Distributed Backup Service for the Internet

Distributed Systems

Turma 3 - Grupo 26

Afonso Mendonça *up201706708@fe.up.pt*

João Campos *up201704982@fe.up.pt*

João Renato Pinto *up2017005547@fe.up.pt*

Leonardo Moura *up201706907@fe.up.pt*

30.05.2020

Contents

Introduction

This project, developed in the class of Distributed Systems, has the objective of creating a peer-to-peer distributed backup system. Using free disk space of computers and communication through the Internet, each user can backup their files and later restore them. Our goal is to elaborate a system as scalable and fault tolerant as possible.

Overview

The application consists of several peers. Any peer can backup a file in the other peers and at a later time restore them. They may also delete the file from the system. Each peer is responsible for the amount of disk space it uses for storing files.

Every peer in the system is part of a Chord Ring, this makes it easy for every peer to easily find the responsible for storing their files

In order to allow for concurrent functioning of the system we use worker threads, which are responsible for receiving a task and processing it.

For communication between two peers we make use of the SSLEngine class provided by the Java Secure Socket Extension (JSSE) framework.

Whenever a user wants to make use of this application, he must use the client interface. This is an RMI interface that allows running tasks in each peer.

ClientInterface

Each peer has access to four protocols and the state action, through the Client Interface class:

- Backup Protocol: Store a number of copies of given file;

- Restore Protocol: Restore file that has been previously backed up in the system by the same peer that is requesting;

- Delete Protocol: Delete all copies of the stored file backed up by the same peer that is requesting its deletion;

- Reclaim Protocol: Define, in KBytes, how much space can be used for the backup system. Stored copies in excess are removed from the peer;

- State: Display the information about that peer's files stored in the system, files stored in the peer from other peers and node information on the Chord ring.

Storage

Each peer has a Storage class that holds the maximum space allowed and current occupied storage space, the list of files that the peer backed up in the system, and the list of chunks the peer is currently storing. All peers start with an initial unlimited storage space, which can be changed later using the **Reclaim Protocol**. This class also provides all methods needed for adding, removing and accessing information from storage.

```
public class Storage {
  private long maxStorage; // -1 equals to unlimited
  private long currStorage;

  private List<FileInfo> filesBacked;
  private List<ChunkInfo> chunksStored;
  private int peerId;
```

FileInfo

FileInfo is a utility class which holds all the information about a file: its id, its path, the desired number of copies in the system (repDegree), and a list of chunks. The id of a file is generated through the hashing of data and metadata.

The class also provides methods to retrieve the information about the file asked to backup and for a specific chunk**.**

```
public class FileInfo {
  private String id;
  private final String path;
  private final int repDegree;
  private final List<ChunkInfo> chunks;
```

ChunkInfo

This class holds all the information about a chunk: the id of the file it belongs to, its number, that represents its index when the file was split, the desired number of copies in the system (wantedRepDegree), the current amount (currRepDegree), started as 0, its size in bytes, and a boolean that represents if the file is currently kept in storage or was delegated to a successor, if it was delegated, stores the **NodeReference** of the node that is storing it. Also provides all the necessary methods to retrieve information about the chunk.

```
public class ChunkInfo {
  private int no;
  private String fileID;
  private int wantedRepDegree;
  private int currRepDegree = 0;
  private int size;
  private int copyNo;
  private boolean delegated;
  private NodeReference receiver;
```

Protocols

Backup Protocol

For the backup protocol the user must provide a file through its path and the desired number of copies to put in the system. The system checks the file for its size (maximum of 1 GByte) and splits it into chunks of 16 KBytes.

```
public static int CHUNK_SIZE = 16000; // Max size of the chunks
```

For each of those chunks, the ***PeerMethods.backupChunk()*** method is called that receives fileID, chunk number, the number of wanted copies in the system for the chunk to backup, and, for each one of the copies, uses Chord to get the node where it should be stored and sends a PUTCHUNK *(FORMAT: "PROTOCOL PUTCHUNK <FileID> <ChunkNo> <CopyNo> <CRLF> <CRLF> <CONTENT OF CHUNK>") m*essage, checks reply for errors ("ERROR"), and on success ("SUCCESS") increments the Storage current number of copies in the system tracker.

On the receiver peer end the message gets processed in the MessageProcessor class, and the ***PeerMethods.saveChunk()*** receives the chunk content and checks if there is enough space available to store it, if yes calls the ***Storage.saveFile()*** (which stores the contents chunk on "Peers/dir<PeerID>" directory) and returns true, so the chunk is stored and SUCCESS message is sent, if there isn't enough space, prints a message and returns false.

In that case, the peer tries to delegate the chunk to its successor, maintaining the ChunkInfo in storage but not the content of the chunk.

Delegation

When a peer does not have enough space to store a chunk, the MessageProcessor calls the ***PeerMethods.delegateChunk()*** method for that specific chunk, which delegates the chunk to the node successor with a DELEGATE message *(FORMAT: "PROTOCOL DELEGATE <FileID> <ChunkNo> <CopyNo> <CRLF> <CRLF> <CONTENT OF CHUNK>")* and if it receives a SUCCESS, stores that successor **NodeReference** updating the delegated boolean, so it knows where the chunk is for restoring and deletion purposes. When receiving that DELEGATE message the Peer follows the same code of the PUTCHUNK message, returning SUCCESS when enough space is available and contents are stored or also tries to delegate and returns SUCCESS if concluded.

Restore Protocol

In the Restore Protocol the user must also provide a file through its path, and the System checks the Peer Storage for the correspondent **FileInfo,** if not found it means the Peer never backed up that file so an error message is displayed, if found, for each of the chunks the ***PeerMethods.restoreChunk()*** method is called that receives fileID, chunk number and the number of copies in the system for the wanted chunk and, for each one of the copies, uses chord to get the node that should have them and sends a GETCHUNK message *(FORMAT: "PROTOCOL GETCHUNK <FileID> <ChunkNo> <CopyNo> "),* checks reply for errors, and on success breaks the loop, and returns the chunk content in byte[] form.

When a chunk is retrieved the ***Storage.restoreChunk()*** method is called and the chunk is stored in "Peers/dir<PeerID>/temp/<FileId>" directory. When this procedure is completed for all chunks the ***PeerMethods.dechunkifyFile()*** method is called which basically restores the original file from all the chunks on the "Peers/dir<PeerID>/restored" directory and deletes the temporary files.

On the receiving end of the GETCHUNK message, the message goes through MessageProcessor and the ***PeerMethods.retrieveChunk()*** method is called that receives the information about the chunk that it needs to send, and checks its storage for it, checks if a chunk was delegated, if not retrieves the chunk from storage and returns it as byte[], if it was delegated, gets the node that stored from the Storage and sends it a GETCHUNK message, receives the chunk contents as an answer and returns it as byte[], that is later sent as a CHUNK message *(FORMAT: "PROTOCOL CHUNK <FileID> <ChunkNo> <CopyNo> <CRLF> <CRLF> <CONTENT OF CHUNK>")*.

Delete Protocol

For the Delete Protocol the user must provide the file through its path, and the System checks the Peer Storage for the correspondent **FileInfo,** if not found it means the Peer never backed up that file so an error message is displayed, if found, for each of the chunks the ***PeerMethods.deleteChunk()*** method is called that receives fileID, chunk number and the number of copies in the system for the desired chunk and, for each one of the copies, uses chord to get the node that should have them and sends a DELETE (*FORMAT: "PROTOCOL GETCHUNK <FileID> <ChunkNo> <CopyNo>* "), message, checks reply for errors, and on the success of deletion of all copies returns true which signal that copy was deleted. After that is completed for all chunks the protocol ends successfully.

On the receiving end of the DELETE message, the message goes through MessageProcessor and the ***PeerMethods.deleteSavedChunk()*** method is called that Receives the information about the chunk that it needs to delete, and checks its storage for it, checks if a chunk was delegated, if not deletes the chunk from storage and from the system, if it was delegated, gets the node that stored from the Storage and sends it a DELETE message, returns true if successfully deleted, and false if otherwise. which sends the SUCCESS or ERROR message correspondingly.

Reclaim Protocol

The Reclaim Protocol mainly updates the maximum storage of a Peer, through the ***PeerMethods.spaceReclaim*** method, if the new maximum is over the currently occupied storage calls the ***PeerMethods.manageStorage*** method that receives the needed bytes to free from storage and checks storage for saved chunks to delete, on the deletion of a chunk, removes it from local system and delegates it to its successor, updating the ChunkInfo information with the receiver **NodeReference,** on enough deletions returns true.

State:

The state “protocol” is just a utility protocol where a user can see the Peer information as following:

Files Backed Up:
- Path of file
- Desired Replication Degree
- Number of Chunks
- For each chunk composing the file:
  - FileID
  - Chunk Number
  - Size in KBytes
  - Current Replication Degree

Chunks being stored:
- FileID
- Chunk Number
- Size in KBytes
- Current Replication Degree

Peer Max Storage

Peer Used Storage

Chord ID

Chord Sucessor

Chord Predecessor

Concurrency design

Concurrency is the ability of a system to run several parts out of order, meaning those parts can be executed in parallel. This will significantly improve the speed of the execution in multi-core systems.

For our System to handle all tasks in parallel, each peer in the system has a ***ScheduledExecutorService*** with 50 threads in the pool. When started it schedules the Chord Handler task to execute every 5 seconds, and creates a ***PeerThread*** task that permanently is listening for messages in its socket, every time it receives a message it creates another thread to handle it via a ***MessageProcessor*** task. Whenever a peer receives a task from the Client Interface via RMI, a thread from that pool is called to execute the said task.

JSSE

Using default Sockets, communication is susceptible to attacks such as Man In The Middle. These attacks aim to intercept unencrypted communication and steal valuable information held by the packets. This is a clear aspect to protect in the service so that important files can not be stolen by malicious users.

In order to achieve a secure and trustworthy communication the Java Secure Socket Extension (JSSE) framework provides a java implementation of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols which include functionality for data encryption, server authentication, message integrity and client authentication.

Because the default interface that provides this service, the SSLSocket, is based on a blocking I/O model, which limits the scalability of the service and the transport layer protocol used (TCP), JSSE provides **SSLEngine** class which separates the secure communication functions (such as handshakes and encryption) from the transport layer, thus allowing us to use a more scalable transport interface, such as **AsynchronousSocketChannels**.

```
/**
 * Wraps a message using SSLEngine and sends it through the channel
 * @param msg The message to be sent
 * @return The number of bytes written to the channel
 * @throws SSLManagerException When something goes wrong
 */
public int write(byte[] msg) throws SSLManagerException {
    debugPrint( msg: "WRITE CALLED");

    this.outAppData.put(msg);

    SSLEngineResult res = null;
    try {
        res = this.wrapAndSend();
    } catch (TimeoutException e) {
        return -1;
    }

    debugPrint( msg: "WRITTEN " + res.bytesProduced() + " bytes");

    return res.bytesProduced();
}
```

Because the SSLEngine is a very complex interface and full of nuances, we opted to encapsulate its functionality and implementation details in the **SSLManager** abstract class. This class provides all the methods needed to interact with the SSLEngine and its chosen transport interface, the **AynchronousSocketChannel**.

SSLManager only provides the core methods to ensure a communication using SSL or TLS, and as an abstract class it needs to be extended by an instantiable class.

In our case the SSLServerInterface and **SSLClientInterface** provide interfaces to operate the SSLManager and deal with the necessary KeyStores and TrustStores. There is also the **SSLEngineServer** class that is used to accept new connections in a server with the accept method that returns a new **SSLServerInterface** that can be used to communicate with the client.

```
/**
 * Unwraps a message and copies it to the array passed as parameter
 * @param message The array where the message will be placed
 * @return The number of bytes of the message read
 * @throws SSLManagerException When something goes wrong
 */
public int read(byte[] message) throws SSLManagerException {
    debugPrint( msg: "READ CALLED");

    try {
        this.unwrap();
    } catch (TimeoutException e) {
        return -1;
    }
    this.inAppData.flip();

    int packLen = this.inAppData.remaining();

    this.inAppData.get(message, offset: 0,packLen);

    debugPrint( msg: "READ " + packLen + " bytes");

    return packLen;
}
```

SSLManager deals with all the technicalities of the secure communication in this project, here

we describe some implementation details referring to our own implementation. First of all, SSLEngine is configured to use the AES256-SHA and AES128-SHA ciphers. These are AES ciphers to be used with TLS (the protocol SSLEngine is configured to use in our project). The engine is also configured to require client authentication. Another important implementation detail is the fact It is also worth noting that TLS is used in all TCP communications using the classes described above.

Finally, SSL manager provides the read and write methods, used all throughout the project. The read method unwraps the receiving network buffer and returns the unwrapped information as a byte array. The write method wraps a message using SSLEngine and sends it through the socket channel.

Scalability

Scalability is the capacity of a system to handle a growing amount of work. With our project in mind, this would be equivalent of increasing the number of peers and files being stored. The main scalability problems would be, firstly, how to know which peers store which chunks of a file and, secondly, how to handle the communication between the peers.

Chord

Every peer will have ***Node*** object field, a representation of themselves in the Chord Ring. Here they will have access to its successor, predecessor, its finger table and the different methods and operations of the Chord protocol. These other references to other peers in the Chord Ring are represented through the ***NodeReference*** *class*. While the first one has the direct logic behind the Chord protocol, the latter establishes the connection, using the IP and Port mentioned in the class fields, and asks the respective peer/node to make the asked operations and return the results.

Every node has an hash calculated using the *<IP>;<Port>* formula with the SHA-1 cryptographic hash function. It outputs a 160-bit string which is then converted to a *BigInteger* and, after that, truncated down to 32-bits. All in all, M=32 and, therefore, the finger table has 32 entries.

Node

As mentioned earlier, the ***Node*** class has all the information needed for the Chord protocol logic. Its constructor receives the peer IP, Port and calculates its Chord ID using the two values. It also initializes the finger table and creates a ***NodeReference*** of himself.

If this Peer is not joining an already established Chord Ring, it must create one by running the ***create()*** method. Since, at time of creation, he will be the only Node in the Chord Ring, he assigns himself as his own successor. Since he will also be the closest Node to every entry in the finger table, he also assigns himself to every finger table entry. Predecessor is first assigned a *null* value because this will eventually be corrected when this

```
public void create() {
  this.predecessor = null;
  this.successor = this.ownReference;

  for (int i = 0; i < M; i++) {
    fingerTable[i] = this.ownReference;
  }
}
```

```java
public class Node {
  public BigInteger id;
  public String ip;
  public int port;

  NodeReference successor;
  NodeReference predecessor;
  NodeReference[] fingerTable;
  NodeReference ownReference;

  private static int M = 32;

  /**
   * Node constructor. Calculates its own hash and initializes fingertable
   */
  public Node(String ip, int port, Peer peer) throws NoSuchAlgorithmException {
    this.ip = ip;
    this.port = port;
    this.id = getHash(ip, port);
    this.fingerTable = new NodeReference[M];
    this.ownReference = new NodeReference(ip, port);
  }
```

Node is properly notified (possibly by himself if another Node doesn't join the Ring quick enough).

Otherwise, he will use the ***join()*** method. Naturally, he needs a reference Node to join the Chord Ring. In other other words, a Node who already is part of the Chord Ring he is about to join. Here he is given the IP and Port of the Node he has the reference of and then creates a ***NodeReference*** object with the given information (the workarounds of this class are explained below). Next he asks this Node to find his successor and the query result will be assigned as such. After this, since, at this stage the joining node only knows about himself and the successor he got from the query, he builds the finger table entries using this information.

```java
public void join(String ip, int port) throws NoSuchAlgorithmException {
  this.successor = new NodeReference(ip, port).findSuccessor(this.id);
  this.predecessor = null;

  fingerTable[0] = this.successor;
  for (int i = 1; i < fingerTable.length; i++) {
    BigInteger fingerId = (this.id.add(new BigInteger("2").pow(i))).mod(new BigInteger("2").pow(M));
    if (clockwiseInclusiveBetween(fingerId, this.id, this.successor.id)) {
      fingerTable[i] = this.successor;
    } else {
      fingerTable[i] = this.ownReference;
    }
  }
}
```

To find a successor, the Node,before anything else, checks if the ID whose successor is being searched if between the Node itself and its successor. If so, it means the successor is the query result. Otherwise, we check the our finger table for the closest

preceding node to that ID. If its ourselves, we return our own ***NodeReference*** as the query result. If not, we ask that Node to find the Successor for the respective ID.

```
public NodeReference findSuccessor(BigInteger id) throws NoSuchAlgorithmException {
  if (clockwiseInclusiveBetween(id, this.id, this.successor.id)) {
    return this.successor;
  } else {
    NodeReference n = closestPrecedingNode(id);
    if (n.id.equals(this.id)) {
      return this.ownReference;
    }
    return n.findSuccessor(id);
  }
}
```

The ***closestPrecedingNode()*** method searches the finger table for the closest preceding node of the given ID. If the finger table is wrong and/or needs updating, it returns the Node successor hoping that he has the needed entry on the finger table. On the worst case scenario, it opts for a linear search all the way through the query.

```
public NodeReference closestPrecedingNode(BigInteger id) {
  for (int i = M - 1; i > 0; i--) {

    if (clockwiseExclusiveBetween(fingerTable[i].id, this.id, id)) {
      return fingerTable[i];
    }
  }
  // return this;
  return this.successor;
}
```

The ***stabilize()*** method basically checks if the Node's successor predecessor should be our successor instead. That can happen when a new Node enters the Chord Ring with an ID between this Node and it's successor. When that new Node eventually notifies him, he will update his Predecessor. This method makes sure that the Node keeps his successor reference up to date. The ***getSuccessorPredecessor()*** is what the name implies, basically.

```
public void stabilize() throws NoSuchAlgorithmException {
  NodeReference x = getSuccessorPredecessor();

  if (x != null && clockwiseExclusiveBetween(x.id, this.id, this.successor.id)) {
    this.successor = x;
    this.fingerTable[0] = this.successor;
  }
  this.successor.notify(this.ownReference);
}

public NodeReference getSuccessorPredecessor() throws NoSuchAlgorithmException {
  return this.successor.getPredecessor();
}
```

The ***notify()*** method is used when another Node, the one given as argument, notifies this Node. He then checks if the notifier Node should be the Node's predecessor. If it is, it

then also gives the chunks that were initially assigned to the Node before the entry of the notifier Node into the Chord Ring.

```
public void notify(NodeReference n) {
  if (this.predecessor == null || clockwiseExclusiveBetween(n.id, this.predecessor.id, this.id)) {
    this.predecessor = n;
    try {
      if(!Peer.givingChunks) {
        Peer.givingChunks = true;
        Peer.giveChunks(n);
      }
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }

  }
}
```

fixFingers() updates the finger table by querying for the respective ID of each entry.

```
public void fixFingers() throws NoSuchAlgorithmException {
  for (int i = M - 1; i >= 1; i--) {
    BigInteger fingerId = (this.id.add(new BigInteger("2").pow(i))).mod(new BigInteger("2").pow(M));
    fingerTable[i] = findSuccessor(fingerId);
  }
}
```

NodeReference

This class has the respective methods:

- ***findSuccessor(BigInteger id)***
- ***notify(NodeReference n)***
- ***getPredecessor()***
- ***hasFailed()***

In all of them, a connection with the ***NodeReference***, using the IP and Port mentioned on its fields, is made. After the connection are established, a message is sent. Respectively to the method mentioned above, the methods are as follow:

- **CHORD FINDSUCCESSOR <ID>**
- **CHORD NOTIFY <IP> <PORT>**
- **CHORD GETPREDECESSOR**
- No message. Just checks if the Peer is still available.

In the ***findSuccessor()*** and ***getPredecessor()*** methods, the response is given in the following format:

- **CHORD NODE <IP> <PORT>**

AsynchronousSocketChannel

As mentioned in the previous section, SSLManager, used for secure TCP communication using TLS, uses the AsynchronousSocketChannel class for asynchronous non blocking I/O, as well as Threads for SSLEngine related task execution.This allows for an optimized use of the processor using concurrency to perform multiple tasks at the same time, thus allowing a large amount of simultaneous communications.

Fault-tolerance

Fault tolerance is the property of a system that allows it to continue operating even after the failure of one or more components. In our case, the most common fault in our system is the disconnection of a peer from the system. This fault leads to several smaller problems. For example, while the peer is offline, all chunks of files stored in it are not accessible, stopping the restoring of those files. It is also a problem when a file is deleted from the system while the peer is offline, as it will not follow the protocol and delete all of the chunks of the deleted file.

As to resolve such issues, we have implemented redundancy in our system, which allows the system to continue working even after failure. Due to our design with Chord a Peer can leave the System and the system still operates just fine, of course the chunks stored in it are lost though, also to allow simpler communications, if a Peer joins after files have been backed, we implemented the ***PeerMethods.giveChunks()*** method that given an entry of a new Node into the Chord Ring, this method checks the Storage to see if there are any chunks that belong to the newly added Node. In other words, it checks if the chunk ID isn't between the new predecessor and the node. If it isn't, it means the chunk actually belongs to the new predecessor.

Conclusion

This project was essential for our understanding of the functioning of a distributed system, using several methods studied for greater efficiency and scalability of the product. This new knowledge is useful for us to better understand the digital world around us and will play an important role in our future when we want to investigate more about this area of computing.

As a final note we would like to leave it written that, despite potential flaws that might come from misuse of our application, we view our project as a success, following the specification presented to us.

References

Oracle's JSSE Reference Guide

Pitt, E., 2010. *Fundamental Networking In Java*. [New York]: Springer.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. and Balakrishnan, H., 2001. Chord. *ACM SIGCOMM Computer Communication Review*, 31(4), pp.149-160.

Oracle's RMI Hello World

Barbara Liskov, Practical uses of synchronized clocks in distributed systems