Faculdade de Engenharia da Universidade do Porto

Sistemas Distribuídos

Peer-to-peer backup service for the Internet

2019/2020

sdis1920-t5-g21

Ana Macedo – up201909572

Guilherme Sousa - up201909575

Joel Coelho - up201909577

Miguel Castro - up201909578

Index

Overview

This project aims to develop a peer-to-peer distributed backup service for the Internet, using the free disk space of the computers on the Internet for backing up files on one's own computer.

As in the first project, the service supports all the required operations: **backup**, **restore,** and **deletion** of files and the **reclaim** of the space used by a peer. For the backup protocol, each file is split in **12KB size chunks**, except the last one that can have a smaller size. The chunks are merged in one file before being stored.

JSSE was implemented for secure communication between peers, using the **SSLEngine** interface. This interface is more flexible than SSLSockets and allows achieve higher concurrency.

To address the scalability issue at the **design level**, the service implements a distributed hash table (DHT) using **Chord**, to locate the peers storing a given chunk of a file in a "peer-to-peer" system. At the implementation level, **thread-pools** and asynchronous I/O (**Java NIO**) are used.

To avoid single-points of failure, **fault-tolerant** features were implemented.

Throughout this report is described, explained, and shown the design and implementation of the backup service developed by our group.

With this project we could learn about peer-to-peer communications and their security, as well as how to implement a scalable and fault-tolerant P2P backup service.

Protocols

With the 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.

In this section the remote interface connection is explained as well as the four protocols (backup protocol, restore protocol, delete protocol and reclaim protocol) implemented for the mentioned operations.

The protocols for this project were implemented using TCP connections with JSSE and when a Peer wants to send a message to another Peer, it uses the class Send (src\protocolActions\Send.java) to format the message that they want to send. This message can have the operation, a file, the peer and the desired replication degree.

When a Peer receives a message from another Peer it is parsed and handled in the parseMessage() method (communicationChannels.peer.actions.PeerMessageAction.java) depending on the desired operation received:

```
public static Send parseMessage(Send received, Peer p) throws NumberFormatExc
eption, Exception {
switch (operation) {
      case Send.SET_PREDECESSOR: …
```

```
        …
        case Send.BACKUP: …
        case Send.CONFIRM_BACKUP: …
        case Send.RESTORE: …
        case Send.CONFIRM_RESTORE: …
        case Send.DELETE: …
        case Send.REMOVED: …
        case Send.NEW_BACKUP: …
}
```

For the backup, restore and delete protocols the destination peer of the Send message is generated from a key containing the file info.

```
MessageDigest md3 = null;
            try {
                md3 = MessageDigest.getInstance("SHA1");
            } catch (NoSuchAlgorithmException e2) {
                e2.printStackTrace();
            }
            md3.reset();
            md3.update(path.getBytes());
            byte[] hashBytes3 = md3.digest();
            BigInteger hashNum3 = new BigInteger(1, hashBytes3);
            int key3 = Math.abs(hashNum3.intValue()) % Peer.numDHT;
             destination = ChordManager.find_successor(key3);
```

Remote Interface

A client (src/TestApp.java) tries to connect to a Peer's access point, using a Remote Interface (RMI) communication, to initialize the services (Backup, Restore, Delete, Reclaim).

The Remote Interface (src/rmi_interface/RMI_Interface.java) supports a function for each protocol, that are called by the class client after a successfully connection:

```
public interface RMI_Interface extends Remote {
        void backup(String path, int replicationDegree, String senderId) thro
ws RemoteException, Exception;
      void restore(String path, String senderId) throws RemoteExceptio;
      void delete(String path, String senderId) throws RemoteException;
      void reclaim(int spaceToReclaim, String senderId) throws RemoteExcepti
on;
}
```

The RMI connection for the Peer is initialized on the lines 173-175:

```
final RMI_Interface stub = (RMI_Interface)
UnicastRemoteObject.exportObject(peer, 0);

            final Registry registry = LocateRegistry.getRegistry();

            registry.bind(access_point, stub);
```

Backup Protocol

For the **backup protocol** (src\threads\BackupThread.java), the peer that initiates this service starts by checking if there are enough nodes available to satisfy the desired replication degree and finds its successor, a node responsible for a generated key hashing the file info.

After that, the peer reads the file content and a new SendFileThread thread is created (line 61) and executed (line 63) to send the file's chunks to the successor. It starts by creating a new Send instance to format the PUTCHUNK message to be sent.

```
Send send = new Send(Send.BACKUP, file, p.me);
```

After formatting the message, it is called the writeFile() function (line 100), where the file is split into chunks up to 12KB and each chunk is sent to the peer's successor by calling the function write() (src\protocolActions\Send.java, line 182), where the "send" argument has the operation, the chunk and the initiator peer and the hostAddress and hostPort are the successor's address and SslEnginePort, respectively:

```
write(send, protocol, hostAdress, hostPort);
```

When a Peer receives the Backup message (src\communicationChannels\peer\actions \PeerMessageAction.java, line 76), it merges the chunks into one file, saves it if the peer is not the initiator, increments the replication degree and sends back the STORED confirmation.

If the peer is the initiator, it decrements the 5 retries to achieve the replication degree sends the same message to its successor.

Restore Protocol

For this protocol (src\threads\RestoreThread.java), a peer that wants to restore a file previously backed up to the Chord network creates a new Send instance, to format the GETCHUNK message to be sent, with the file info.

The message is sent to the peer's successor using the write() method, where the destination.address and destination.getSslEnginePort() are the successor's address and SslEngine port.

```
Send restoreSend = new Send(Send.RESTORE, file, p.me);
```

```
        restoreSend.write(restoreSend,  Peer.PROTOCOL, destination.address, d
estination.getSslEnginePort());
```

If the successor does not have the file backed up, it sends the same message to its successor, otherwise, it sends back the backed-up file.

Delete Protocol

For the delete Protocol (src\threads\DeleteThread.java), the initiator peer that wants to use a similar process to the restore protocol, creating a Send instance, to format the DELETE message to be sent, with the file info.

The message is sent to the peer's successor using its address and SslEngine port:

```
Send send = new Send(Send.DELETE, file, p.me);
                send.write(send, Peer.PROTOCOL, destination.address, destinat
ion.getSslEnginePort());
```

When receiving a delete request, a peer removes the file from its storage and decreases the perceived replication degree. If this perceived replication degree is greater than 0, the peer sends the same message to its successor.

Reclaim Protocol

The reclaim Protocol (src\threads\ReclaimThread.java) allows a client to reclaim the space on peers for backing up files. The initiator peer sorts its stored files by perceived replication degree and removes the smaller ones until the total storage space is less than the desired size. After that, the peer sends a REMOVED message to its successor, who sends the same message to its successor and so on. When receiving the request, a peer decrements the perceived degree and sends a new backup request if it becomes smaller than the replication degree.

Concurrency Design

For the concurrency issue, it was implemented a solution capable of dealing with multiple instances of protocols and synchronizing their tasks.

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.

```java
public synchronized void backup(final String path, final int replicationDegree, final String senderId)
        throws Exception {
    // Send putchunk message to BackupThread
    File directory = new File(path);
    if (!directory.exists()) {
        System.out.println("Could not find file " + path);
        return;
    }
    final BackupThread backup = new BackupThread(this, path, replicationDegree, senderId, enhanced);
    this.getThreadPoolExecutor().execute(backup);
}
```

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.

To manage all peers, it was created a centralized chord manager which controls how all chord nodes are connected that lets the peers get information about the connected peers.

```java
public class CentralizedChordManager {
    static ServerRunnable serverRunnable;      You, 11 days ago • Initial commit to chord

    static public int byteNumber;
    public static int numPeers;
    static public int numDHT;
    private static Peer[] nodeList;
    static private CopyOnWriteArrayList<Integer> nodeIDList = new CopyOnWriteArrayList<Integer>();
    static private CentralizedChordManager chordManager;

    public CentralizedChordManager(int maxNumPeers) {
        byteNumber = (int) Math.ceil(Math.log(maxNumPeers) / Math.log(2));
        numDHT = (int) Math.pow(2, byteNumber);

        nodeList = new Peer[numDHT];
    }

    /**
     * @param nodeIP
     * @param receivedPort
     * @return
     */
    public String registerPeer(String nodeIP, int receivedPort) {
        synchronized (this) {
            int nodeID = 0;
            String initInfo = "";
            String nodePort = String.valueOf(receivedPort);
```

JSSE

To tackle the JSSE issue, SSLEngine was the desired interface used to communicate Send messages as explained in the Protocol section, due to its capacity to use secure communication channels using protocols such as, in this case, the TLSv1.2 version. In comparison to SSLSockets, SSLEngine is more flexible allowing a better integration with other API.

This interface includes Integrity Protection to protect mutations of messages by active wiretappers, Authentication for each Peer and client, and Confidentiality providing a more protected communication of messages between server and client by encrypting the content. For the authentication mode, it was used the files provided in the practical classes, therefore, the server uses a truststore and its server keys and the client uses the same truststore and its client keys.

In order to implement the interface, each peer and the centralizedChordManager initializes a new serverchannel thread.

```
public Peer(final int peerId, final int sslEnginePort, String centralizedChordManagerAddress,
    int centralizedChordManagerPort, String address) throws IOException {
  this.peerId = peerId;
  this.sslEnginePort = sslEnginePort;
  try {
    mainChannel = new ServerChannel(this, "TLSv1.2", address, sslEnginePort);
    // Register this peer on Centralized server
  } catch (Exception e) {
    e.printStackTrace();

  }
  this.threadPoolExecutor = Executors.newScheduledThreadPool(100);
  this.storage = new Storage();
  this.initiatorPeerFiles = new CopyOnWriteArrayList<FileModel>();
  this.peerRepliesMap = new ConcurrentHashMap<>();
  this.nodesChord = new ArrayDeque<>();
  this.address = address; // CHANGE TO OTHER IP IF NEEDED
  this.centralizedChordManagerAddress = centralizedChordManagerAddress;
  this.centralizedChordManagerPort = centralizedChordManagerPort;
}
```

Every time a message needs to be sent it is created a clientchannel thread for the communication.

```
public void write(final Send send, final String protocol, final String hostAdress, final int hostPort) {
  try {
    ClientChannel clientChannel = new ClientChannel(protocol, hostAdress, hostPort);
    ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    ObjectOutputStream objStream = new ObjectOutputStream(byteStream);
    objStream.writeObject(send);
    objStream.flush();
    clientChannel.write(appendFinalInfo(byteStream.toByteArray()));
  } catch (final Exception e) {
    e.printStackTrace();
  }
}
```

Scalability

In order to provide scalability to the solution, it was used Chord, at design level, which assigns a key to every peer(node) that is connected, using distributed hash map for the communication between peers, letting each peer have information about other peers, known as successors. However, a peer only keeps track of O(log-N) peers, in a N-network, trying to equalize the number of successors of each node, which generates a finger table for each one of them, therefore the network takes less messages, preventing it from being flooded.

```java
public void bootChord() {
    System.out.println("Building Finger table ... ");
    for (int i = 1; i <= thisPeer.maxPeersBytes; i++) {
        finger[i] = new FingerTable();
        finger[i].setStart((me.getPeerId() + (int) Math.pow(2, i - 1)) % numDHT);
    }
    for (int i = 1; i < thisPeer.maxPeersBytes; i++) {
        finger[i].setInterval(finger[i].getStart(), finger[i + 1].getStart());
    }
    finger[maxPeersBytes].setInterval(finger[maxPeersBytes].getStart(), finger[1].getStart() - 1);

    for (int i = 1; i <= thisPeer.maxPeersBytes; i++) {
        finger[i].setSuccessor(me);
    }

    if (pred.getPeerId() == me.getPeerId()) { // if predcessor is same as my ID -> only node in DHT
        System.out.println("Done, all finger tablet set as me (only node in DHT)");
    } else {
        try {
            init_finger_table(pred, finger);
            System.out.println("Initiated Finger Table!");
            update_others();
            System.out.println("Updated all other nodes!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    try {
        // SEND FINISH JOINING
        Send request = new Send(Send.FINISH_JOINING, me.getPeerId() + "");
        makeConnection(thisPeer.centralizedChordManagerAddress, thisPeer.centralizedChordManagerPort, request);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

At implementation level, it was used thread pools with Java NIO, as mentioned earlier, running each peer and its respective operations in their own threads providing a scalable and more efficient solution to introducing new peers to the network, isolating all responsibilities of every peer to itself.

Fault-tolerance

To address the fault-tolerance issue, Chord has features that run periodically in the background to ensure that all successor pointers of nodes are up to date. They try to correct irregularities introduced into the ring by a peer failing, joining or leaving. These features correspond to the 3 functions: **stabilize**(), **fix_fingers**() and **notify**(). All of these functions are implemented on the ChordManager class (src\chord\ChordManager.java)

The **stabilize** function of a node aims to find the predecessor of it's immediate successor, in the case if a peer recently joined the system. If the found predecessor is between the node itself and his successor than it is the new successor. After that, the node sends a message to its successor, notifying that it is his predecessor. The successor node when receiving the message invokes the **notify**() function to determine whether this node is closer to him than his current predecessor, and assigns it accordingly.

```java
public static void Stabilize() {
    Send send = new Send(Send.GET_PREDECESSOR);
    try {
```

```
     Peer x = makeConnection(finger[1].getSuccessor().address,
finger[1].getSuccessor().getSslEnginePort(),send).peer;
     Peer p = finger[1].getSuccessor();
     if (thisPeer.getPeerId() < p.getPeerId()) {
     if   (x.getPeerId()   >   thisPeer.getPeerId()   &&   x.getPeerId()   <
p.getPeerId()){
                  finger[1].setSuccessor(x);
            }
     } else if (thisPeer.getPeerId() == p.getPeerId()) {
                finger[1].setSuccessor(x);
     } else {
     if   (x.getPeerId()   >   thisPeer.getPeerId()   ||   x.getPeerId()   <
finger[1].getSuccessor().getPeerId()) {
                    finger[1].setSuccessor(x);
                }
             }
    Send send2 = new Send(Send.NOTIFY);
 send2.peer     =     new     Peer(thisPeer.getPeerId(),     thisPeer.address,
thisPeer.getSslEnginePort());
    makeConnection(finger[1].getSuccessor().address, finger[1].getSuccessor()
.getSslEnginePort(), send2);

    } catch (Exception e) {
            e.printStackTrace();
    }
}



public static void notify(Peer s) {
    Peer predecessor = pred;
       if (predecessor != null) {
          if (predecessor.getPeerId() < thisPeer.getPeerId()) {
             if (s.getPeerId() > predecessor.getPeerId() && s.getPeerId() <
thisPeer.getPeerId()) {
                predecessor = s;
             }
          } else if (predecessor.getPeerId() == thisPeer.getPeerId()) {
              predecessor = s;
          } else {
              if (s.getPeerId() > predecessor.getPeerId() || s.getPeerId() <
thisPeer.getPeerId()) {
                 predecessor = s;
              }
          }
```

```
        } else {
            predecessor = s;
        }
}
```

Regarding the **fix_fingers**, this function updates finger tables, iterating each of its entry, due to Chord's volatility.

```
public static void fix_fingers() {
        for (int i = 1; i <= maxPeersBytes; i++) {
            try {
                finger[i].setSuccessor(find_successor(finger[i].getStart()));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Another function (**check_predecessor**()) should be called periodically to check whether a predecessor has failed or not, but was not successfully implemented.

In order to tolerate the unavailability of peers, our service also backs up each chunk on a given number of peers - **replication degree**. All the chunks of each file are backed up with a **desired replication degree**. When a file is deleted, the replication degree can drop below the desired value, the service initiates backup protocol again.