Distributed Backup Service for the Internet - 2nd Project Report

Integrated Master in Informatics and Computing Engineering

Distributed Systems

João Carlos Machado Rocha Pires - **201806079**

João Ricardo Ribeiro Cardoso - **201705149**

Miguel Maio Romariz - **201708809**

Nuno Miguel Fernandes Marques - **201708997**

Class 2, Group 22

June 2, 2021

Contents

1 Overview

Our second project is the follow-up of the first project, but this time using the Internet for the distributed peer-to-peer backup system.

We've used the code developed by Nuno and Miguel as the basis for the second project, as it was more object oriented.

That said, our code coverage includes the protocols (described in the next section) used in the first project: backup, restore, reclaim and delete.

Our solution to achieve fault-tolerance is similar to the one presented in the following picture, except queries and replies are done between Peers, while Trackers simply assist with Peer discovery:

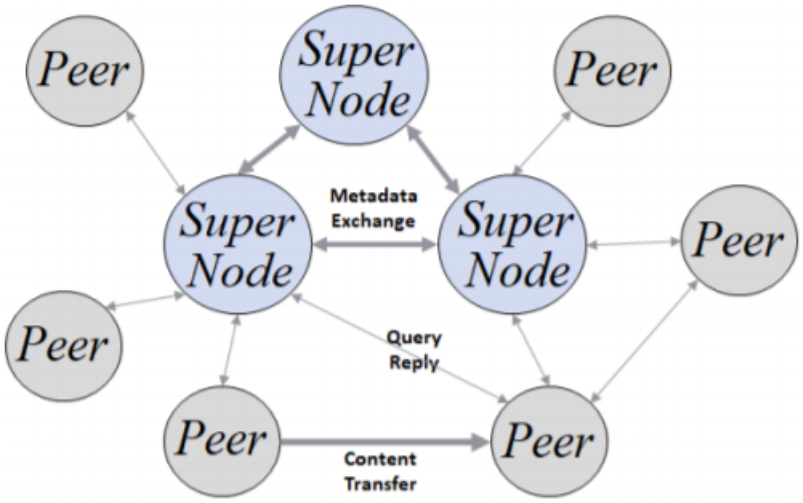

Figure 1: Schema of a distributed peer-to-peer network with super nodes

Note: We'll use the term Tracker to mention the Super Node in the above picture.

We're using a structure similar to the Torrent system, i.e. with Trackers whose responsibility is to help Peers connect within each other by providing Peer discovery, allowing them to have full control over the communication once it's fully established. Also, the Trackers share their internal state between them, which includes the information regarding Peers and all the other Trackers within the system.

To achieve fault tolerance and concurrency, Java NIO, Thread Pools, SSL Sockets and a semi-centralized architecture were used, which will be described in detail in the next sections.

2 Protocols

The Peer protocols were unchanged from the first project, meaning that we still use RMI as a remote interface of the Peers and backup files using chunks. All communication is done with message objects using object streams. The streams use TCP with SSL sockets. Each message object has at least three fields: protocol version, message type and sender id. Depending on the message type, the message may include more fields.

2.1 Backup

To backup a chunk, the initiator-peer sends all other Peers a TCP message of type "BACKUP" with a body that contains the contents of that chunk. This message also includes the sender ID, the chunk number and the desired replication degree.

A Peer that stores the chunk upon receiving the previous message should reply by sending a confirmation message of type "STORED" with the file ID and chunk number.

2.2 Restore

To recover a file, the initiator-peer shall send a message of type "GETCHUNK" to all other peers requesting each chunk of that specific file. This message includes the file ID and the chunk number.

Upon receiving this message, a Peer that has a copy of the specified chunk shall send it in the body of a message of type "CHUNK" directly to the initiator-peer with the file ID, chunk number and chunk body. If another Peer replies to the "GETCHUNK" first, no "CHUNK" message will be sent.

2.3 Delete

In order to allow the deletion of all chunks of a file from the backup service, the protocol sends a "DELETE" message with the filed ID.

Upon receiving this message, a Peer will remove all chunks belonging to the specified file. No reply is needed.

2.4 Reclaim

Peers can choose to reclaim space allocated to stored chunks (backup of other's chunks/files). If space is reclaimed, the result will be a chunk deletion and then all other Peers will be notified using a "REMOVED" message with the file ID and chunk number.

After the reclaim, if the actual replication degree of a chunk is less than the desired one, the Backup protocol is immediately started so that the replication degree can be satisfied.

2.5 Other messages

- REGISTER

  This new type of message is sent whenever a Peer enters the system. It contains the port number of the new Peer and it's sent to a Tracker, which depends on the one in which the Peer is being registered.

- NEWPEER

  Upon receiving the previous message, the Tracker will then update all the other Trackers, as well as the Peers, with a "NEWPEER" message, including the Peer's address and port number.

  Also, the new Peer receives a "NEWPEER" message for each peer already in the system.

- ALIVE / OK

  To both check the aliveness of Trackers and Peers, we've created a new set of messages: "ALIVE" and "OK". Each second, a Tracker sends an "ALIVE" message to all the Peers and waits for period of time for the "OK" message as a response. The same occurs while checking for the aliveness of other Trackers, as they exchange between them "ALIVE" and "OK" messages to check whether or not they're available.

- REMPEER

  If the "OK" message from the Peers, previously mentioned, wasn't received within a time limit of 2 seconds, the Tracker will send a message of type "REMPEER" to all Trackers and Peers to inform the nodes they should remove the Peer which is no longer available/online.

- TREGISTER

  This new type of message is sent whenever a Tracker enters the system. It contains the IP address and port number of the new Tracker and it's sent to all the other Trackers of the system.

  Also, this new Tracker receives a message of this type for each other Tracker in the system.

The following list includes the specific Java files that are responsible of creating, sending and handling messages:

- Message.java
- TrackerDispatcher.java
- Dispatcher.java

Peer message dispatcher

```
switch(this.msg.getMessageType()) {
    case "PUTCHUNK":{
        this.putChunkHandler();
        break;
    }
    case "STORED":{
        this.storedHandler();
        break;
    }
    case "DELETE":{
        this.deleteHandler();
        break;
    }
    case "CHUNK":{
        this.chunkHandler();
        break;
    }
    case "GETCHUNK":{
        this.getChunkHandler();
        break;
    }
    case "REMOVED":{
        this.removedHandler();
        break;
    }
    case "NEWPEER":{
        this.newPeerHandler();
        break;
    }
    case "ALIVE":{
        this.aliveHandler();
        break;
    }
    case "REMPEER":{
        this.removePeerHandler();
        break;
    }
    default:{
        System.out.println("Unknown message type");
        break;
    }
}
```

Note: The above code is a simplified version of the one used and that was just made for report organization purposes.

3 Concurrency Design

Each node contains a thread pool used to run various tasks, such as our SSL handler thread which is responsible for accepting messages and starting a new dispatcher thread to process each message. Tracker nodes also run two threads at a scheduled fixed rate to check whether or not other nodes are still connected: one to check for tracker nodes and another for peer nodes. Concurrent hash maps and synchronized functions are used for thread sensitive data. File operations use the Java NIO API to avoid concurrency problems.

SSLHandler.java

```
public class SSLHandler implements Runnable {
    protected ConcurrentHashMap<...> trackers;
    protected ConcurrentHashMap<...> peers;
    protected ConcurrentHashMap<...> timestamps;
    @Override
    public void run() {
        while(true) {
            try {
                SSLSocket socket = (SSLSocket)
                    this.sslSocket.accept();
                ...
                node.handleMessage(msg, (InetSocketAddress)
                    socket.getRemoteSocketAddress());
                ...
            } catch (Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

Peer.java

```
public class Peer extends Node implements InitiatorPeer {
    public void handleMessage(Message msg, InetSocketAddress
        remoteAddress) {
        this.threadPool.execute(new Dispatcher(this, msg,
            remoteAddress));
    }
}
```

4 JSSE

In our implementation we use SSL Sockets in every protocol to make sure the communication is secure.

Our SSL Socket use requires client authentication and the cipher-suites used are the ones supported (i.e. the ones returned after calling the method *getSupportedCipherSuites*).

The Java file responsible for this part is the SSLHandler.java.

5 Scalability

To create a scalable implementation, we used a thread pool to perform multiple tasks at the same time.

Each node runs a thread to receive messages. That thread also creates, upon receiving a message, a new thread to process that message. This allows a node to process multiple messages at the same time.

Trackers also run two threads with a fixed delay to check the aliveness of other Trackers and Peers, as described in the protocols section.

We also made use of the Java NIO API to avoid blocking while managing files, which would cause concurrency issues and concurrent hash maps and synchronized methods to avoid concurrency problems when accessing and modifying internal data in different threads, like the maps of Peers and Trackers addresses and ports.

The use of multiple Trackers also allows the task division among Trackers, specially the communication with peers. This contributes to a scalable "network" of Peers and Trackers.

The code examples provided in the Concurrency Design section can also be applied here. Also, some files worth of mentioning where Java NIO is used are: LocalFileInfo.java, State.java and Dispatcher.java.

6 Fault-Tolerance

In order to have fault tolerance and avoid a single point of failure, we allow the use of multiple interchangeable Trackers. This avoids the failure of the peer discovery task in case a Tracker becomes unavailable.

Also, if all the Trackers are disconnected at any given point, the Peers that were previously connected may continue to operate normally.

Since fault tolerance was achieved by the correct design decision, we don't have relevant code to add here as all the code is relevant to this goal.