knows, and if one of them a has the backup the initiator peer seeks, it will send the content of the backup file to the initiator peer.

```
public void run() {
    // Create File object based on the message
    // Create new RestoreMessage

    while (!stop) {
        try {
            // Send message to all peers
            Thread.sleep(100);
        } catch (Exception exception) {
            stop();
            return;
        }
    }

    try {
        // Restore the file
        // Send the message to the client
        Server.getInstance().write(connection, new SSLMessage("Success"));
    } catch (IOException exception) {
        exception.printStackTrace();
        try {
            Server.getInstance().write(connection, new SSLMessage("Could not restore
                file"));
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

From class RestoreAction from line 23 to line 53 (skipping prints)

1.4 Delete Protocol

As it was previously described, the delete sub-protocol deletes all the backups made from a specific file. The following code handles the initiator peer's running of the delete sub-protocol. It sends the message "DELETE" to the other peers asking to delete a backup from a file. Every peer that receives this message deletes the backup it has from this file. There is no response to the "DELETE" message.

```
public void run() {
    // Create File object based on the message
    if (file != null) {
        // Create Delete message
        for (InetSocketAddress key : file.getPeersWithCopy()) {
            try {
                // Send message to peers with a copy of the file
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } else {
        messageToClient = new SSLMessage("Could not find file, did you make a backup
            of this file using this address before?");
    }

    stop();
    if (messageToClient == null)
            messageToClient = new SSLMessage("Success");

        try {
            if (connection != null) {
                Server.getInstance().write(connection, messageToClient);
            }
        } catch (IOException exception) {
            exception.printStackTrace();
        }
}
```

From class DeleteAction from line 20 to line 48 (skipping prints)

1.5 Reclaim Protocol

As it was previously described, the reclaim sub-protocol sets a maximum disk space to be used by the peer and can lead to removing backed up files to not surpass this space. The following code handles the initiator peer's running of the reclaim sub-protocol. It sends the message "RECLAIM" to the other peers asking to reclaim disk space from a peer. Whenever a backed up file is deleted in this protocol, the peer sends a "REMOVED" message to the initiator peer, so that it can try to keep the desired replication degree.

```
public void run() {
    // Get maximum disk space
    // Get storage size

    while (storage.getStoredSize() > maxDiskSpace) {
        for (File file : storage.getFiles().values()) {
            // Get largest file
        }
        if (largestFile != null) {
            // Delete largest file and remove it from Storage

            // Create Removed message
            try {
                // Send message
                server.send(largestFile.getInitiatorPeer(), removedFileMessage);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    // Set new maximum disk space
    Server.getStorage().setMaxDiskSpace(maxDiskSpace);

    stop();

    SSLMessage messageToClient = new SSLMessage("Success");

    try {
        Server.getInstance().write(connection, messageToClient);
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

From class ReclaimProtocol from line 20 to line 57 (skipping prints)

1.6 State Protocol

As it was previously described, the state sub-protocol retrieves storage information of a peer, including initiated and made backups and disk usage. The following code handles the initiator peer's running of the state sub-protocol. A client sends the message "STATE" to a peer in order to retrieve its information. This protocol was very useful during the development of the service.

```
public void run() {
    // Generate State string and create message
    SSLMessage messageToSend = new SSLMessage(getStateString());
    try {
        // Write message
        Server.getInstance().write(connection, messageToSend);
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

From class StateAction from line 70 to line 77

2 Concurrency design

In the implementation of the concurrency design of this project, thread pools were used, in order for it to be possible to execute multiple protocols at the same time. Anytime a message from the client or from a peer is received, the receiving peer creates a thread to deal with that message, performing the desired action.

The server's run method calls the respond() method when it receives a message. That method will check what type of message it is receiving: if it is a message that represents a request from a client or peer, it calls the processMessage() method. This method will then create the Service to be executed from the message and call the ThreadPoolExecutor's execute() method with that Service, creating the thread.

```
class Server {
    // rest of class implementation

    private SSLMessage processMessage() {
        // Check if it is a request from a Peer or from a Client
        // Create service
        // Call execute()
    }

    public static void execute(service) {
        // Adds service to servicesRunning list
        instance.executor.execute(service);
    }

    public void respond(message) {
        // Parse message
        // Call processMessage if message is a request from a Peer or Client
        // Send response
    }

    // rest of class implementation
}
```

Class Server from line 37

Furthermore, Java NIO was used in this project for all the SSL implementation. NIO means Non-blocking I/O, which concretely means the input and output solutions offered by NIO do not block, this is very useful when being accessed by more than one thread.

Finally, the more basic read and write functions for SSL are also surrounded by a mutex, preventing messages from being lost or mixed up and disallows two threads to send messages at the same time.

```
class SSLPeer{}
    // rest of class implementation

    public final SSLMessage read() {
        synchronized (mutex) {
            // Read message

            // Deal with SSLEngine
            }
        }
```

```
    }

    public final boolean write() {
        boolean success = false;
        synchronized (mutex) {
            synchronized (mutex) {
            // Write message

            // Deal with SSLEngine
            }
        }
    }

    // rest of class implementation
}
```

Class SSLPeer from line 43

3 JSSE

The project uses SSLEngine for the communication between processes, its functions for wrap and unwrap are used in the ssl.data.SSLStream. These are some examples of code related to JSSE:

3.1 Creation in the client

We can see the creation of the SSLEngine in the client using this example taken from line 16 of ssl.application.SSLClient.

```
public SSLClient(InetSocketAddress hostAddress) {
    // beginning of constructor
    engine = context.createSSLEngine(hostAddress.getHostName(), hostAddress.getPort());
    engine.setUseClientMode(true);
    // rest of constructor
}
```

Class SSLClient from line 16

3.2 Accepting a connection

The run method of the server exemplifies how it receives a connection request and reads from a client.

```
public void run() {
    // Starts server
    while (active) {
        // get selected keys
        while (selectedKeys.hasNext()) {
            SelectionKey key = selectedKeys.next();
            // other key processing
            if (key.isAcceptable()) {
                // accepts key
            } else if (key.isReadable()) {
                // creates a new connection from key
```

```
                // reads message
            }
        }
    }
}
```

Class SSLServer from line 43

4 Scalability

This section is mandatory only if your service includes scalability provisions. In this section you should describe how your design or implementation contribute to the scalability of your service implementation. For any of the suggested features in the assignment description, or additional features, that you have adopted, you must describe how it is implemented/used and why. Again, you must include references to the relevant code.

In this project, a CHORD design was implemented, which means that scalability is accounted for: the cost of a CHORD lookup grows as the log of the number of nodes, which means large systems are feasible.

```
class CHORDPeer{
    // rest of class implementation

    public PeerAddress findResponsible(int key) {
        // Get successor
        do {
            // Ask for the position of key
            // Try 10 times if nothing is received
            // If received what I wanted, break
            // Get successors predecessor
            // If predecessor is smaller than what I wanted, then this is the
                successor, break;
            // Get next successor
        } while (true);

        return successor;
    }
    // rest of class implementation
}
```

Class CHORDPeer from line 406 - CHORD lookup operation

At the implementation level, thread pools are used, and naturally they improve scalability as they yield concurrency and parallelism, as discussed in the second section.

5 Fault-tolerance

As CHORD is being used, all its fault tolerance features were implemented, namely periodic stabilization, to prevent failure when a peer stops working or leaves the ring.

When a peer joins the ring, periodic calls to a stabilization function are scheduled, using a ScheduledThreadPoolExecutor (). This function works through 4 simple steps: first, the peer tries to reach its successor by sending a message. If its successor is reached, it does nothing. If its successor cannot be reached, the peer goes through its following successors, finding the

nearest alive successor, asking for its predecessor. If that predecessor is between the peer and its nearest alive successor, the peer sets that predecessor as its successor. After that, the peer notifies its successor and its predecessor of its existence, so they can update their predecessor and successor accordingly. Lastly, the peer updates its finger table.

Moreover, the peer tries to connect with all the peers it knows, if it's not able to connect to it, it's removed from the known peers and the fixFingers deals with that afterwards.

```
class CHORDPeer{
    // rest of class implementation
    protected void start() throws Exception {
        // rest of method implementation
        // Enter the ring
        // Schedule calls to stabilize() method every 20 seconds
        // rest of method implementation
    }

    public void stabilize() throws Exception {
        // Gets its successor
        // Sends message to successor, if not found, sends to next successors until
            finding a live one
        // If the peers successors predecessor is between the peer and the peers
            successor,
        // then it should be the peers successor
        notifySuccessors(); // Notifies its successors
        fixFingers(); // Updates its finger table
    }
    // rest of class implementation
}
```

Class CHORDPeer from line 357