U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 2 - Distributed Backup Service

Final Report

SDIS 2020/2021 - 2nd Semester

3MIEIC05 - Grupo 24

Ivo Saavedra	**up201707093@edu.fe.up.pt**
Telmo Baptista	**up201806554@edu.fe.up.pt**
Tiago Duarte da Silva	**up201806516@edu.fe.up.pt**

1. Overview

Our project supports all operations: backup, delete, reclaim, restore and state. Our implementation relies on Chord with full Java NIO Selectors, ThreadPoolExecutors and addresses both scalability and fault-tolerance.

1.1 Implementation details

Each peer in the network is identified by the hash of its IP address and port, following chord's specification which relies on consistent hashing to distribute nodes through the network.

The storage distribution of files within the chord ring is done according to a file's key. This key is obtained by hashing the file's hash using the same algorithm as the one for obtaining a peer's id. This makes it so that every peer will own a set of keys (the ones between its predecessor and itself) and, for each key, a set of files that node is responsible for. We will reference this as the file or key's owner/holder throughout this report.

Given that the same hashing used for peers is used for the files, we can assume it will be consistent and average out the storage load evenly between each Peer (given the standard deviation for a file's size is not too significant).

Our implementation supports 2 types of file ownership. Either a universal network where anyone can reclaim or delete any file or another where each file can only be reclaimed or deleted by the initiator peer. This is accomplished by the inclusion (or not) of the initiator peer's id in the hashing process. The version included in the delivered version is the second one.

The use of threads in our implementation is restricted to chord nodes' maintenance and to the state backup, as the use of Java NIO selectors allowed the management of each server socket without recurring to threads. All operations relative to Chord's maintenance, such as the stabilization of the ring and the revision of the finger table are done periodically by a SingleThreadExecutorService with a fixed delay. The backup service follows a similar logic, with the addition of a dirty bit to mark if any changes occured.

2. Protocols

For the Chord implementation RMI was used, with every Peer extending a Node class that implemented the INode interface, to force the implementation of the required methods for chord to be successful.

All communication related to the operations supported and the protocols created is done through TCP sockets. All messages implement the ISocketMessage interface and are prepended with the respective protocol and the sender's id. When sent, a terminator that consists of [CR, LF, CR, LF] is appended. Below we specify all the implemented messages and their respective format.

- Backup, Replicate

<SENDER_IP> <SENDER_PORT> <FILEHASH> <REP_DEGREE> <FILE_SIZE>

- DelKey, DelCopy, GetFile, Removed
  <SENDER_IP> <SENDER_PORT> <FILEHASH>

- FileHere
  <FILE_AT_ID> <FILEHASH> <SIZE>

- ReplicationLostMessage
  <PROTOCOL> <SENDER_ID> <ID_LOST> <FILEHASH>

- FileExists, Replicated
  <FILEHASH>

- Ack
  <STATUS>

- State
  <SENDER_IP, SENDER_PORT>

All protocols that send the respective message are considered trivial: backup uses **Backup**, delete uses **DelKey** and restore uses **GetFile**. The **Reclaim** protocol is the sole exception to this rule.

All our messages are all context-sensitive, as such, they can take different meanings and trigger different responses. Below are all specified all possible message chains[1] in our implementation.

2.1. Backup

This message is sent from an initiator peer wanting to backup a file to the peer responsible for that file.

Upon receiving this message, the file owner will check if there is already a file with the indicated hash. If so, it will start by triggering a **DelKey** message for the respective file. Afterwards, the file will be added to the respective key and it'll be stored. The key owner will do so by reading directly from the same socket, as after the **Backup** message an initiator peer will always send the file's contents.

Finally, after having stored the file, the key owner will initiate a **Replicate** message to ensure the initiator peer's desired replication degree is achieved.

There is, however, one unimplemented case when the responsible peer does not have enough storage to store the file. Nevertheless, this case is still contemplated within our specification: the file owner would create a special connection with a successor that could

[1] Due to the use of the Java NIO Selector, we need to create multiple consecutive *ISocketManager*s to handle the logic of sequential flow of events such as "after receiving X, send Y", "after sending X, read Y" or simply to handle a message and specify the next course of action. This will be abstracted throughout this entire section, to be later explained in Concurrency design & Scalability

store the file, through means similar to the **Replicate** message, in terms of propagation through the network. Then the file owner would receive the file from the initiator and directly redirect it to the peer that replied to the message.

2.2. Replicate

This message is sent from a peer that stores a backup whose desired replication degree was not yet achieved to its successor in order to replicate the file. This message is then propagated through each successor until it reaches the desired replication degree.

This message stops being forwarded when either the replication degree reaches 0 or when the message loops back around to the original node.

The desired replication degree is diminished by one whenever the node can store the file. In this case, the node will initiate a pseudo-restore protocol using **GetFile** and **FileHere** messages to store a copy in itself. On successful replication it will then inform the file key's owner via a **Replicated** message.

2.3. Replicated

This message is sent when a peer finishes receiving and storing a file sent by another peer after receiving a **Replicate** message. This is done so the owner of the file key can keep track of where the copies of the file are kept. This message requires no confirmation response.

2.4. DelKey

This message is sent whenever a client wishes to delete a file from storage. It is aimed at the file owner in order to remove the replicated copies in other peers. The client connects to a given peer and initiates the deletion protocol. If this peer is the one responsible for the file that's being deleted, the message is not sent, as the peer already knows where the file copies are located.

Upon receiving this message, the file owner peer starts the **deleteFileCopies** protocol, whose job is to contact other peers who have a local copy of the given file with a **DelCopy** message. This message requires no confirmation response.

2.5. DelCopy

This message is received by any peer who has a local copy of a file that is to be deleted after a delete request by the client. Upon receiving this message the peer removes from its storage the file with the corresponding hash received in the message header. This message warrants no response.

2.6. Removed

This message is sent when a peer removes a file from its storage due to lack of capacity and needs to notify the network of its removal to maintain the consistency of information and initiate replication protocol if it goes below the desired replication degree.

2.7. GetFile

This message is sent whenever a peer wants to retrieve a file from the network. There are two possible replies, a negative **Ack** message because we don't either store the file or are the peer responsible for it. The other possible response is a **FileHere** message containing the id of a peer storing the file. If the returned id is our own, then we will immediately send that file's contents over the connection afterwards.2.8. State

In order to assure the preservation of the keys each peer is responsible for, we backup its state in its first successor. This way, whenever the key holder crashes, its successor detects this failure and assumes control of the keys that were kept in the peer that just crashed.

In order to accomplish this we created a new message that is sent to a key holder's direct successor whenever any operation alters the keys for which it is responsible.

Upon receiving this message, the target peer prepares the for reception of the PeerInfo sent by its successor.

2.9. FileExists

This message is sent whenever a peer wants to know if another peer has a given file. After sending the request to the target peer, the sender waits for an acknowledgment. Although this message is fully implemented, it is not being currently used in any protocol or error recovery.

2.10. ReplicationLostMessage

This message is sent whenever a peer detects that its successor died and while processing its latest state finds a file it used to store. For each file, a **ReplicationLost** message is sent to the respective file owner.

Upon reception of this message, the file owner removes that copy from its list and, in case the replication dropped below the desired level, sends a **Replicate** message to restore the desired replication.

3. Concurrency design & Scalability

The usage of Java NIO on all communication protocols is a major component of our concurrency design and greatly contributes to the scalability of the application, by allowing the server to tend to multiple clients without significant delays.

In order to implement NIO, the *Selector* class of Java NIO needed to be used to be able to tend to all connections without needing additional threads to listen to the multiple channels. Therefore, we only have a single thread that manages all existing channels connected. According to our research, using multiple threads for this selector would not improve our performance, as the limits of the *Selector* would still apply, just distributed between each *Thread*.

We created the ServerSocketHandler to deal directly with the selector. This Runnable registers the peer's ServerSocket and then is forever in a while loop waiting for the blocking call *'selector.select()'* to return, meaning a new set of keys is available.

To deal with all the possible connections, we implemented an interface to manage all the possible actions, called ISocketManager. This interface is used to specify the action to be executed once a socket has been selected through these following methods.

```
void onSelect(SelectionKey key);
void init() throws IOException;
int interestOps();
```

This interface also defines the `static void transitionTo(SelectionKey key, ISocketManager manager)` method, which transitions from the current ISocketManager to the next, calling init on the manager (to initialize file buffers, for example) and to set the new set of interestOps with the selector.

Anytime we expect to receive a message from the socket we use a SocketManager. This class accepts an ISocketManagerDispatcher in the constructor. This interface only forces the implementation of the `ISocketManager dispatch(ISocketMessage message, SelectionKey key)` method, which accepts a fully parsed message and returns a new ISocketManager to be responsible for the next step of the process. When the value returned is null, it represents that the connection should be closed and no further operations are expected on said socket. The SocketMessage class also has a constructor receiving a Peer that loads the DefaultSocketManagerDispatcher, which handles the very first message received from a new server socket connection.

Other dispatchers were implemented to handle more specific scenarios, either inline or using other classes, such as AckNackDispatcher, which receives two callbacks for success and failure, ReplicateDispatcher, to handle file replication and RestoreDispatcher, to handle file restoration.. The latter also include other methods to encapsulate and initiate some processes.

Given the asynchronous nature of NIO, the connection to a ServerSocket was also non-blocking, as such, our implementation to handle this was inspired by Futures. Nearly every time we initiate a connection we need to send a message header first, for that motive, the DefaultSocketManagerDispatcher has an overloaded constructor that also accepts a Supplier<ISocketManager>. We use this to run an anonymous function as soon as the connection is established.

As the logic behind every operation is very simple (sending a reply, opening a new socket, setting some values in the peer), our implementation scales very well due to how the Selector can handle multiple connections. This means that the thread will only execute an operation when needed, without the overhead of a context switch between threads or the need for more blocking calls. As such, the thread context switches become reserved for periodic tasks, such as the maintenance of chord's stability.

The only blocking methods used were Monitors (see GeneralMonitor). They were only required for methods where we should block until an outcome is known (or timeout). As

such, they are only found in the interface with the client, in the backup, delete, reclaim and restore protocols. This ensures that we did not make any sacrifice or concession in the speed and scalability of our project.

Whenever this monitor times out due to some unknown reason, we return 'timeout' to the client. Otherwise, 'success' or 'failure' is returned.

4. JSSE

JSSE provides a service that guarantees a higher degree of security and trust when communicating via TCP.

In our project we were unable to integrate our system with JSSE SSLEngine, but we had an implementation of an SSLPeer that acts as a base for a future implementation of a Peer with SSLEngine. The peer has four buffers, two to store its own application data and network data (*myAppData* for application data and *myNetData* for encrypted data), and two to store other peer's data (*peerAppData* for application data and *peerNetData* for encrypted data). The abstract class also contains the function to do the *handshake* protocol between the peers handling possible overflows and underflows as well as other situations described by the *SSLEngineResult* status or all the possible *HandshakeStatus*.

The server specific peer is then implemented on *SSLServer* where the keystore and truststore are initialized, as well as the buffers and the *IP:port* are binded to the server socket and registered to the *Selector*.

Server constructor

```
char[] password = "123456".toCharArray();
KeyStore keyStore = KeyStore.getInstance("PKCS12");
keyStore.load(new FileInputStream("./keys/server.keys"), password);
// Load trustStore
KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
kmf.init(keyStore, password);
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(trustStore);
this.context = SSLContext.getInstance("TLS");
this.context.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
SSLSession session = this.context.createSSLEngine().getSession();
this.myAppData =
ByteBuffer.allocate(session.getApplicationBufferSize());
this.myNetData = ByteBuffer.allocate(session.getPacketBufferSize());
// Other buffers
session.invalidate();
this.selector = SelectorProvider.provider().openSelector();
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.configureBlocking(false);
serverSocketChannel.socket().bind(new InetSocketAddress(host, port));
serverSocketChannel.register(this.selector, SelectionKey.OP_ACCEPT);
```

It also includes functions to start the server, accept a connection, shutdown the server and read/write from a socket. To start the server, we start the *Selector* loop to accept connections which can be done via the function *start*.

```
while(this.isActive()) {
    This.selector.select
    Iterator<SelectionKey> selectedKeys =
this.selector.selectedKeys().iterator();
    while(selectedKeys.hasNext()) {
        SelectionKey key = selectedKeys.next
        selectedKeys.remove();
        if (key.isValid()) {
            if (key.isAcceptable()) this.accept(key);
            else if (key.isReadable
                this.read((SocketChannel)key.channel(),
(SSLEngine)key.attachment());
        }
    }
}
```

In order to accept a connection we need to configure the *SocketChannel* as non-blocking, create the *SSLEngine* via the context and set all the configurations such as disable client mode, set the cipher suites, set the protocols so that we can start the handshake and register the socket if the handshake was successful. This is done by calling the function *accept*.

```
SocketChannel socketChannel =
((ServerSocketChannel)key.channel()).accept();
socketChannel.configureBlocking(false);
SSLEngine engine = this.context.createSSLEngine();
engine.setUseClientMode(false);
engine.setEnabledCipherSuites(engine.getSupportedCipherSuites());
engine.setEnabledProtocols(new String[] {"TLSv1.3"});
engine.beginHandshake();
if (this.handshake(socketChannel, engine))
    socketChannel.register(this.selector, SelectionKey.OP_READ, engine);
```

Finally, the server implements the *read* and *write* that do the *unwrap* and *wrap* of the data, respectively, to the correct buffers.

```
int bytesRead = socketChannel.read(this.peerNetData);
if (bytesRead > 0) {
    this.peerNetData.flip();
    while(this.peerNetData.hasRemaining()) {
        this.peerAppData.clear();
        SSLEngineResult result = engine.unwrap(this.peerNetData,
this.peerAppData);
```

```
        switch(result.getStatus()) {
        // Handle statuses
        }
    }
} else if (bytesRead < 0) { // Handle EOF }
```

```
this.myAppData.put(message.getBytes());
this.myAppData.flip();
while(this.myAppData.hasRemaining()) {
    this.myNetData.clear();
    SSLEngineResult result = engine.wrap(this.myAppData,
this.myNetData);
    switch(result.getStatus()) {
    case OK:
        this.myNetData.compact();
        while(this.myNetData.hasRemaining()) {
            socketChannel.write(this.myNetData);
        }
        break;
    // Handle others
}
```

The client peer functions in similar form to the server, it only differs by switching the *SSLEngine* to client mode and using the method *connect* to connect to a server instead of *accept*.

```
this.socketChannel = SocketChannel.open();
this.socketChannel.configureBlocking(false);
this.socketChannel.connect(new InetSocketAddress(this.remoteAddr,
this.port));
while(!this.socketChannel.finishConnect()) {}
this.sslEngine.beginHandshake();
this.handshake(this.socketChannel, this.sslEngine);
```

5. Fault-tolerance

The purpose of fault tolerance is to prevent and avoid total or partial system failures and ensure the highest levels of reliability and availability.

One of the mechanisms implemented is saving the peer's state. This is done by executing a state backup task regularly. This task sends a *State* message to store its state on its successor within the chord ring.

State backup recurrent task

```
ScheduledExecutorService stateBackup =
Executors.newSingleThreadScheduledExecutor();
stateBackup.scheduleWithFixedDelay(() -> {
   if (this.dirtyState) {
       this.backupState();
   }
}, 15000, 10000, TimeUnit.MILLISECONDS);
```

Whenever a peer crashes, its successor becomes in charge of its keys and all requests regarding those keys will be forwarded to him (following chord's logic). If the successor detects a drop in the replication of the keys it has inherited from its predecessor it will then start a replication protocol to ensure that each file's replication degree is maintained. This operation is handled in the **on_predecessor_death** method.

To remove a single point of failure, if a user requests a replication degree of 1, it will be increased to 2. This ensures that if the single peer that stored the file dies, the chord ring will be able to recover the backed up file from it.

.