U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Backup Service for the Internet

Sistemas Distribuídos

T6G22: Diogo Nunes (up201808546), Diogo Santos (up201806878),
Jéssica Nascimento (up201806723) e João Vítor Fernandes (up201806724)

Overview

Our project has 4 main protocols (services): **Backup, Delete, Restore and Reclaim**. To implement those, we used **TCP** (using thread-based concurrency) and **Chord** (decentralized design). With Chord, we guarantee **scalability**; however, we did not implement fault tolerance.

Moreover, every time we needed to access files (read and write) we used Asynchronous File Channel (**Java NIO**).

T6G22: Diogo Nunes (up201808546), Diogo Santos (up201806878),
Jéssica Nascimento (up201806723) e João Vítor Fernandes (up201806724)

Protocols

Our **backup** implementation starts with the command: `test.sh <id_peer> BACKUP <file_path> <rep_degree>`, where the peer backup function is invoked by the client (using RMI).

Then, the peer initiates a thread that reads from the file at most 64k bytes (chunk size) and asks Chord which are the available peers to receive that chunk (taking into account the free space).

After that, it sends a lookup to obtain those peers' addresses and tries to send to all of them the chunk (PUTCHUNK message) to the MDBThread. When one peer receives this message, if it can store the chunk, replies with a STORED message, otherwise replies with an empty message. Each peer knows the peers that have stored those.

```
private void initiateBackup() throws Exception {
    int size = this.fileManager.getFileSize();
    String fileId = this.fileManager.getFileId();

    // Verifies if the file was already backedup
    if (this.peer.isBackedUp(fileId)) {
        System.out.println("The file: " + this.pathName + " was already backedup by this peer");
        return;
    }

    // Tells FilesystemManager that he started a backup of a file, if some chunk was saved
    this.peer.initiateBackup(fileId, this.pathName, this.replicationDegree);

    // each chunk should have a maximum size of 64K bytes
    int CHUNK_SIZE = 64000;
    int numChunk = 0;
    int totalChunks = size / CHUNK_SIZE;
    int finalChunkSize = size % CHUNK_SIZE;
    boolean somePeerBackup = false;

    while (numChunk != totalChunks) {
        // Reads the content of a chunk
        byte[] content = FileUtils.readFile(this.pathName, CHUNK_SIZE, offset: CHUNK_SIZE * numChunk);
        if (new BackupChunk(this.peer, fileId, content, numChunk++, this.replicationDegree).start())
            somePeerBackup = true;

        System.out.println("\n------------------------\n");
    }

    // Reads the content of last chunk
    byte[] content = FileUtils.readFile(this.pathName, finalChunkSize, offset: CHUNK_SIZE * numChunk);
    if (new BackupChunk(this.peer, fileId, content, numChunk, this.replicationDegree).start())
        somePeerBackup = true;

    // Tells FilesystemManager that he started a backup of a file, if some chunk was saved
    if (!somePeerBackup) {
        this.peer.removeBackup(fileId);
        System.out.println("None of the chunks of the file was saved!");
    }
}
```

BackupThread (54-93)

The **delete** implementation works in a similar way: the client invokes the peer delete function and the peer initiates a thread responsible for this service (again, using RMI): **test.sh <id_peer> DELETE <file_path>**.

This thread starts to verify if the file we want to delete was backed up before and if so it sends a DELETE message to the MCThread of all peers that have chunk backups of that file. The receiver peer of that message goes to the directories and deletes all chunk backups it has from that file.

```
private void initiateDelete() throws Exception {

    // Verifies if the file was already backedup
    if (!this.peer.isBackedUp(this.fileManager.getFileId())) {
        System.out.println("The file: " + this.pathName + " was not backedup by this peer");
        return;
    }

    this.peer.removeBackup(this.fileManager.getFileId());

    Set<Integer> peersBackedUp = this.peer.getPeersBackedupFiles(this.fileManager.getFileId());
    System.out.println("Peers Backed Up: " + peersBackedUp);

    if (peersBackedUp.isEmpty()) {
        System.out.println("No peers have chunks of this file");
        return;
    }

    // Get peer info
    HashMap<Integer, String> peerInfo = new HashMap<>();
    for (int peer: peersBackedUp) {
        String lookupResponse = this.peer.lookup(peer);
        String[] parts = lookupResponse.split( regex: " ");
        peerInfo.put(peer, this.peer.getPeerInfo(parts[2], Integer.parseInt(parts[3])));
    }

    byte[] message = ChannelMessageBuilder.buildDELETEmessage(this.peer.getProtocolVersion(), this.peer.getChordId(), this.fileManager.getFileId());

    for (int peer: peerInfo.keySet()) {
        String[] split = peerInfo.get(peer).split( regex: " ");
        String address = split[1];
        int port = Integer.parseInt(split[2]);

        System.out.println("Sending DELETE message to peer: " + peer);
        ChannelUtils.sendMessage(message, address, port);
    }
}
```

DeleteThread (45-81)

One more time, the **restore** protocol starts with the client invoking the peer restore function, using RMI: **test.sh <id_peer> RESTORE <file_path>**.

The peer checks if it has done the backup of that file before and if so goes for it: verify which peers have chunk backups of that file, send a GETCHUNK message to the MCThread of them. If no errors are found, the peers with the chunks reply with a CHUNK message with the file content, otherwise reply with an empty message.

The peer responsible for this service, as long as it receives CHUNK messages, starts to save the content to a new file with the same name as the original one.

```java
public void initiateRestore() throws Exception {
    String fileId = this.fileManager.getFileId();
    int chunkN = 0, length;

    String filePathToRestore = "../../dirs/" + this.peer.getPeerId() + "/restores/" + this.fileManager.getFileNameFromPathName(this.pathName);
    File file = new File(filePathToRestore);

    if(file.exists()) {
        System.out.println(ConsoleColors.YELLOW + "-> Restore of the specified file already done!" + ConsoleColors.RESET);
        return;
    }

    file.createNewFile();
    FileWriter fileWriter = new FileWriter(file);

    System.out.println("-> Restore will be made into: " + filePathToRestore);

    do {
        System.out.println("Chunk: " + chunkN);

        byte[] body=this.sendGETCHUNKmessage(fileId, chunkN);

        if(body==null)
            length = -1;
        else{
            // appends onto file
            FileUtils.writeFile(filePathToRestore, body, offset: chunkN*64000);
            length = body.length;
        }
        chunkN++;
    } while (length == 64000); // it ends only when the size of the chunk is different from 64KB

    fileWriter.close();

    if (length == -1) {
        file.delete(); // deletes the restored file because didn't obtained an answer for some chunk
        System.out.println(ConsoleColors.YELLOW + "-> Couldn't restore the chunk: " + (chunkN - 1) + ". Deleting restored file..." + ConsoleColors.RESET);
    }
}
```

RestoreThread (56-97)

The **reclaim** protocol, as well as the other protocols, starts with the client invoking the peer reclaim function (using RMI) that initiates a thread responsible for the restore service: **`test.sh <id_peer> RECLAIM <max_space_size>`**.

It starts to save the new maximum size the peer will use to save chunks. Then, if the actual size used does not exceed the maximum size, the reclaim protocol is done; otherwise, it has to select and delete some chunks, until the actual size doesn't exceed the maximum size.

The selection function selects the smallest chunk that is larger than a given size; if it's not possible, it selects any chunk. Deleting this chunk is not enough, we also have to notify the peer responsible for the backup of the file that contains this chunk, sending a REMOVED message to the MCThread, so that it can initiate the backup of the deleted chunk.

```
private boolean deleteChunk(String chunkId) throws Exception {
    // separates fileId and chunkN
    System.out.println("CHUNK ID: " + chunkId);
    if (chunkId != null) {
        int num = chunkId.indexOf("$");
        String fileId = chunkId.substring(0, num);
        int chunkNo = Integer.parseInt(chunkId.substring(num + 1));

        // Deletes the chunk
        FileUtils.deleteFile("../../dirs/" + this.peer.getPeerId() + "/chunks/" + chunkId);

        // Tells fileSystemManager that he deleted chunk
        int initiatorId = this.peer.removeStoredChunk(fileId, chunkNo);

        this.tellInitiator(fileId, chunkNo, initiatorId);

        return true;
    }
    else {
        System.out.println("Could not conclude RECLAIM");
        return false;
    }
}
```

ReclaimThread (88-110)

```java
private void initiateReclaim() throws Exception {
    this.peer.setMaxSpace(Integer.parseInt(this.spaceDisk));

    // Reclaims all space so it needs to delete all chunks
    if (Integer.parseInt(this.spaceDisk) == 0) this.deleteAll();
    else {
        int usedSpace = this.peer.getUsedSpace();
        int space = usedSpace - Integer.parseInt(this.spaceDisk);

        System.out.println("UsedSpace: " + usedSpace);
        System.out.println("MaxSpace: " + this.spaceDisk);

        // It only needs to free some space
        if (space > 0) this.deleteSome(space);
        else System.out.println("No need to delete anything.");
    }
}

private void deleteAll() throws Exception {
    List<String> chunks = this.peer.getAllChunksStored();

    for (String chunkId: chunks) {
        if (!this.deleteChunk(chunkId)) return;
    }
}

private void deleteSome(int space) throws Exception {
    int sizeMinChunk = space % CHUNK_SIZE;
    int numMaxChunks = space / CHUNK_SIZE;

    System.out.println("Space: " + space);
    System.out.println("Rest: " + sizeMinChunk);
    System.out.println("DivInt: " + numMaxChunks);

    for (int i = 0; i < numMaxChunks; i++) {
        if (!this.deleteChunk(this.peer.selectChunkBySize(CHUNK_SIZE))) return;
    }

    if (sizeMinChunk > 0)
        this.deleteChunk(this.peer.selectChunkBySize(sizeMinChunk));
}
```

ReclaimThread (46-86)

Our **Remote Interface** sends the requests to the peers, so they can initialize the implemented protocols.

Our **message format** follows the first assignment model, and if some error occurs, the message would be empty.

- To the backup protocol, we used 2 kind of messages:
  - **`<Version> PUTCHUNK <SenderId> <FileId> <ChunkNo> <ReplicationDeg> <ChunkSize> <CRLF><CRLF><Body>`**: Sent by the initiator peer to the peers that will receive the backup.
  - **`<Version> STORED <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>`**: Sent from the peers that made the chunks backup to the initiator peer.

- To the delete protocol, we used only 1 kind of message:
  - **`<Version> DELETE <SenderId> <FileId> <CRLF><CRLF>`**: Sent by the initiator peer to all the peers that made backup of a chunk of the file we pretend to delete.

- To the restore protocol, we used 2 kind of messages:
  - **`<Version> GETCHUNK <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>`**: Sent by the initiator peer to the peers that made backup of a chunk.
  - **`<Version> CHUNK <SenderId> <FileId> <ChunkNo> <ChunkSize> <CRLF><CRLF><Body>`**: Sent by the peers that made a backup of a chunk to the initiator peer.

- To the reclaim protocol, we used only 1 kind of message:
  - **`<Version> REMOVED <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>`**: Sent by the initiator peer to the peer that initiated the backup of the chunk.

- To Chord we also needed to use many messages:
  - **`FINDSPOT_REQUEST <id>`**: Request sent by a peer that wants to join Chord, specifying its id.
  - **`FINDSPOT_RESPONSE <successor_id> <successor_address> <successor_port> <predecessor_id> <predecessor_address> <predecessor_port>`**: Reply sent with the location where it will be located in Chord, specifying both the predecessor and successor.
  - **`GETSUCCESSOR`**: Message sent in order to get the successor.
  - **`PREDECESSOR <predecessor_id> <predecessor_address> <predecessor_port>`**: Message that contains a predecessor node.
  - **`UPDATE_SUCCESSOR <node_id> <node_address> <node_port>`**: Message sent for a node to update its successor.
  - **`UPDATE_PREDECESSOR <node_id> <node_address> <node_port>`**: Message sent for a node to update its predecessor.
  - **`UPDATE_FINGERTABLE <node_id>`**: Message sent for a node to update its fingertable, specifying the id of which peer initiated the updates so that we can stop when we reach it.
  - **`LOOKUP_REQUEST <node_id>`**: Request sent to make a lookup of a peer.
  - **`LOOKUP_RESPONSE <node_id> <node_address> <node_port>`**: Reply from lookup message, containing the id, address and port of the current peer.
  - **`UPDATESTATE <chordNodeState> <node_id>`**: Message sent to a node to update the current state (possible states: “CREATING”, “UPDATING”, “READY”).
  - **`GETAVAILABLEPEERS <id> <size>`**: Message sent to get the available peers to receive a chunk with a given size. <id> represents the peer id of the request invocation, so that it can stop when it's done.
  - **`GETPEERINFO`**: Message sent to a peer to get its information.
  - **`<peer_chord_id> <peer_McAddress> <peer_McPort> <peer_MdbAddress> <peer_MdbPort> <peer_MdrAddress> <peer_MdrPort>`**: Message sent in response to the previous one containing information about the peer.
  - **`FREEID <id>`**: Message sent to know if a given Chord id is still free.
  - **`<message>`**: Message sent in response to the previous one, and it can only be one of the following: “FALSE” if it is not free; “TRUE” otherwise.

Concurrency design

The concurrency design we implemented supports the execution of multiple subprotocols. Threads are created to communicate with other peers. This communication between peers happens through mcThread and mdbThread.

In order to execute each subprotocol, receive requests, and use Chord we created **ThreadPools** so that information doesn't get lost when trying to execute more than n subprotocols. This happens because the request is being held in a queue until there's a free thread to execute that request.

```
private void initiateThreads() {
    this.executorService = Executors.newFixedThreadPool(6);

    System.out.println("Initiating Threads for peer " + this.peerId + "...");
    new Thread(this.mcThread).start();
    new Thread(this.mdbThread).start();
    System.out.println(ConsoleColors.GREEN + "Done.\n" + ConsoleColors.RESET);
}
```

Peer.java (214-221)

```
public Chord(String chordAddress, int chordPort, String askAddress, int askPort, Peer peer) {
    this.executorService = Executors.newFixedThreadPool(2);

    if (askAddress.equals("null") || askPort < 0) this.createFirstNode(chordAddress, chordPort, peer);
    else this.createNode(chordAddress, chordPort, askAddress, askPort, peer);
}
```

Chord.java (39-44)

Regarding the reading and writing of a file, we use Asynchronous File Channel (**Java NIO**) which is associated with thread pools.

```
public static byte[] readFile(String fileName, int bufferSize, int offset) throws Exception {
    AsynchronousFileChannel reader = AsynchronousFileChannel.open(Paths.get(fileName), StandardOpenOption.READ);

    ByteBuffer bufferRead = ByteBuffer.allocate(bufferSize);

    Future<Integer> operation = reader.read(bufferRead, offset);
    while (!operation.isDone());

    reader.close();

    return bufferRead.array();
}

public static void writeFile(String fileName, byte[] content, int offset) throws Exception {
    AsynchronousFileChannel writer = AsynchronousFileChannel.open(Paths.get(fileName), StandardOpenOption.WRITE);

    ByteBuffer buffferWriter = ByteBuffer.wrap(content);

    Future<Integer> operation1 = writer.write(buffferWriter, offset);
    while (!operation1.isDone());

    writer.close();
}
```

FileUtils.java (12-34)

Scalability

In order to achieve a scalable architecture, we decided to implement a Chord. It makes use of thread-pools, to facilitate concurrency.

Let's now explain how we implemented Chord.

Whenever a node wants to join the Chord ring, we generate an id using a hash function that uses its address, port, and the number of the attempt to generate the id. After that, two scenarios can happen:

- it's the first node, and therefore it joins Chord directly (no extra steps needed);
- it's not the first node, and the user needs to pass 2 extra parameters (address and port) so that it can join Chord. In this case, the node that wants to join Chord sends a message(*FINDSUCCESSOR*) to the other node. This message goes through the Chord nodes until it finds the place where the node fits, returning the address and port of both successor and predecessor nodes.

```
public synchronized String findSpot(String request) {
    int id = Integer.parseInt(request.split(" ")[1]);

    if (ChordUtils.isSuccessor(id, this.chordNode))
        return ChordMessageBuilder.buildFINDSPOTresponse(this.chordNode.getSelfInfo(), this.chordNode.getPredecessor());
    else if (ChordUtils.isPredecessor(id, this.chordNode))
        return ChordMessageBuilder.buildFINDSPOTresponse(this.chordNode.getSuccessor(), this.chordNode.getSelfInfo());
    else
        return ChordMessageSender.sendMessageAndWait(request, this.chordNode.getSuccessor().getAddress(),
         this.chordNode.getSuccessor().getPort());

}
```

Chord.java (137-144)

After that, we need that all Chord nodes update their FingerTables.

```
public void updateFingerTable() {
    this.fingerTable.clear();

    this.fingerTable.add(this.successor);
    for (int i = 1; i < this.m; i++) {
        int nodeToFindSuccessor = Math.floorMod(this.selfInfo.getId() + (int) Math.pow(2, i), (int) Math.pow(2, this.m));

        String response = ChordMessageSender.sendMessageAndWait(ChordMessageBuilder.buildFINDSPOTrequest(nodeToFindSuccessor),
         this.successor.getAddress(), this.successor.getPort());
        this.fingerTable.add(ChordUtils.getSuccessorFromFINDSPOTResponse(response));
    }
}
```

ChordNode.java (113-123)

These are structures used to facilitate the search, achieving a search time of O(log(N)), that is, a very good time.
This structure was implemented as a list of m elements of type ChordNodeInfo that keep the id, address, and port of each one.

```
private int m;
private List<ChordNodeInfo> fingerTable;
```

ChordNode.java (22-23)