
# Project 1 - Distributed Backup Service

Distributed Systems, MIEIC

11/04/2021

André Filipe Meireles do Nascimento - up201806461
Gustavo Sena Mendes - up201806078

Turma 4 Grupo 04

# Contents

# Introduction

In this project we developed 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.

## Project Structure

Our project is divided into 4 folders:

- doc/: Contains the report and the readme file.
- files/: Contains files to be used in testing.
- scripts/: Contains bash scripts to help compile, test and execute the program in different operating systems.
- src/: Contains the source code developed during this project.

Regarding the src/ folder, we organized it in packages, each of which are responsible for fulcral parts of the project:

- channel: Responsible for communication and request dispatching.
- messages: Contains message classes used in peer communication.
- peer: Contains the main methods for starting sub protocol actions and the peer storage class.
- storage: Contains classes responsible for information representation of chunks and stored files.
- task: Classes responsible for processing the different messages received by the peer.
- test: Responsible for sending the protocols to be executed to each peer using RMI (TestApp).
- utils: Contains useful methods and constants used by other packages.
- workers: Classes responsible for executing the protocols called by the RMI.

## Concurrency Design

To ensure concurrency with the different subprotocols, different strategies were used.

Firstly, for implementing the Threads themselves each peer has three ThreadPools, one for each message channel. Each ThreadPool's capacity is decided based on the usage of its respective channel: since the control channel is the most used it has a pool of 128 threads, whereas the backup and restore's ThreadPool only has 32 threads. Each peer also has 3 threads running each Multicast channel (**src.channel.MulticastChannel** class, that extends MulticastSocket and implements Runnable), and has a ScheduledThreadPoolExecutor used by the protocols to schedule tasks with 256 threads.

To save the state of a peer (all the sent and stored chunks, backed up and deleted files) the **src.peer.PeerStorage** class implements Serializable, and is saved periodically (using a scheduler that runs on a 1 minute loop) and also when the system receives a SIGINT signal.

```
// Save peer storage periodically (every minute)
peer.scheduler.scheduleAtFixedRate(new Thread(peer.storage::saveState), initialDelay: 1, period: 1, TimeUnit.MINUTES);
Runtime.getRuntime().addShutdownHook(new Thread(peer.storage::saveState));
```

While using various threads to run the subprotocols, the same objects are accessed in different threads so they are stored using a ConcurrentHashMap in **src.peer.PeerStorage**, instead of an HashMap. This data structure is a thread-safe structure and is optimized for multi-threading.

The space occupied in a peer is stored in a variable on **src.peer.PeerStorage** and is incremented or decremented using synchronized functions so it remains consistent throughout the peers "life".

As for when a protocol request is sent to a peer by the TestApp (using RMI), the corresponding peer function is called and then the protocol workers (**src.workers** package) are submitted to the corresponding ThreadPool.

When a packet is received by one of the Multicast channels of the peer, the packet is processed, the corresponding message type is created and its Task (a Runnable function that processes the message according to the protocol it belongs to) is submitted to the thread pool of the corresponding Multicast channel. The different messages classes are in the **src.messages** package, and its tasks in the **src.tasks** package.

```
public void run() {
    while (true) {
        byte[] buf = new byte[65000];
        DatagramPacket packet = new DatagramPacket(buf, buf.length);

        try {
            this.receive(packet);

            // Create respective message from received packet
            Message message = Message.create(packet);

            // Process message task if it was not sent by this peer
            if (!message.messageOwner(this.peer.getId())) {
                message.submitTask(this.peer);
            }

        } catch (Exception e) {
            System.err.println("Error receiving DatagramPacket: " + e.getMessage());
        }
    }
}
```

Instead of Thread.sleep(), Java.util.concurrent.ScheduledThreadPoolExecutor is used to delay the execution of the threads when waiting is needed, which allows us to schedule a "timeout" handler, without using any thread before the timeout expires. This proved useful, for example, when sending the Putchunk message at max 5 times, because instead of sleeping between retransmissions we just schedule a new Runnable task, which can be seen in **src.workers.BackupChunkWorker**:

```
    PutChunkMessage putChunkMessage = new PutChunkMessage(this.peer, this.chunk);
    // Try to send PUTCHUNK message max 5 times or until Replication degree is met
    this.peer.sendBackupMessage(putChunkMessage);
    this.peer.getScheduler().schedule(() -> this.sendPutchunkMessage(putChunkMessage,  attempt: 1),  delay: 1000, TimeUnit.MILLISECONDS);
}

private void sendPutchunkMessage(PutChunkMessage putChunkMessage, int attempt) {
    if (attempt < Utils.MAX_5_ATTEMPTS && this.chunk.needsReplication()) {
        this.peer.sendBackupMessage(putChunkMessage);

        int finalAttempt = attempt+1;
        this.peer.getScheduler().schedule(() -> this.sendPutchunkMessage(putChunkMessage, finalAttempt),  delay: (long) Math.pow(2, attempt) * 1000,
    } else {
        this.chunk.clearBody();
    }
}
```

Despite not being specified, since we are using UDP to send messages, all other subprotocols send the message more times just like in the backup, but instead to a max of 3 times. We decided to implement it this way to guarantee that the request messages are received, since UDP is not reliable. These “improvements” can be seen in **src.workers.DeleteFileWorker** (sends DELETE message max 3 times), **src.workers.DeleteChunkWorker** (sends REMOVED message max 3 times), and **src.workers.RestoreChunkWorker** (sends GETCHUNK max 3 times or until it has received the chunk data).

Another example is when a peer receives a PUTCHUNK message it should sleep for a random amount (between 0 and 400ms), before sending a STORED message to other peers. We schedule a task after that random amount of time, which can be seen in **src.tasks.PutchunkTask**:

```
    // Just schedule randomly between 0-400ms if default
    else {
        this.peer.getScheduler().schedule(() -> this.storeChunk(chunk), Utils.getRandom( max: 400), TimeUnit.MILLISECONDS);
    }
}

private void storeChunk(Chunk chunk) {
    // Check if peer has enough space to store chunk
    if (!this.peer.getStorage().hasEnoughSpace(chunk.getSize())) {
        System.err.println("[BACKUP] Not enough space to store chunk " + chunk.getUniqueId());
        this.peer.getStorage().removeStoredChunk(chunk.getUniqueId());
        return;
    }

    try {
        // Store chunk in file
        this.peer.getStorage().storeChunk(chunk, this.message.getBody());

        // Acknowledge that chunk is stored and add it to peer ack Set
        chunk.setStoredLocally(true);
        chunk.addPeerStoring(this.peer.getId());

        // Send stored message
        StoredMessage message = new StoredMessage(this.peer.getProtocolVersion(), this.peer.getId(), chunk.getFileId(), chunk.getChunkNo());
        this.peer.sendControlMessage(message);

        //System.out.println(String.format("Sent STORED: chunk no: %d ; file: %s", chunk.getChunkNo(), chunk.getFileId()));
    } catch (IOException e) {
        System.err.printf("Failed to store chunk %s\n", chunk.getUniqueId());
    }
}
```

# Enhancements

## Chunk backup subprotocol

Since a peer stores the chunk upon receiving the PUTCHUNK message, regardless of the current replication degree of that chunk, every working peer will store all received chunks. This scheme will deplete the backup space rather rapidly because the actual replication degree of the chunks will be equal to the number of online peers, instead of the given desired replication degree.

In order to solve this problem, the enhanced peer only stores the chunk if the desired replication degree has not yet been achieved. The enhanced peer checks, after receiving the PUTCHUNK and waiting the random time uniformly distributed between 0 and 400 ms, if the desired replication degree of the chunk is already achieved (this is easily done since the **Chunk** class has a Set with all the ids of the peers that are storing that chunk, in other words, the ids of the peers that sent a STORED message for that chunk), and in that case, the peer just ignores the received PUTCHUNK message. Otherwise, the peer sends the STORED message normally.

The changes made to implement this enhancement can be seen in **src.worker.BackupChunkWorker** and **src.tasks.PutchunkTask**.

## Chunk restore subprotocol

In the restore subprotocol only the peer that sends the GETCHUNK messages needs to receive the CHUNK messages, but we are using a multicast channel so all peers will receive the CHUNK message.

To avoid sending large chunks of data to peers that don't need the CHUNK message, the enhanced version sends the chunks through TCP directly to the peer that sent the GETCHUNK. In order to implement this enhancement and make it interoperable we made some changes to the CHUNK message. The enhanced peer that initiated the restore subprotocol sends the "default" GETCHUNK messages with the enhanced protocol version in the header, then the enhanced peer that receives the GETCHUNK will verify the message version, and in case it is an enhanced message it will create a socket, send a modified CHUNK message whose *body* contains the IP address and Port number of the soccket, in the format "address:port", instead of the the chunk data, wait for the connection to the accepted and then write the chunk data to the socket outputstream. When the initiator peer receives this modified CHUNK message it will create a socket with the received address and port, and read the chunk data from the inputstream of the socket. When the other peers also receive this modified message, since the header is the same as the "default", they will know that the chunk with the chunkNo of the message has already been transmitted and that they don't need to send it. This way interoperability is maintained, and in the enhanced peers the chunk data will only be sent to the restore initiator peer.

The changes made to implement this enhancement can be seen in **src.tasks.GetchunkTask** and **src.tasks.ChunkTask**.

## File deletion subprotocol

If a Peer that backs up some chunks of the file is not running at the time the initiator peer sends a DELETE message for that file, the space used by these chunks will never be reclaimed. Our solution for this problem required to create a new message, the WakeyWakey message, with the following format: <Version> WakeyWakey <SenderId> <CRLF><CRLF><Body>. The *body* of this message contains the IP address and Port number, in the format "address:port", because we are using TCP to receive the DELETE messages, as explained in the next paragraph.

When an enhanced peer is initialized a **WakeyWorker** is submitted, creating a server socket for a connection and sending an WakeyWakey message through the MulticastControl channel to alert other peers that he is running and ready to receive information in the given address and port. When the other enhanced peers receive this message they will search their deleted files Map, and if the map contains the WakeyWakey senderId then a new socket is created in the received address and port and the DELETE message is sent to the outputstream of the socket, for each file that the WakeyWakey sender had chunks backed up.

We decided to implement this enhancement using TCP instead of UDP because this way only the peer that was initialized receives the DELETE messages instead of all the peers that are running at the time.

The classes made to implement this enhancement are **src.message.WakeyMessage**, **src.worker.WakeyWorker** and **src.tasks.WakeyTask**.