Distributed Systems - Project 1

T5G02

Eduardo Correia
up201806433@fe.up.pt

Ana Inês Barros
up201806593@fe.up.pt

April 12, 2021

Chapter 1

Introduction

This report was elaborated for the *Distributed Systems* (SDIS).

It serves as a complement for the course unit's first project and aims to describe the specification of the enhancements made to the several sub-protocols and how the concurrent execution of sub-protocols was achieved.

Chapter 2

Enhancements

2.1 Restore sub-protocol

If the files we're restoring are too large, this protocol may not be desirable, for the following reasons:

Only one peer needs to receive the chunks that correspond to the file that's being restored, however, we are using a multicast channel to send them all, so even peers that didn't request the restore of that file are receiving its chunks.

To eliminate this problem, we opted to use TCP, since it provides a direct and safe connection and it was required to obtain full credit.

Initiator Peer:

```
socket = new ServerSocket(0);
socket.setSoTimeout(1000);

// Send Getchunk Message
message = new GetChunkEnhancedMsg(version, initiator_peer.id, file_id, 0,
↪  socket.getLocalPort());
control_channel.send(message.getBytes(null, 0));

// Open socket
Socket clientSocket = socket.accept();
DataInputStream inputStream = new DataInputStream(clientSocket.getInputStream());

// Receive message
int msg_len = inputStream.read(msg);

// Close connection
clientSocket.close();
socket.close();
```

Peer who has chunk:

```
// Open connection
socket = new Socket("localhost", get_chunk_msg.getPort());

// Send message
DataOutputStream outputStream = new DataOutputStream(socket.getOutputStream());
outputStream.write(message_bytes);
outputStream.flush();
socket.close()
```

For this enhancement, the initiator peer opens a TCP server to receive data directly from the peer instead of the chunk being sent to the multicast channel. Port 0 is passed to the constructor of the server socket so that the socket uses an available port.

We had to create a different `GETCHUNK` message for the enhanced version of this protocol where we add the port the peer needs to use in the constructor of the socket in order to open the connection with the initiator peer. If the connection is established, the peer sends the `CHUNK` message to the initiator peer using a `DataOutputStream` and the connections is closed.

Note: Snippets of the code are simplified in order to focus on the TCP/IP connection

2.2 Delete sub-protocol

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.

```
// Send WOKEUP message
WokeUpMsg woke_msg = new WokeUpMsg(peer.id);
peer.control_channel.send(woke_msg.getBytes(null, 0));

// Set of delete file's ids
private Set<String> deleted_files;
deleted_files = new ConcurrentHashMap<String, String>().newKeySet();

// Add file_id to set when asked to delete
deleted_files.add(file.getId());

// Upon receiving WOKEUP message start delete protocol for each set entry
for(String file_id: deleted_files)
    pool.execute(new Delete(peer, version, file_id, mc_channel));
```

To solve this problem, we created a new message `WOKEUP`.

This message is sent when a peer starts running and is sent to the MC channel. Its purpose is to signal the beginning of the execution of a peer.

With the enhanced Delete protocol, the peer stores the file IDs of the file it has asked to delete. When a `WOKEUP` message is received, the `WokeUpMessageHandler` proceeds to start the delete sub-protocol for each of the file ids of files that a peer asked to delete.

By doing this, if a peer did not receive the `DELETE` messages while it was not running, then it will receive when it wakes up. Furthermore, if a peer asks to backup a previously deleted file again, the peer won't send more delete messages regarding that file when it receives a `WOKEUP` message.

Note: The code is simplified in order to focus on the enhancement.

Chapter 3

Concurrency Design

Regarding concurrency, we opted for following most of the hints proposed.

3.1 Threads

For each multicast channel, there is a thread running which is responsible for receiving packets.

```
@Override
public void run() {
    if (start() != 0)
        return;

    running = true;

    while (running) {
        try {
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet); // Receive packet
            parseMessage(packet.getData(), packet.getLength());
        }

        catch (IOException e) {
            System.err.println("No more messages.");
            stop();
        }
    }
}
```

For the processing of the messages received, we created message handlers for each message type, and for each of these handlers there is a thread.

This makes it possible to process different messages at the same time. Each channel has its own `CachedThreadPool` that will execute each handler for each message.

An example follows below:

```
public class MDR_Channel extends Channel {
    // etc ...

    @Override
    protected void parseMessage(byte[] msg, int msg_len) {
        byte[] header = Message.getHeaderBytes(msg);
        String[] header_fields = Message.getHeaderFields(msg);
```

```
        int sender_id = Integer.parseInt(header_fields[Fields.SENDER_ID.ordinal()]);
        String type = header_fields[Fields.MSG_TYPE.ordinal()];

        // Ignore message from itself
        if (sender_id == peer.id) return;

        if (type.equals("CHUNK")) {
            ChunkMessage chunk_msg = new ChunkMessage(header_fields);
            byte[] body = Message.getBodyBytes(msg, msg_len, header.length);

            // Log
            System.out.printf("< Peer %d received: %s\n", peer.id,
            ↪  chunk_msg.toString());

            // Chunk message handler
            pool.execute(new ChunkMessageHandler(chunk_msg, peer, body));
        }
    }
}

public class DeleteMessageHandler extends MessageHandler{
    private final Peer peer;

    public DeleteMessageHandler(DeleteMessage delete_msg, Peer peer){
        super(delete_msg.getFile_id(), peer.storage);
        this.peer = peer;
    }

    @Override
    public void run() {
        // Deletes all chunks from file
        deleteAllChunksFromFile(file_id);
        peer.saveStorage(); // Update storage
    }
    // etc
}
```

In order to make it possible for a peer to execute multiple sub-protocols at the same time (backup, restore, etc...), each sub-protocol runs on a thread and a peer has its own `CachedThreadPool`.

This pool is responsible for executing the multicast channels threads and the sub-protocol threads.

An example follows below:

```
public class Peer implements RMI {
public ExecutorService pool;

    public Peer(){
        pool = Executors.newCachedThreadPool();

        // Start listening on channels
        peer.pool.execute(peer.backup_channel);
        peer.pool.execute(peer.control_channel);
        peer.pool.execute(peer.restore_channel);
    }
```

```
    @Override
    public void restoreFile(String file_path) {
        BackedUpFile file = storage.getFileInfo(file_path);

        if (file == null) {
            System.out.println("File to restore needs to be backed up first.
            ↪  Aborting...");
            return;
        }

        Runnable task;
        if(version.equals("2.0"))
            task = new RestoreEnhanced(this, version, file.getPath(), file.getId(),
            ↪  file.getNumberOfChunks(), restore_channel, control_channel);
        else
            task = new Restore(this, version, file.getPath(), file.getId(),
            ↪  file.getNumberOfChunks(), restore_channel, control_channel);
        pool.execute(task);
    }
}
```

To avoid sleeping, we used the class `java.util.concurrent.ScheduledThreadPoolExecutor`, which allowed us to schedule a "timeout" handler, without using any thread before the timeout expires.

```
for(int i = 0; i < MAX_TRIES; i++) {
    scheduledPool.schedule(() -> {
        // Send message
        control_channel.send(message_bytes);
    }, 400 * i, TimeUnit.MILLISECONDS);
}
```

In conclusion, although threads allow us to process many messages and to run many sub-protocols at the same time, thread objects use a significant amount of memory.

Additionally, in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead. By using worker threads (thread pools) we minimized the overhead due to thread creation. We opted for using a `CachedThreadPool` because it creates new threads as needed, but will reuse previously constructed threads when they are available.

We also used synchronized methods to ensure that only one thread could execute them at a time. Example of a synchronized method:

```
public synchronized int getPerceivedRP(String file_path, int chunk_no) {
    BackedUpFile file = backed_up_files.get(file_path);
    if (file != null)
        return file.getPerceivedRP(chunk_no);

    Chunk chunk = stored_chunks.get(file_path);
    if(chunk != null)
        return chunk.getPerceivedRP();

    return 0;
}
```

3.2 Appropriate Data Structures

With the use of threads, a need for appropriate data structures and thread-safe and synchronized methods. By using the `java.util.concurrent` package we were able to achieve concurrency in our program.

We used a `ConcurrentHashMap` which supports full concurrency of retrievals and adjustable expected concurrency for updates.

Using the key set of a `ConcurrentHashMap`, we were also able to have a concurrent set.

We also used used `AtomicBooleans` and `AtomicLongs` from the `java.util.concurrent.atomic` package which contains a small toolkit of classes that support lock-free thread-safe programming on single variables.

The next snippet contains some examples:

```
// Atomics
public final AtomicLong max_space;
public AtomicLong used_space;

// Sets
private final Set<Observer> observers;
observers = ConcurrentHashMap.newKeySet();

// Concurrent Hash Map
private ConcurrentHashMap<Integer, Chunk> chunks;
this.chunks = new ConcurrentHashMap<>();
```

3.3 No Blocking

At some point in the development of the application, we used semaphores from `java.util.concurrent.Semaphore` package to access shared variables between threads. However, we ended up removing them to reduce blocking.

For ultimate scalability, we removed whatever blocking calls are left, in this case, the calls for file-system access.

Previously, we used `DataOutputStream` and `DataInputStream` to write to and read from files, respectively, which were replaced with `AsynchronousFileChannel`.

An example of reading a chunk using this method follows below.

```
Path path = Paths.get(chunk.getPath());
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.RE

ByteBuffer buffer = ByteBuffer.allocate(Storage.MAX_CHUNK_SIZE);

Future<Integer> operation = fileChannel.read(buffer, 0); // Read chunk

int read_bytes = operation.get();
```