Final Report

Sistemas Distribuídos
2020/2021

Mestrado Integrado em Engenharia Informática e Computação

Peer-to-peer backup service for the Internet

Beatriz Costa Silva Mendes- up201806551
Carlos Miguel Sousa Vieira- up201606868
José Pedro Nogueira Rodrigues up201708806
Mariana Oliveira Ramos- up201806869

Professor Hélder Fernandes Castro

Index

1. Overview

The main goal of the report is to describe the design and implementation of the developed peer-to-peer distributed service for the Internet, using the free disk space of the computers on the Internet for backing up files onone's own computer.

Our service supports all the required operations: **backup**, **restore** and **deletion** of files, as well as the **reclaim** of the space used by a peer. Additionally we implemented a **state** operation for each peer as well as for the Main Peer. Each file is sent in messages of 10KB that are merged before storing the file.

To satisfy the security requirements as well as scalability of our system, the following features were also implemented:

- Security was assured by encrypting traffic between peers, making use of Java SE's JSSE package through the **SSLSockets** API.
- Scalability was assured at implementation level by using **thread-pools** and **Java NIO** as an asynchronous I/O.
- Reliability was assured on an application level by enabling users to specify a **desired replication degree**, and on the service level by implementing some **fault tolerance methods**.

2. Protocols

With this backup service, a client can backup, restore and delete a file, as well as reclaim the maximum disk space used by a peer for storage. The client can also check the current state of a peer. In this section we go over the interface as well as the four protocols implemented for the mentioned operations.

For our project the protocols were implemented using TCP connections with JSSE.

2.1 - Interface

A client (src/TestApp.java) connects to a Peer's access point using an interface communication, to initialize the services (Backup, Restore, Delete, Reclaim).

Our interface (src/ServiceInterface.java) supports a function for each protocol that is called by the class client after a successful connection.

```
public interface ServiceInterface {
    String backup(String file_name, int replicationDegree);
    String restore(String file_name);
    String delete(String file_name);
    String reclaim(long max_disk_space);
    String state();
}
```

The call syntax is the following: `java <peer_address> <ssl_port> <sub_protocol> <opnds>`

2.2 - Backup Protocol

`java TestApp.java <peer_address> <ssl_port> BACKUP <file> <rep_deg>`

The **backup protocol** (src/Backup.java) implemented in this project is similar to the backup protocol implemented in the first project in many aspects. The peer that initiates this service (src/Peer.java line 340) starts by checking if it owns the requested file to backup and if it has been already backed up. If the file exists and hasn't been backed up yet, the peer sends to the main peer the message to backup the requested file in the right number of peers, according to the desired replication degree.

Message: `<peer_id>:BACKUP:<file_name>:<rep_degree>:<file_length>`

When the main peer gets this message, it finds out which peers are able to backup the requested file. Only the peers that aren't the main peer nor own the requested file and have enough space can do a backup. The main peer sends a list with all the peers' information (address and port) that can do the backup.

The peer that requests the backup will now have the list of all the peers that can perform the backup. After parsing the list, the peer sends to all the peers from the list a message to perform the backup of the requested file.

Message: `BACKUP:<file_name>:<file_length>`

The peers that receive this message perform the backup of the file, saving it in a folder called backups.

With this, the backup protocol is complete.

```
public synchronized void performBackup(PeerContainer peerContainer) {
        try {
            FileManager backedUpFile = new FileManager("peer " + pID + "/" +
"backups/" + filename, -1);
            peerContainer.addBackedUpFile(backedUpFile);
            Path path = Path.of("peer " + pID + "/" + "backups/" + filename);
            saveFile(peerContainer, backedUpFile, path, content, pID,
filename);
        } catch (IOException e) {
            System.err.println("> Peer " + pID + " exception: failed to save
file " +  filename);
            e.printStackTrace();
        }
    }

static void saveFile(PeerContainer peerContainer, FileManager restoredFile,
Path path, byte[] content, int pID, String filename) throws IOException {
        AsynchronousFileChannel fileChannel =
AsynchronousFileChannel.open(path, StandardOpenOption.CREATE,
StandardOpenOption.WRITE);
        fileChannel.write(ByteBuffer.wrap(content), 0);
        peerContainer.addFreeSpace(-restoredFile.getFile().length());
        fileChannel.close();
        System.out.println("> Peer " + pID + ": saved file " + filename);
}
```

2.3 Restore Protocol

`java TestApp.java <peer_address> <ssl_port> RESTORE <file>`

The **restore protocol** (src/Restore.java) is responsible for restoring previously backed up files on a certain peer. The peer that calls the restore protocol (src/Peer.java line 389) , starts by checking if the desired file has been previously backed up or if it actually exists on that peer's file system. If the file exists and has already been backed up, the peer sends a message to the main peer to restore the desired file.

Message: `<peer_id>:RESTORE:<file_name>`

After receiving this message the main peer finds out which of the peers own the desired file and are able to send it to the initial peer. The main peer then sends a list with all the peers' information (address and port) that can perform the restore.

The peer that requested the restore will now have the list of all the peers that can perform it. After parsing the list, the peer sends to all the peers from the list a message to perform the restore of the requested file.

Message: `REQUESTRESTORE:<file_name>:<my_address>:<my_port>`

The peers that receive this message perform the restore of the file, sending it to the peer that requested it.

Message: `GETRESTORE:<file_name>:<file_length>`

If nothing goes wrong and at least one peer able to restore the file is online, the peer that requested the file restore will recreate the file on it's own filesystem, and this protocol ends here.

```
public synchronized void performRestore(PeerContainer peerContainer) {
        try {
            FileManager restoredFile = new FileManager("peer " + pID + "/" +
"files/" + filename, -1);
            peerContainer.addStoredFile(restoredFile);
            Path path = Path.of("peer " + pID + "/" + "files/" + filename);
            Backup.saveFile(peerContainer, restoredFile, path, content, pID,
filename);
        } catch (IOException e) {
            System.err.println("> Peer " + pID + " exception: failed to save
file " +  filename);
            e.printStackTrace();
        }}
```

2.4 Delete Protocol

`java TestApp.java <peer_address> <ssl_port> DELETE <file>`

For the **delete Protocol** (src/Delete.java), the initiator peer (src/Peer.java line 389) starts by asserting that the requested file was previously backed up by other peers or if it exists in the peer's filesystem.

If so, a message is sent to the main peer in order to perform the deletion of the file.

Message: `<peer_id>:DELETE:<file_name>`

Like what happens on the backup protocol, the main peer will then search for the peers that have restored that file and return their information (address and port) to the initial peer in a list.

The peer that requested the delete will now parse the list, and send to all those peers a message to perform the deletion of the requested file.

Message: `DELETE:<file_name>`

The peers that receive this message will perform the deletion of the file, ending this protocol's lifecycle.

```
public synchronized void performDelete() {
        ArrayList<FileManager> toBeDeleted = new ArrayList<>();
        for (FileManager file : peerContainer.getBackedUpFiles()) {
            if (file.getFile().getName().equals(filename)) {
                peerContainer.addFreeSpace(file.getFile().length());
                Executors.newScheduledThreadPool(5).schedule(() -> {
                    peerContainer.deleteStoredBackupFile(file);
                }, 0, TimeUnit.SECONDS);
                toBeDeleted.add(file);
            }
        }
        // Delete From Memory
        for (FileManager file : toBeDeleted) {
            peerContainer.getBackedUpFiles().removeIf(f -> f.equals(file));
        }
}
```

2.5 Reclaim Protocol

`java TestApp.java <peer_address> <ssl_port> RECLAIM <max_disk_space>`

The **reclaim protocol** (src/Reclaim.java) is responsible for redefining the max capacity of a peer to store backed up files from other peers. To initiate this protocol, the referenced peer (src/Peer.java line 389) finds out if the total used space occupied by backed up files from other peers is less than the new desired max capacity. If this doesn't happen, then the peer selects the first backed up file from its workspace to be deleted and notifies the main peer with a message.

Message: `<peer_id>:RECLAIM:<file_name>`

After receiving this message the main peer finds out which of the peers don't own the desired file and are able to store it. The main peer then sends a list with all the peers' information (address and port) that can perform the backup of the deleted file, to preserve its replication degree.

The peer that requested the reclaim will now parse the list, and send to one of those peers a message to perform the backup.

This process is repeated until the occupied space in the initial peer is less or equal to the new desired max capacity.

```
public synchronized void performReclaim() {
        boolean ownedFile = false;
        for (FileManager file : peerContainer.getStoredFiles()) {
            if (file.getFileID().equals(fileId)) {
                ownedFile = true;
file.setActualReplicationDegree(file.getActualReplicationDegree() - 1);
            }
        }
    }
```

2.6 State Protocol

`java TestApp.java <peer_address> <ssl_port> STATE (any normal peer)`
`java TestApp.java <peer_address> <ssl_port> MAINSTATE (main peer)`

We implemented a **state protocol**, where it is possible to perceive the peer's actual state.

If the state command refers to the **MAINSTATE**, it will display all the information regarding the peers that are registered in the main peer.

On the other hand, if the state command refers to the **STATE**, only the information about the given peer file system will be displayed.

The information includes the stored files in the peer's directory and the files that were previously backed up from other peers. The stored files information includes the file names, the files ids, if the files were backed up and, if they were, it also includes the files replication degree (current and desired). The backed up files information is similar to the stored files information.

In addition to this information, in the state protocol it is also shown the maximum size of the peer and it's empty space size.

The peer state is mainly updated in two different ways. The first way recurs to the usage of a thread that automatically updates and saves the peer state every 3 seconds. The second way happens every time we make changes on the peer container due to protocol operations. The state is being saved after each successful file back up, delete or store.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::                                PEER 2                                  :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::                              OWN FILES                                 :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::: FILE 1                                                                 :::
:::                                                                        :::
::: IS BACKED UP: NO                                                       :::
::: PATH: peer 2\files\imagem.jpg                                          :::
::: FILE_ID: 6478e746486a1304b1ba56a93179951aee4e9e361a3a3888932c0067c1658e9a :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::: FILE 2                                                                 :::
:::                                                                        :::
::: IS BACKED UP: NO                                                       :::
::: PATH: peer 2\files\teste.txt                                           :::
::: FILE_ID: 850d2635d196f4bbc7fee56bd18a62984535f915cca1f859c095c99c445c72fd :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::                          OTHER BACKED UP FILES                         :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::: NO BACKED UP FILES FROM OTHER PEERS                                    :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::                          PEER STORAGE CAPACITY                         :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::: TOTAL DISK SPACE: 10000000 Bytes                                       :::
::: FREE SPACE: 10000000 Bytes                                             :::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

3. Concurrency Design

In order to increase the possibility of concurrency between processes and, by doing so, decreasing the time necessary for the peers to get the desired files we adopted several practices.

We implemented a solution capable of dealing with multiple instances of protocols and synchronizing their tasks. Therefore, instead of using HashMaps as the data structure to save the information of the communication between peers, we used ConcurrentHashMap. This data structure is an efficient solution because it is thread-safe and has a great performance in multithreaded environments. In the code below it's demonstrated the usage of this data structure in the class Peer.

```
static ConcurrentHashMap<String, ConcurrentHashMap<Integer, byte[]>>
backupOpFiles = new ConcurrentHashMap<>();
static ConcurrentHashMap<String, ConcurrentHashMap<Integer, byte[]>>
restoreOpFiles = new ConcurrentHashMap<>();
```

A thread pool implementation was chosen to synchronize all services, creating a new thread for every action of each protocol, ensuring, with the use of synchronized methods, that each peer completes its action without getting interrupted.

```
ScheduledThreadPoolExecutor peerChannelExecutors = new
ScheduledThreadPoolExecutor(Utils.MAX_THREADS);
```

Implementation of Java NIO was useful for solving the resource consumption issues these threads provide, by reducing the amount of time a thread stays blocked, ensuring all IO operations are non-blocking, which provides a more efficient use of threads.

4. JSSE

Data that travels across a network can be easily accessed or even modified by someone who is not the intended recipient. It is important to protect private information and make sure data hasn't been modified, either intentionally or unintentionally, during transport.

The Java Secure Socket Extension (JSSE) provides a framework and an implementation for a Java version of the SSL and TLS protocols, protocols designed to help protect the privacy and integrity of data while it is being transferred across a network.

We were given the choice between using the SSLSockets or SSLEngine from the JSSE Java Packet. We opted to use the more basic interface: SSLSockets. This interface allowed us to use Java NIO's non-blocking SocketChannels.

```
public class PeerChannel extends Thread {
    AsynchronousServerSocketChannel mainChannel;
    AsynchronousSocketChannel serverChannel;
    ScheduledThreadPoolExecutor peerChannelExecutors = new
ScheduledThreadPoolExecutor(Utils.MAX_THREADS);
```

5. Scalability

In order to provide scalability to the solution, we made use, at implementation level, of thread-pools with Java NIO, running each peer on his own thread and it's operations on thread pool executors, as well as in the Main Peer channel, each connection to each peer being treated as a Callable *worker*, being assigned to a thread pool that would deal with the communications of that same peer.

```
clientChannel = asynchFuture.get();
Callable<String> worker = () -> {
      String host = clientChannel.getRemoteAddress().toString();
      System.out.println("PeerChannel: Incoming connection from a peer at: " +
host);
      while(!closeChannel) {
            ByteBuffer buffer = ByteBuffer.allocate(Utils.MAX_BYTE_MSG);
            Future<Integer> result = clientChannel.read(buffer);
            result.get();
            byte[] msgBytes = buffer.array();
            buffer.flip();
            peerChannelExecutors.execute(() -> {
            Future<Integer> writeResult;
                  if(isMainPeer) writeResult =
clientChannel.write(ByteBuffer.wrap(MainPeer.messageFromPeerHandler(msgBytes).
getBytes()));
                  else writeResult =
clientChannel.write(ByteBuffer.wrap(Peer.messageFromPeerHandler(msgBytes).getB
ytes()));
                  try {
                   System.out.println("Sent " + writeResult.get() + "
bytes.");
                  } catch (InterruptedException | ExecutionException e) {
                   e.printStackTrace();
                  }
                  buffer.clear();
            });
      }
      clientChannel.close();
      System.out.println("PeerChannel: Closing connection to one client.");
      return host;
};
```

6. Fault tolerance

Each time the peers communicate to fulfill a protocol, we have several verifications and conditions in place testing several fault situations to prevent the protocols from failing because of the sudden unavailability of a peer.

We are also verifying the existence the files on peers' filesystems or if they have been backed up before, on operations that require it, to minimize possible exceptions.

```
if (filemanager == null)
         return "Unsuccessful BACKUP of file " + file_name + ", this file does
not exist on this peer's file system";
     if (filemanager.isAlreadyBackedUp()) {
        System.out.println("This file is already backed up, ignoring command")
        return "Unsuccessful BACKUP of file " + file_name + ", backup of this
file already exists";
        }
```