Distributed Backup Service
Enhancements

Mestrado Integrado em Engenharia Informática e Computação

Sistemas Distribuídos

João Lemos - 201000660
Tomas Mendes - 201806522

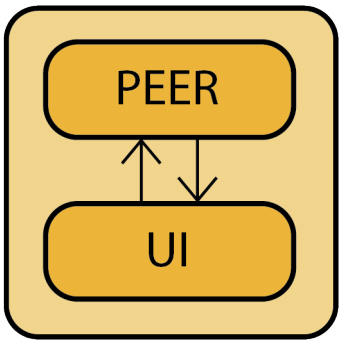
Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

April 12, 2021

Contents

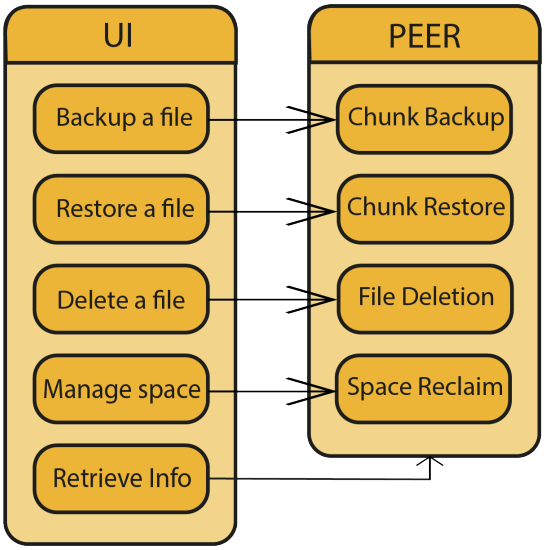
1 Introduction

This project was developed with the intent of developing a distributed backup service for a local area network (LAN). The idea is to use the free disk space of the computers in a LAN for backing up files in other computers in the same LAN. The service is provided by servers in an environment that is assumed cooperative (rather than hostile). Nevertheless, each server retains control over its own disks and, if needed, may reclaim the space it made available for backing up other computers' files. Because no server is special, we call these servers "peers". (This kind of implementation is often called serverless service.) Each peer is identified by an integer, which is unique among the set of peers in the system and never changes. The servers are called peers, each peer is identified by an integer, which is unique among the set of peers in the system and never changes. Users can request the service to perform certain tasks through the use of a User Interface, in this case the UI can be used though the java class TestApp.

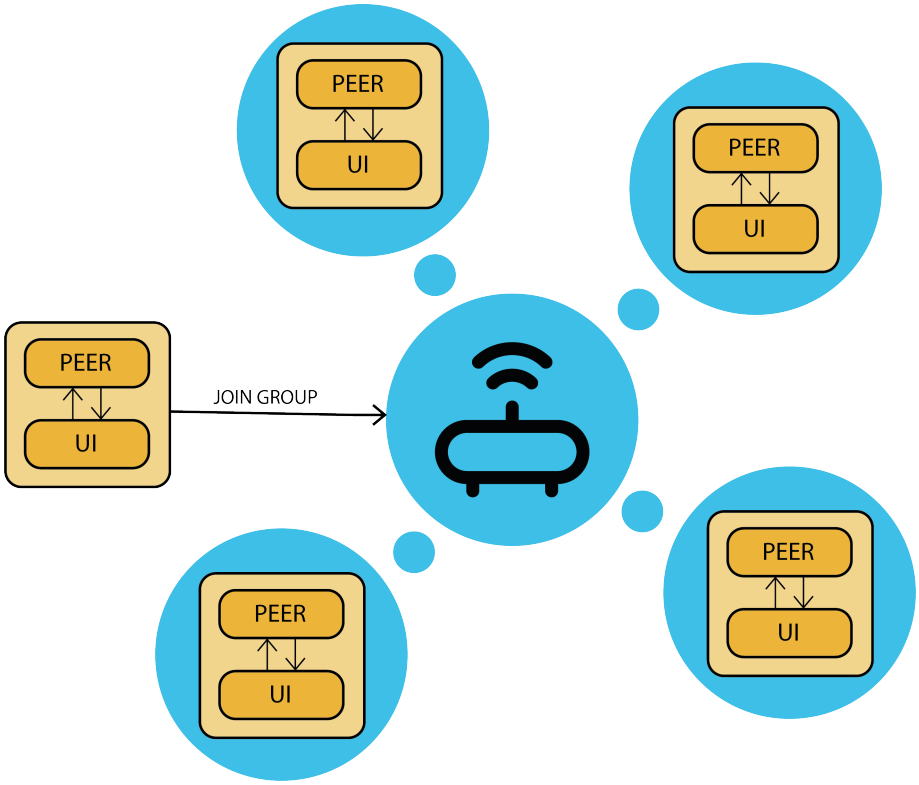


The use of the distributed backup system means that the files processed in the system need to be able to be divided into a fragment of information called chunks. Each chunk contains, besides the information of the parent file a replication degree that descrives the number of times that same chunk is stored somewhere in the system.

The UI described earlier represents the Service Interface that allows users to send requests to their peer. The Peer protocol used by the backup service comprises several smaller subprotocols, which are used for specific tasks, and that can be run concurrently.



A Multicast Control Channel is a point-to-multipoint downlink channel used for transmitting multimedia broadcast multicast service, control information from the network to the UE, for one or several multicast traffic channels. In our service each peer when created joins a multicast group.



2 Thread Scheduling

As each peer protocol can require a large number of threads the choice was made to use ThreadPoolExecuters instead of ThreadSleeps to prevent the coexistence of a large number of threads running at the same time.
A ThreadPoolExecutor that can schedule commands to run after a given delay, or to execute periodically. This class is preferable to Timer when multiple worker threads are needed, or when the additional flexibility or capabilities of ThreadPoolExecutor (which this class extends) are required. Delayed tasks execute no sooner than they are enabled, but without any real-time guarantees about when, after they are enabled, they will commence. Tasks scheduled for exactly the same execution time are enabled in first-in-first-out (FIFO) order of submission.

```
Peer peer_ap = new Peer(args);
scheduledThreadPoolExecutor.execute(multicastBackup);
scheduledThreadPoolExecutor.execute(multicastControl);
scheduledThreadPoolExecutor.execute(multicastRecover);
```

3 Enhancement - Chunk backup sub protocol

One problem of having the PUTCHUNK messages spread in a Multicast Channel is that sometimes several peers could end up backing up the same chunk at the same time ending up with a greater Replication Degree tham the one described in the ReplicationDeg field of the message. To prevent this problem a key was created using a ConcurrentHashMap that associates a chunk with it's current Replication Degree. This key was named chunkPercRepDegree.

```
private final ConcurrentHashMap<String, Integer> chunkPercRepDegree = new ConcurrentHashMap<>();
```

This chunkPercRepDegree is them compared with the Desired Replication Degree described in the message and while the Perceived RD is smaller tham the Desired RD the service will store the chunks in different peers.

```
case "PUTCHUNK":
    try {
        String chunkID = this.messageHeader[3] + this.messageHeader[4];
        int desired_rep_degree = Integer.parseInt(this.messageHeader[5]);
        int randomNum = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 401);
        Thread.sleep(randomNum);
        int perceived_rep_degree = Peer.getStorage().getChunkPercRepDegree().getOrDefault(chunkID,  defaultValue: 0);
        if(perceived_rep_degree < desired_rep_degree){
            handlePUTCHUNK();
        }
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
    break;
```

4 Enhancement - File deletion sub protocol

During the running time of the system a peer that had backed up some chunks of the file might not be not running at the time the initiator peer sends a DELETE message for that file. This type of event is not reliably preventable and as such a way of correcting the errors that can come from this event was created.
To start with the Java Serializable Interface was implemented in the protocol version 1.1 allowing us to Serialize and Deserialize.

```java
// Fonte: https://www.tutorialspoint.com/java/java_serialization.htm
public void serializeStorageInformation() throws IOException {
    String mainDir = "chunks";
    File newMainDir = new File(mainDir);
    boolean res1 = newMainDir.mkdir();

    //new directory for peer
    String dir = newMainDir.getAbsolutePath() + "/peer" + Peer.getPeerID();
    File newDir = new File(dir);
    boolean res2 = newDir.mkdir();


    //new file
    String file = newDir.getAbsolutePath() + "/storage_information";
    File newFile = new File(file);
    boolean res3 = newFile.createNewFile();

    try {
        FileOutputStream fileOutputStream = new FileOutputStream(newFile);
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
        objectOutputStream.writeObject(this);
        objectOutputStream.close();
        fileOutputStream.close();
        System.out.println("Serialized data is saved in " + newFile.getCanonicalPath());
    } catch (IOException i) {
        i.printStackTrace();
    }
}
```

Serialization allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine. And deserialization allows us to reverse the process, which means reconverting the serialized byte stream to an object again.

```java
// Fonte: https://www.tutorialspoint.com/java/java_serialization.htm
public void deserializeStorageInformation(){
    String filepath = "chunks/peer" + Peer.getPeerID() + "/storage_information";
    File storageInfo = new File(filepath);
    if(!storageInfo.exists() || !storageInfo.isFile()) {
        System.out.println("File doesn't exist or it is not a file");
        return;
    }


    try {
        Storage storage = new Storage((long) (64 * Math.pow(10, 9)));
        FileInputStream fileInputStream = new FileInputStream(storageInfo);
        ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
        storage = (Storage) objectInputStream.readObject();
        objectInputStream.close();
        fileInputStream.close();
        Peer.setStorage(storage);
    } catch (IOException i) {
        i.printStackTrace();
    } catch (ClassNotFoundException c) {
        System.out.println("Storage class not found");
        c.printStackTrace();
    }
}
```

With this even when a peer shuts down, once restarting all the information will return to it working as a small waking up instead of a full reboot.
Upon waking up the peer with the protocol version 1.1 warns every other peer that he is back on. Peers will then send back information of all deletions that happend in the meanwhile, saved in a list in storage (filesDeleted).

```java
@Override
public void delete_protocol(String file) throws IOException, InterruptedException, NoSuchAlgorithmException {
    boolean flag = false;

    if(!Peer.getStorage().getFileList().isEmpty()){

        for(int i = 0; i < Peer.getStorage().getFileList().size(); i++) {
            if (Peer.getStorage().getFileList().get(i).getFilepath().equals(file)) {
                flag = true;
                Message deleteMessage = new Message(protocol_version,
                        messageType: "DELETE",
                        peerID,
                        Peer.getStorage().getFileList().get(i).getFileId(),
                        chunkNumber: null, rep_degree: null, body: null);
                byte[] deleteMessageByteArray = deleteMessage.buildMessage();
                multicastControl.sendMessage(deleteMessageByteArray);
                Peer.getStorage().getFilesDeleted().add(Peer.getStorage().getFileList().get(i).getFileId());
                for(int j = 0; j < Peer.getStorage().getFileList().get(i).getChunkList().size(); j++){
                    String chunkID = Peer.getStorage().getFileList().get(i).getChunkList().get(j).getFileId() +
                            Peer.getStorage().getFileList().get(i).getChunkList().get(j).getChunkNumber();
                    if(Peer.getStorage().getChunkPercRepDegree().containsKey(chunkID)){
                        int numStores = Peer.getStorage().getChunkPercRepDegree().get(chunkID);
                        Peer.getStorage().getChunkPercRepDegree().remove(chunkID);
                    }
                }
                Peer.getStorage().getFileList().remove(i);
                i--;
            }
        }

        if(!flag) System.out.println("\nPeer" + peerID + " doesn't contain " + file + " :(");
    }
    else System.out.println("\nNo files to delete in peer" + peerID + " :(");
```

If our peer has the chunk refereed in the message saved up it will proceed to delete it.

```java
private void handleREADY(String[] messageHeader) throws IOException {

    for(int i = 0; i < Peer.getStorage().getFilesDeleted().size(); i++){
        Message deleteMessage = new Message(messageHeader[0],
                messageType: "DELETE",
                Peer.getPeerID(),
                Peer.getStorage().getFilesDeleted().get(i),
                chunkNumber: null, rep_degree: null, body: null);
        byte[] deleteByteArray = deleteMessage.buildMessage();
        Peer.getMulticastControl().sendMessage(deleteByteArray);
    }
}
```

In case a new backup is made of a file previously deleted, this deletion will be removed from the filesDeleted list and as such the peer wont delete them after waking up.

```java
@Override
public void backup_protocol(String file, int rep_degree) throws RemoteException {
    System.out.println("starting Backup Protocol\n");

    try {
        String fileId = generateFileID(file);
        if(Peer.getStorage().getFilesDeleted().contains(fileId)) Peer.getStorage().getFilesDeleted().remove(fileId);
```