JOHANNES KEPLER
UNIVERSITÄT LINZ

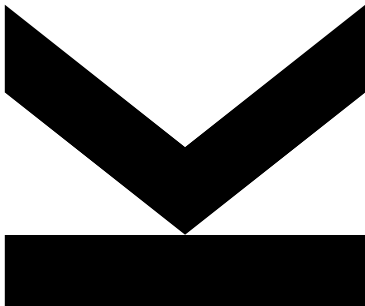
Submitted by
Osama Mohammad Khalil

Submitted at
Johannes-Kepler-Universität / Institute for System Software

Supervisor
a. Univ.-Prof. Dipl.- Ing. Dr. Herbert Prähofer

December 2022

A NETTY-BASED FILE SYNCHRONIZATION SYSTEM



Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor Program

Computer Science

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
jku.at

JOHANNES KEPLER
UNIVERSITÄT LINZ

Statutory Declaration

declare instead of an oath that I have written this bachelor thesis independently and without the help of others, that I have not used any sources or aids other than those specified, or that I have marked the passages taken literally or analogously as such. This bachelor thesis is identical to the electronically transmitted text document.

Linz, December 2022 Osama Mohammad Khalil

JOHANNES KEPLER
UNIVERSITÄT LINZ

Kurzfassung

Diese Arbeit beschreibt eine Fallstudie zur Implementierung eines Dateisynchronisationssystems mit dem Netty-Framework. Diese Art von Systemen basiert auf einem Client-Server-Modell (Anfrage/Antwort Modell). Die erste Aufgabe dieser Systeme besteht darin, Dateien vor Verlust zu schützen, indem Kopien von Dateien auf dem Server gespeichert werden. Die zweite Aufgabe besteht darin, geänderte Dateien zwischen den verbundenen Clients zu synchronisieren. Netty ist ein Java-Framework, das auf dem Java NIO-Netzwerkkommunikationsprotokoll basiert. Es ermöglicht eine asynchrone, nicht-blockierende Ein- und Ausgabe. Der asynchrone, nicht-blockierende Ansatz ist im Vergleich zu einem blockierenden Thread-basiertem Ansatz effizienter, insbesondere für Anfrage/Antwort-Modelle, bei denen die Antwortzeit, die Leistung und die Auslastung der Ressourcen sehr wichtig sind. In dieser Arbeit soll das Netty-Framework durch die Implementierung eines Dateisynchronisationssystems auf der Grundlage des Client-Server-Modells evaluiert werden. Die Arbeit soll in dieser Weise die Vorteile des nicht-blockierenden Ansatzes im Allgemeinen und des Netty Frameworks im Speziellen bei der Entwicklung solcher Systeme zeigen.

Abstract

This thesis describes a case study implementing a file synchronization system with the Netty framework. This type of systems is based on a client-server (request-response) model. The first task of these systems is to save files from loss by saving copies of files on the server storage. The second task is to synchronize modified files between the connected clients. Netty is a Java framework based on Java NIO network communication protocol. It allows asynchronous non-blocking input and output. The asynchronous non-blocking approach is more efficient compared to a blocking thread-based approach, especially for request/response models, where the response time, performance, and utilization of resources are very important. This thesis aims to evaluate the Netty framework by implementing a file synchronization system based on the client-server model. It will work out the advantages of the non-blocking approach in general and the Netty framework in particular in developing such systems.

Contents

1 Introduction **6**
1.1. Outline of this Thesis 7
2 File synchronization systems **8**
2.1. Rsync 8
2.2. Google Drive 9
2.3. Dropbox 10
3 Netty framework **12**
3.1. Netty Server 13
3.2. Netty Client 18
3.3. Result 19
4 Netty-based file synchronization systems **21**
4.1. Separation of concerns 21
4.2. Reusability 21
4.3. ByteBuf class 22
4.4. Memory management 22
5 Use Cases and Requirements **24**
5.1. Use cases 24
5.1.1. Single client-server case 24
5.1.2. Multiclient-server case 26
5.1.3. Case with conflict 27
5.2. Requirements 28
6 Implementation Description **30**
6.1. System Design 30
6.1.1. System architecture 30
6.1.2. State diagrams 31
6.2. Implementation 35
6.2.1. Main system components 35
6.2.2. Server program 56
6.2.3. Client program 57
7 Conclusion **59**
References **60**

1 Introduction

This thesis describes a case study implementing a file synchronization system with the Netty framework. Netty [6] is a Java framework based on the Java NIO network communication protocol. In distinction to previous protocols, Java NIO allows asynchronous and non-blocking input and output. Thus, the distinction between synchronous and asynchronous systems is essential for this work. A synchronous system is a system in which the tasks are executed in sequence, so task B cannot be executed until task A is finished. This is inefficient for client-server models, where the client sends requests and must wait (idle time) for the response from the server. In contrast, asynchronous is a multithreaded model that's most applicable to networking and communication systems. The asynchronous non-blocking approach does not block execution while one or more operations are in progress [10]. The asynchronous non-blocking approach is very efficient, especially for request/response models, where the response time, the performance, and the utilization of the resources are very important. In contrast, the synchronous blocking approach is not effective for these models because of waiting times, thus wasting of resources and reduction of performance.

According to its definition, Netty is an asynchronous event-driven network application framework for rapid development of maintainable high-performance protocol servers & clients [6]. Thus, the Netty framework provides an asynchronous non-blocking event-driven approach. This is an extremely convenient approach for implementing high-performance client-server (request/response) models. This is due to its ease of use and robust design that reduces the waste of resources and time and allows the developers to focus more on the logic of their applications than network configuration, as we have seen in the file synchronization system implemented in this thesis.

The goal of this thesis is to evaluate the Netty framework by implementing a file synchronization system based on the client-server model. The first task of these systems is to save files from loss by saving copies of files on the server storage. The second task is to synchronize modified files between the connected clients. In these systems, the server has to be able to serve multiple clients concurrently. Furthermore, it is also able to recognize conflicts between clients (when two or more versions of a file have to be synchronized at the same time).

1.1. Outline of this Thesis

The thesis is structured as follows:

- In Chapter 2, we will review file synchronization systems like Rsync, Google Drive, and Dropbox.
- In Chapter 3, we will learn about the Netty framework and how can it be used to create a client-server system based on a simple example.
- In Chapter 4, we will discuss which advantages the Netty framework provides.
- Chapter 5 multiple use cases of the Netty-based file synchronization system are described and requirements for its implementation are given.
- In Chapter 6, the design and the implementation of the server-based file synchronization system are described.
- The thesis concludes with a summary of the work.

2 File synchronization systems

Cloud-based file synchronization systems like Dropbox or Google Drive allow synchronizing local file content with the cloud storage. Thus, modern storage technology has shifted from a traditional offline state to cloud-based technology for some time now. Synchronization of files and keeping a history of changes are critical parts of any cloud system [1].

This section will provide an overview of some file synchronization algorithms and tools.

2.1. Rsync

According to Andrew Tridgell and Paul Mackerras [2], Rsync is an algorithm for synchronizing files between multiple devices connected via a two-way communication link. Rsync only sends those fragments that don't match by calculating a set of differences between the target files. Let's imagine this scenario. We have two files f1 and f2 and we want to update file f2 to be identical to file f1. The clear and simple way is to copy the content of file f1 to file f2. This is of course if the two files are on the same file system. Now imagine that these two files are on different connected devices by a slow connection link e.g., dial-up IP. If file f1 is large, then copying it to file f2 will be slow. To make the sending and copying process faster, file f1 can be compressed before sending it, but this usually only gains a factor of 2 to 4.

Assuming the two files are the same and probably both derived from the same original file to speed things up. We'll need to take advantage of that similarity. One common way is to send the differences between f1 and f2 and then use this list of differences to rebuild the file. The traditional methods must access both files to create the differences between them. Therefore, these algorithms cannot be used if the files are not on the same machine. The Rsync algorithm calculates the identical parts between files f1 and f2 that should not be sent. The non-identical parts are sent from the source file to the target file so that the receiver then creates a copy of the source file using the references to parts of the existing destination file and the verbatim material.

The algorithm of Rsync:

Suppose we have two computers c1 with file f1 and c2 with file f2. These two files are the same and there is a slow connection between the two computers. The Rsync algorithm works with the following steps:

1. Computer c2 splits the file f2 into a series of non-overlapping fixed-sized blocks of size S bytes (the value of S between 500 and 1000 is quite good for most purposes). The last block may be shorter than S bytes.
2. For each of these blocks c2 calculates two checksums: a weak "rolling" 32-bit checksum and a strong 128-bit MD4 checksum.

   Rolling checksum:

   This checksum is used as a first level to compare the file's blocks. In practice, it is found that the probability of this checksum being equal for two non-identical blocks is very low. Then the strong checksum is calculated. In this case, the rolling checksum must have the property that it is very cheap to calculate, and the successive values can be computed very efficiently using the recurrence relation.

3. Then it sends these checksums to computer c1.
4. Computer c1 then searches through f1 to find all blocks of length S bytes (at any offset, not just multiples of S) that have the same weak and strong checksum as one of the blocks of f2. This can be done in a single pass very quickly using a special property of the rolling checksum.
5. Computer c1 then sends c2 a sequence of instructions for constructing a copy of the file f1. Each instruction is either a reference to a block of f2 or literal data. Literal data is sent only for those sections of f1 which did not match any of the blocks of f2.

At the end, both computers c1 and c2 will have the same copies of the files f1 and f2. In case we have several files to be sent, the pipelining process can be used.

2.2. Google Drive

Google Drive is also a cloud storage service. It grants the users the ability to store their data on the online cloud and to synchronize them between multiple devices or multiple team members. So that they have the access to these files everywhere they have a connection to the internet. According to Zhenhua Li, Zhi-Li Zhang, and Yafei Dai [3], each

user who installed the software, has a so-called sync folder. Where the user can add to or directly modify a file and it will be automatically synchronized with the cloud.

Sync technique:

Timer-triggered Full-file Sync (TFS) is the mechanism used by Google drive. In this technique, once a client modifies the content of a file, a timer is set to watch whether there will be following data updates in the subsequent T seconds (e.g., $T \approx 4.2$ for Google drive). If yes, this data update will be ignored, otherwise, the full updated file is delivered to the cloud.

2.3. Dropbox

Dropbox is a service used as cloud storage for files and allows users to save their files online and synchronize their files with several devices. So that the users can access their files from anywhere, they have a connection to the internet [4]. As Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai [5] state, the main function of all cloud storage services is data synchronization, which synchronizes multiple clients with each other and with the cloud. According to the Dropbox report, 30% of the home user have more than one connected device, and 70% of them share at least one folder.

Sync technique:

Dropbox's synchronization architecture provides two types of network traffic from the client to the server. Metadata transfer and data transfer. The index server is responsible for authenticating the client data and storing the metadata of the files (such as file name, creation date, file size, last modified time, file type, author, index data...etc). There is another data server that provides the file data to the connected client according to the index information stored in the index server. Dropbox uses the Update-triggered Delta Sync (UDS) technique. So that once a file f is updated, the difference (Δf) between the original file and the updated file is calculated and sent to the cloud. This technique significantly reduces the sync traffic and shortens the sync delay. When f is large and Δf is small. Nevertheless, this technique is more complicated than the Google Drive technique (full-file sync), since the delta sync process involves at least three steps:

a) the client retrieves the metadata from the index server.

b) The client computes the difference (Δf or "$binary\ diff$") between the original file and the updated file.
c) The client delivers the Δf to the cloud.

3 Netty framework

In this chapter, a brief overview of the Netty framework will be given. We will learn what the Netty framework is and what it is used for. We will see how to create a Netty application step by step based on a simple example. According to Netty's home page [6], **Netty** is an asynchronous event-driven network application framework for the rapid development of maintainable high-performance protocol servers and clients. Asynchronous and event-driven means:

- **Asynchronous**: operations return immediately and notify the user when they are complete.
- **Event-driven:** every event can be dispatched to a user-implemented method of a handler class.

In this section, it will be explained how to build the first client & server system using the Netty framework. The main idea here is to build a server that binds with a specific port and waits for clients to connect. After establishing the connection, the server task is to welcome the newly connected client. The clients in their turn welcome the server.

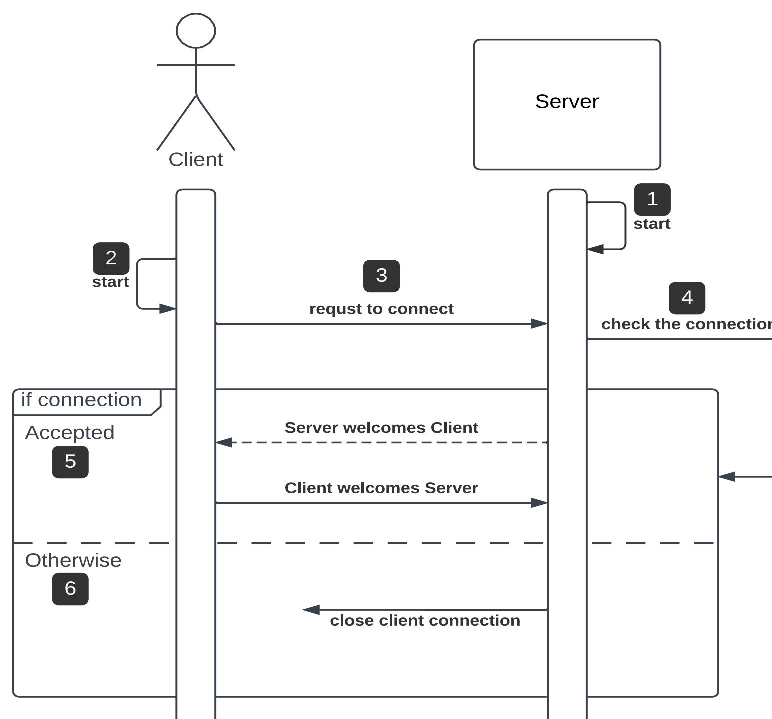


Figure 3.1: Sequence diagram of Netty client-server system

Figure 3.1 illustrates how the client-server system works. In 1 the server application is started. The successful starting means the server is configured at this point and connected

with the specific port, waiting for the client's connection requests. We assume that the client application is started at some time later 2. When starting is successful, it will send a connection request 3 to the server. The server in turn checks the connection 4. If it is accepted 5 and the client is online, the server welcomes the client by sending the message "Hello Client". When the client receives this message from the server, it responds and welcomes the server with the message "Hello Server". Otherwise, if an exception occurs 6, the client connection will be closed, and the server is waiting for other connection requests.

3.1. Netty Server

In this section, we will learn how to create the server program using the Netty framework.

Setting up the server

The first step to create a Netty server is to create `EventLoopGroup` instances as can be seen in Listing 3.1 in lines 2 and 3.

```
1   private void start() throws Exception {
2     EventLoopGroup bossGroup = new NioEventLoopGroup();
3     EventLoopGroup workerGroup = new NioEventLoopGroup();
4     . . .
5   }
```

Listing 3.1: Creating server `EventLoopGroup` instances

We use NIO transport in this example. Thus, an instance of the class `NioEventLoopGroup` must be used to accept and handle new connections. `EventLoopGroup` is a group of one or more `EventLoops`. `EventLoop` defines Netty's core abstraction for handling events that occur during the lifetime of a connection. An `EventLoop` is handled by one Java thread.

Figure 3.2 illustrates the relationships among `Channels`, `EventLoops`, `Threads`, and `EventLoopGroups`:

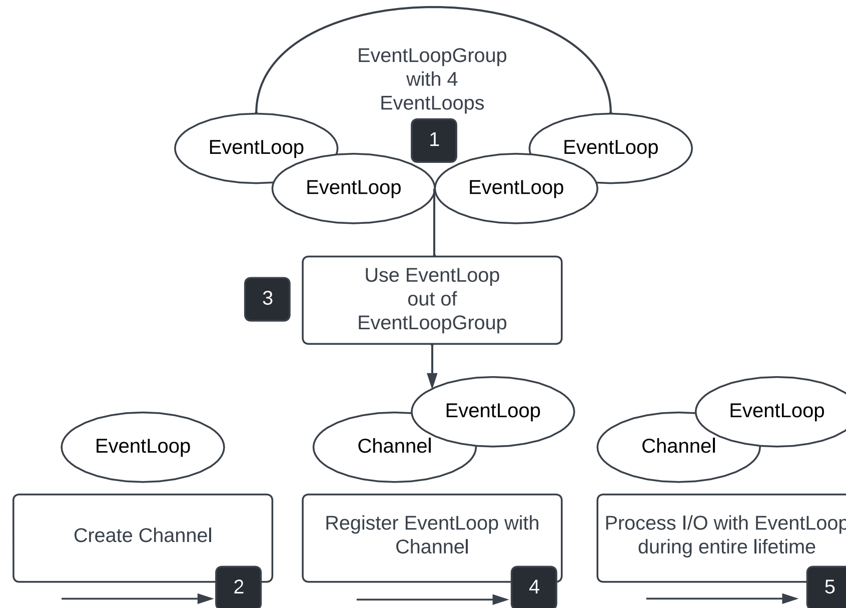


Figure 3.2: The relationships between `Channel`s, `EventLoop`s, and `EventLoopGroup`s [7]

1. An `EventLoopGroup` contains one or more `EventLoop`s.
2. When a connection request from a client is accepted, the client channel is created
3. One of the `EventLoop`s is selected.
4. The created channel is registered for its lifetime with the `EventLoop` assigned for it. Note that, a single `EventLoop` can be assigned to one or more `Channel`s.
5. Since an `EventLoop` is bound to a single thread for its lifetime, all I/O events processed by an `EventLoop` are handled on its dedicated thread.

In this example, two `EventLoopGroup` instances `bossGroup` and `workerGroup` are used. Figure 3.3 illustrates how both `EventLoopGroup`s collaborate with each other.

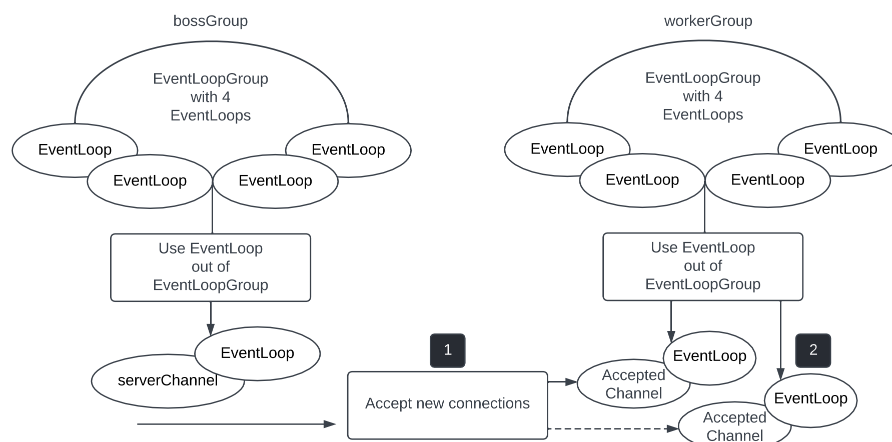


Figure 3.3: Server with two `EventLoopGroup`s [7]

1. The bossGroup on the left side accepts all new incoming connection requests.
2. The workerGroup on the right side assigns an EventLoop for each from bossGroup accepted client channel.

The next step in creating a Netty server is to create and configure the ServerBootStrap. This step is shown in Listing 3.2 in lines 4 to 8. In line 4 the ServerBootStrap instance is created. In line 5 the created EventLoopGroup instances bossGroup and workerGroup are specified. In line 6 the channel type for new expected clients is specified, which is NioServerSocketChannel. In line 7 the socket address is specified using the predefined PORT. The server will bind with this PORT and listen for clients to accept [7]. In line 8 a special handler class called ChannelInitializer is used. This class will be discussed in the following.

```
1   private void start() throws Exception {
2     . . .
3     try {
4         ServerBootstrap serverBootstrap = new ServerBootstrap();
5         serverBootstrap.group(bossGroup, workerGroup);
6         serverBootstrap.channel(NioServerSocketChannel.class);
7         serverBootstrap.localAddress(new InetSocketAddress(PORT));
8         serverBootstrap.childHandler(new ServerChannelInitializer ());
9         . . .
10    }
```

Listing 3.2: Creating and configuring the ServerBootStrap

Creating ChannelInitializer

As Norman Maurer and Marvin Allen Wolfthal [7] state, when a new connection is accepted, a new child Channel will be created. As can be seen in Listing 3.3, the custom ServerChannelInitializer class extends the Netty ChannelInitializer abstract class. The main task of the ChannelInitializer is to add the custom handler instances (in this example HelloClientHandler, see in next section) to the ChannelPipeline. A ChannelPipeline provides a container for a chain of ChannelHandlers and defines an API for propagating the flow of inbound and outbound events along the chain. When a Channel is created, it is automatically assigned its ChannelPipeline.

```
1   public class ServerChannelInitializer extends ChannelInitializer<SocketChannel> {
2       @Override
3       protected void initChannel(SocketChannel channel) throws Exception {
4           ChannelPipeline pipeLine = channel.pipeline();
5           pipeLine.addLast("clientHandler", new HelloClientHandler());
6       }
7   }
```

Listing 3.3: Server ChannelInitializer for initializing the server channel with custom handlers

`ChannelHandler`s are installed in the `ChannelPipeline` as follows:

- A `ChannelInitializer` implementation is registered with a `ServerBootstrap`.
- When method `initChannel` is called, the `ChannelInitializer` installs a custom set of `ChannelHandler`s in the pipeline. The `ChannelInitializer` removes itself from the `ChannelPipeline`.

Creating ChannelHandler

`ChannelHandler` has been designed specifically to support a broad range of use cases. We can think of it as a generic container for any user event handling code. It processes events through the `ChannelPipeline`. These handlers receive events and execute the processing logic for which they have been implemented. When a handler finishes its task, it passes the data to the next handler in the chain. Figure 3.4 illustrates how the server or client `ChannelPipeline` looks like. It has a chain of inbound handlers to address the inbound events step by step. The output of each handler is the input for the next one. In the same way for the outbound events, there is a chain of outbound handlers. This approach increases the code scalability and reusability as can be seen in the next section. The order in which the handlers are executed is determined by the order in which they were added.

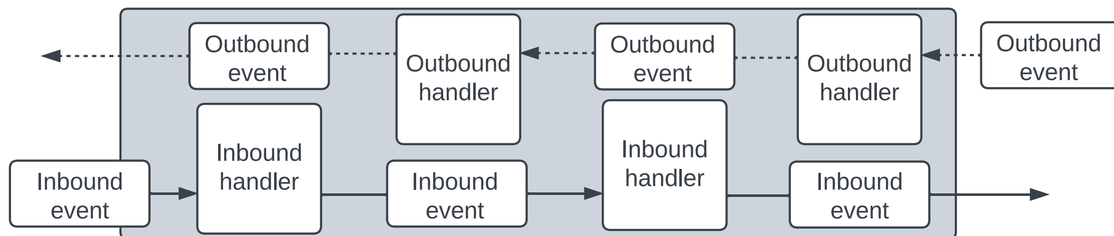


Figure 3.4: Inbound and outbound events flowing through a chain of `ChannelHandlers` [7]

In our example, the class `HelloClientHandler` extends the class `ChannelInboundHandlerAdapter`. It overrides the following three methods as shown in Listing 3.4:

- `channelActive` method is invoked when the client’s request for connection is accepted. In this method, server welcomes the newly connected client with a “Hello Client” message.

- `channelRead` method is called when the server receives any message from the connected clients. In our example, the server writes the received client message to the console.
- `exceptionCaught` method will be invoked when any exception occurs. The stack trace is printed, and the connection will be closed.

```
1   public class HelloClientHandler extends ChannelInboundHandlerAdapter {
2       @Override
3       public void channelActive(ChannelHandlerContext ctx) {
4           Channel clientChannel = ctx.channel();
5           String msgToClnt = "Hello client";
6           System.out.println("--> " + msgToClnt);
7           ByteBuf buf = Unpooled.copiedBuffer(msgToClnt.getBytes(CharsetUtil.UTF_8));
8           clientChannel.writeAndFlush(buf);
9       }
10
11      @Override
12      public void channelRead(ChannelHandlerContext ctx, Object msg) {
13          ByteBuf in = (ByteBuf) msg;
14          System.out.println("<-- " + in.toString(CharsetUtil.UTF_8));
15      }
16
17      @Override
18      public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
19          cause.printStackTrace();
20          ctx.close();
21      }
22  }
```

Listing 3.4: Server handler to handle the incoming messages

The last step in creating the server is to bind it to the predefined `PORT` by calling `bind` method as can be seen in Listing 3.5 in line 5. Method `sync` is called for waiting until the binding process is done. Method `channel` returns the server channel. After this code line, the server is ready to receive client connection requests. Finally, the `EventLoopGroup` instances must be shutdown for stopping all threads assigned to its `EventLoops` when the server is disconnected.

```
1   private void start() throws Exception {
2     . . .
3     try {
4         . . .
5         Channel serverChannel = serverBootStrap.bind().sync().channel();
6         . . .
7     } catch (Exception e) {
8         e.printStackTrace();
9     } finally {
10        bossGroup.shutdownGracefully().sync();
11        workerGroup.shutdownGracefully().sync();
12    }
13  }
```

Figure 3.5: Binding the server to the specified `PORT`

3.2. Netty Client

In this section, it is shown how the Netty client program is created and configured.

Setting up the client

In the same way, as for the server, the process for creating a Netty client starts with creating an `EventLoopGroup` instance as can be seen in Listing 3.6 in line 2. The next step is creating a `Bootstrap` instance. It is used for configuring the Netty client (4-8).

Note that the client-side uses class `Bootstrap`, while the server-side uses class `ServerBootstrap`. In line 4 the client `Bootstrap` instance is created. In line 5 the `EventLoopGroup` is specified doe handle all events for the `Channel`. In line 6 the `Channel` implementation class is specified. In line 7 the `InetSocketAddress` is specified. In line 8 `ClientChannelInitializer` is registered with the `BootStrap`. We see how this is done in the next section.

```
1   private void start() throws Exception {
2     EventLoopGroup group = new NioEventLoopGroup();
3     try {
4         Bootstrap clientBootstrap = new Bootstrap();
5         clientBootstrap.group(group);
6         clientBootstrap.channel(NioSocketChannel.class);
7         clientBootstrap.remoteAddress(new InetSocketAddress(HOST, PORT));
8         clientBootstrap.handler(new ClientChannelInitializer());
9         Channel clientChannel = clientBootstrap.connect().sync().channel();
10        . . .
11    } finally {
12        group.shutdownGracefully().sync();
13    }
```

Figure 3.6: Creating and configuring the Netty client

Creating ChannelInitializer

Listing 3.7 illustrates how to create the `ChannelInitializer` for the client The `ChannelInitializer` is mainly used to initialize the client `ChannelPipeline` with specific custom handlers. These handlers are added to the `ChannelPipeline` when method `initChannel` is invoked.

```
1   public class ClientChannelInitializer extends ChannelInitializer<SocketChannel> {
2       @Override
3       protected void initChannel(SocketChannel channel) throws Exception {
4           ChannelPipeline pipeLine = channel.pipeline();
5           pipeLine.addLast("serverHandler", new HelloServerHandler());
6       }
7   }
```

Listing 3.7: Client ChannelInitializer for initializing the client channel

Creating ChannelHandler

Listing 3.8 illustrates how to create the `ChannelHandler` on the client-side. This handler is responsible for handling all incoming data from the server. This is done mainly in the overridden method `channelRead0`. This method is invoked when a message is received from the server. The parameter `msg` contains the server message as a stream of bytes. In this example, the client writes the received message to the console and welcomes the server. This is done in lines (6-9). The client message of type String is converted to a `ByteBuf` object and is sent to the server by invoking method `writeAndFlush`.

```
1   public class HelloServerHandler extends SimpleChannelInboundHandler<ByteBuf>{
2       @Override
3       public void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) {
4           Channel serverChannel = ctx.channel();
5           System.out.println("<-- " + msg.toString(CharsetUtil.UTF_8));
6           String msgToSrvr = "Hello Server";
7           System.out.println("--> " + msgToSrvr);
8           ByteBuf buf = Unpooled.copiedBuffer(msgToSrvr.getBytes(CharsetUtil.UTF_8));
9           serverChannel.writeAndFlush(buf);
10      }
11
12      @Override
13      public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
14          cause.printStackTrace();
15          ctx.close();
16      }
17  }
```

Listing 3.8: Netty Client handler to address incoming data from the remote peer

3.3. Result

In the section, the results of executing the server and client applications are represented as they appear in the Editor console.

Server-side console

The lines 1 and 2 appear on the console window, when the server application is successfully started. The server can be stopped by pressing any key. After starting, the server is in a waiting state, i.e., it is waiting for connection requests from clients. When a client application is started successfully, the connection between server and client is established. The server welcomes the client with a "Hello Client" message as can be seen in line 3. The client responds with a "Hello Server" message as can be seen in line 4. In line 5 the enter key is pressed. The line 6 is the success information for closing the server application.

```
1   start the server ...
2   press any key to exit
3   --> Hello client
4   <-- Hello Server
5
6   the server closed
```

Client-side console

On the client-side, lines 1 and 2 appear on the console window when the client application is successfully started. That means the connection is established. To stop the client one can press any key. Now the client receives a welcome message from the server in line 3. After that, it welcomes the server in line 4. In line 5 the enter key is pressed. The line 6 is the success message for closing the client application.

```
1   start the client ...
2   press any key to exit
3   <-- Hello client
4   --> Hello Server
5
6   the client closed
```

4 Netty-based file synchronization systems

This chapter answers the question why Netty is preferred to create a server-based file sync system. As we have seen in the second chapter, all cloud storage services are based on the client-server architecture. The client represents the user and his devices. The server represents the cloud itself, where the client files are stored. This server synchronizes all clients with each other to make the state of all connected clients identical. We have seen in the third chapter an overview of the Netty framework and how it can be used to create a high-performance client-server system. Netty provides a lot of advantages, which are discussed in the following.

4.1. Separation of concerns

During the development of network applications, a lot of time is spent in preparing the network settings. This task can be very complicated for developers. The Netty framework provides an architecture, which separates the business logic from the network logic of our applications. Thus, making the developers' life easier and letting them concentrate on their applications. This benefit is provided in the Netty framework by using the `ChannelHandler`s. These handlers are added to the `ChannelPipeline` of the server channel and client channel. They can also be used to think separately about the processing of incoming and outgoing data in several stages. This is done by adding several handlers to the `ChannelPipeline` of server and client, one handler for each task will be applied on the data.

4.2. Reusability

Netty provides a framework that allows developers to integrate incoming and outgoing data processing tasks. This is done by implementing both `ChannelInboundHandler` and `ChannelOutboundHandler` interfaces at the same time. Netty also provides the possibility to separate the tasks of processing incoming and outgoing data. Also, to segment the processing of incoming data into several stages. As well as for the outgoing data, which contributes significantly and maximizes code reusability. In [7] the question "Why would we not use these composite classes all the time in preference to separate decoders (handlers for addressing the incoming data) and encoders (handlers for addressing the outgoing data)?" is answered as follows, Because keeping the two functions separate

wherever possible maximizes code reusability and extensibility, which is the basic principle of Netty's design.

4.3. ByteBuf class

Netty provides the `ByteBuf` API as an alternative to `ByteBuffer` API in the JDK. Using class `ByteBuf` brings a lot of benefits that makes Netty a preferred framework in the developing network applications field. According to [7] `ByteBuf` is a powerful implementation that addresses the limitation of the `ByteBuffer` of JDK and has these advantages:

- It's extensible to user-defined buffer types.
- Transparent zero-copy is achieved by a built-in composite buffer type (It allows to move data quickly and efficiently from a file system to the network without copying from kernel space to user space, which can significantly improve performance in protocols such as FTP or HTTP).
- Switching between reader and writer modes doesn't require calling `ByteBuffer's flip` method (because `ByteBuf` has two distinct pointers for reading and writing).
- Method chaining is supported (using the `ByteBuf` classes as in-output between `ChannelHandlers` chain in the `ChannelPipeline`).
- Reference counting is supported (the ability to know the number of references of resources and release them).
- Pooling is supported (to create Unpooled `ByteBuf` instances).

4.4. Memory management

According to [7], Netty provides the reference counting approach. It is a technique for optimizing memory use and performance. This is done by counting the references (objects) holding a resource and releasing these resources when the number of its references is decreased to 0. Netty supports this technique in version 4 for `ByteBuf` and `ByteBufHolder` by implementing the interface `ReferenceCounted`. Listing 4.1 and Listing 4.2 illustrate this approach.

```
1   Channel channel = ...;
2   ByteBufAllocator allocator = channel.alloc();
3   ....
4   ByteBuf buffer = allocator.directBuffer();
5   assert buffer.refCnt() == 1;
```

Listing 4.1: Reference counting [7]

In line 1 a `Channel` instance is created. In line 2 we get the `ByteBufAllocator` from the channel. Line 3 represents additional pieces of code. In line 4 `ByteBuf` is allocated from `ByteBufAllocator`. In line 5 the expected reference count of 1 is checked.

```
1   ByteBuf buffer = ...;
2   boolean released = buffer.release();
```

Listing 4.2: Release reference-counted object [7]

In line 1 `ByteBuf` instance is created. In line 2 the active references are decremented by calling method `release` of `ByteBuf`. At reference count 0 the object is released, and the method returns true.

Netty also provides the automatic release of `ByteBuf` by extending class `SimpleChannelInboundHandler` on the client-side. This class takes care of releasing the memory references to the `ByteBuf` that holds the message, by passing them to `ReferenceCountUtil.release(Object)`. If it is needed to pass the object to the next handler in the `ChannelPipeline`, the `ReferenceCountUtil.retain(Object)` must be used.

5 Use Cases and Requirements

In this chapter, it will be explained, how the implemented file synchronization system works. This is performed in different cases as. Finally, a list of requirements is derived.

5.1. Use cases

5.1.1. Single client-server case

In this case just one client is connected with the server.

The client will perform some changes in his file system registered to be synchronized with the server. Figure 5.1 illustrates how the synchronization process is performed step by step.

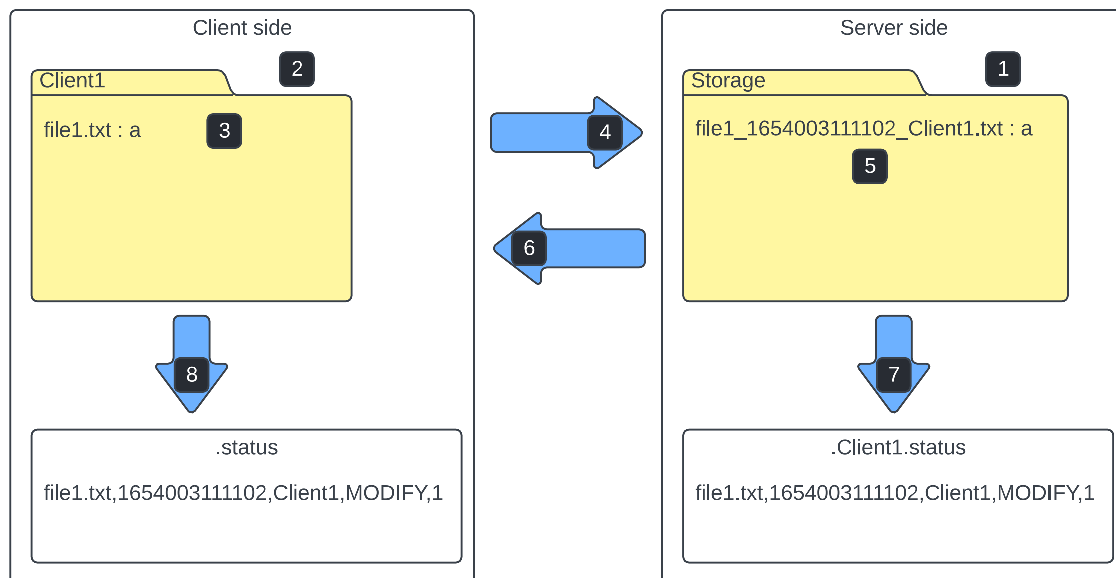


Figure 5.1: Single client connected with the server

1. The server program is started. The server is now waiting for client connection requests.
2. Client1 program is started. It sends a connection request to the server. If the request is accepted, the server creates a status file `".Client1.status"` for this client for the first time as can be seen in Figure 5.1.
3. Client1 pastes the file `file1.txt` with the content `"a"` in his monitored directory. The `FileWatcher` marks this change and creates event `"file1.txt,1654003111102,Client1,Modify,1"`. This event is pushed into the event queue on client-side.

4. `CommandSender` of the Client1 takes this event from the event queue in the next loop. It adds the keyword `COMMAND` in the front of the event. The event becomes: `"COMMAND,file1.txt,1654003111102,Client1,Modify,1"`
5. It receives the event and creates a copy of the file `file1.txt` called `"file1_1654003111102_Client1.txt"` in its storage folder. This name contains the original file name, the last modification time, and the name of the client, who modified the file. Then it tells Client1 that it is ready to receive file data. Client1 starts sending data packets. The server writes them into the newly created file.
6. When the sending process is completed, the server tells Client1 that the file is stored.
7. The server writes the processed event in `".Client1.status"`.
8. At the end Client1 also writes this processed event in his status file `".status"`.

Now we assume that Client1 modifies the content of `file1.txt` from "a" to "abc" on the date (1657481584506 = Sun Jul 10 21:33:04 CEST 2022). Figure 5.2 illustrates how this event is processed and shows the result on both sides client and server.

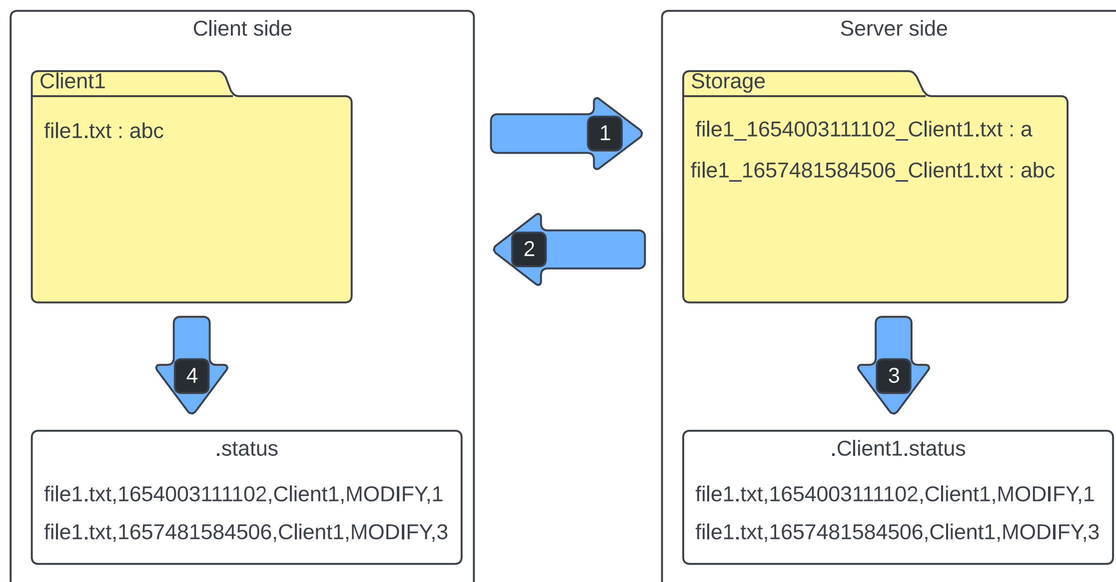


Figure 5.2: Syncing file1.txt after modifying it from "a" to "abc"

The file version `"file1_1654003111102_Client1.txt"` on the server storage will be deleted after a specific time (e.g., 7 days).

Note, that in the case of a single client connected, the conflict case cannot occur.

5.1.2. Multiclient-server case

In this case, there is more than one client connected to the server.

The server will store files and synchronize them with connected clients. Figure 5.3 illustrates the syncing process step by step.

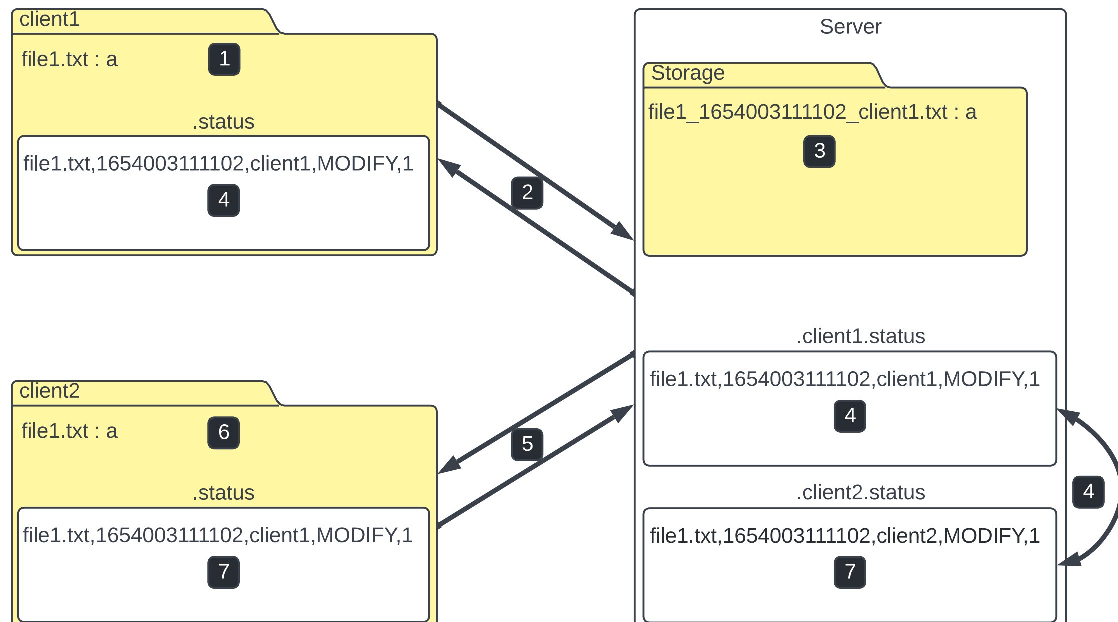


Figure 5.3: Multiclient-server syncing test

At the beginning we assume that the server is online and both clients client1 and client2 are connected.

1. Client1 pastes the file `file1.txt` with the content "a" on the date (1654003111102 = Tue May 31 15:18:31 CEST 2022) into his monitored directory. This change (event) is noticed by the client1 `FileWatcher` that pushes it into the event queue.
2. The client1 `CommandSender` takes this event from the queue and sends it to the server.
3. The server creates the empty copy file "`file1_1654003111102_client1.txt`" and informs the client1, that it is ready to receive file data. Client1 starts sending file data as packets. The server writes received data packets into the newly created file. When the server has received the number of bytes equals to the original file size, it informs client1 that the file is completely received and stored.
4. The server writes the processed event into the "`.client1.status`" file. Client1 writes this event to his "`.status`" file. Now the server compares its status files

`".client1.status"` and `".client2.status"`. As a result of this comparison, it becomes clear to the server that the file `file1.txt` modified by client1 on date 1654003111102 is not synchronized with client2 yet.

5. The server sends an event to client2 for syncing the newly stored file.
6. Client2 creates an empty file copy and informs the server that he is ready to receive file data packets. The server starts sending data packets. When the correct number of bytes has been received, client2 notifies the server that the file is correctly synchronized.
7. Client2 writes the processed event into his status file `".status"`. The server also writes this event into `".client2.status"` file so that the server status files become identical.

Note: in the next comparison of status files, the server discovers no differences between them. This means that no changes are made by any client and all files are synchronized up to the present time.

5.1.3. Case with conflict

The conflict case occurs when the server discovers two unsynchronized versions of the same file at the same time. Figure 5.4 illustrates how this case occurs and which result has to be expected.

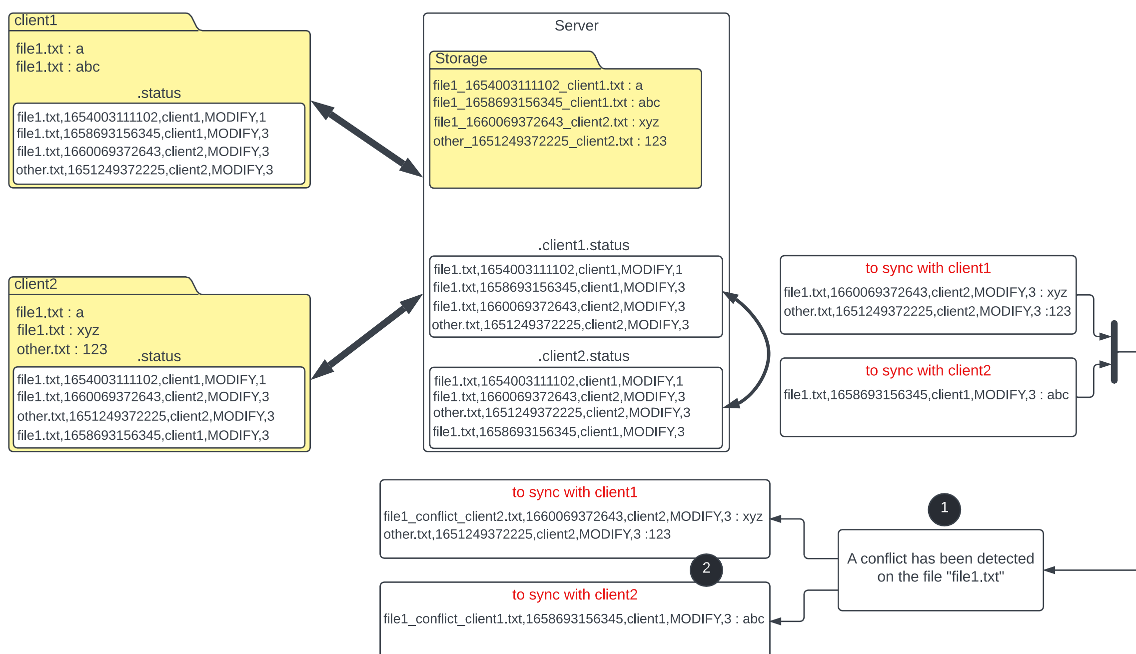


Figure 5.4: Conflict case between 2 clients

We assume that the server and both clients are online, and their monitored directories are empty at the beginning. Client1 pastes `file1.txt` with content "a" on date `1654003111102` in his directory. This file is stored on the server storage and synchronized with client2 in the way mentioned previously. Now both clients' directories are identical and both status files in the server are identical with this content:

`"file1.txt,1654003111102,client1,MODIFY,1"`.

At this point, we assume that the client2 is disconnected. That means any change done by client1 will be stored on the server storage but not synchronized with client2.

Client1 modifies `file1.txt"` content from "a" to "abc" on date `1658693156345`. The modified file is stored on the server. The disconnected client2 modifies the same file from "a" to "xyz" on another date `1660069372643` and pastes the file `other.txt` with content "123" on date `1651249372225` in his monitored directory. Now the conflict case occurs, two versions of the file `file1.txt` are changed but not synchronized. At some time in the future, client2 connects to the server. All changes performed while client2 was offline, are sent and stored on the server first. There is now more than one connected client. That means the comparison between its status files will be performed, to specify which events (file versions) must be synchronized with which client. After that, the server in this system performs the second level of comparison to detect if there is a conflict case or not. This includes the following steps, as can be seen in Figure 5.4:

1. In this scenario, the server detects that there are two unsynchronized versions of `file1.txt` and reports a conflict case.
2. The server creates the file `"file1_conflict_client2.txt"` with content "xyz" to be sent to client1. It also creates file `"file1_conflict_client1.txt"` with content "abc" to be sent to client2. All other files are synchronized as usual. At the end all status files on the server-side are identical.

5.2. Requirements

From the use cases the requirements can be derived. Table 6.1 list the requirements of the server-based file synchronization system. The clients can go online by sending a connection request to the server. A specific client directory is registered to be observed. The modified files in the observed directory are sent to the server to be stored. The server

sends these files to all clients, who don't have them or have an old version of them. The conflicts between clients are recognized (e.g., when two clients change a file at the same time) and solved. The old versions of the stored files on the server are deleted after a specific time (except the last version). The modified files by offline clients are sent to the server once these clients go online (more details in the following sections).

Nr	Requirement
1	Clients can register and log in
2	Changes in client directories are observed
3	Changed files are sent to server and stored
4	Changed files are sent by server to other clients
5	Clients store the changed files from other clients
6	Conflicts are recognized
7	The server keeps all changed versions of files for a certain period, after that only the latest version is kept, all others are deleted
8	Offline changes are considered as soon as the client goes online

Table 5.1: User requirements of the sync system

6 Implementation Description

In this chapter, the implementation of the server-based synchronization system using the Netty framework is described. In Section 6.1 the system design is presented and in Section 6.2 the implementation is described.

6.1. System Design

In this section, the system design will be described using UML diagrams (system architecture, state diagrams).

6.1.1. System architecture

Figure 6.1 illustrates the system architecture of the server-based file sync system and its main components. The observed directory on the client side is registered with the `FileWatcher`. All changes in this directory are added to the Events Queue by the `FileWatcher` as commands for the server. The sending service is responsible for sending these commands and the corresponding modified files to the server. The receiving service on the server side processes the commands sent by the client and stores the modified files in the server Storage. When the receiving process is done, the receiving service writes the processed command in the corresponding status file and notifies the client to write this command as processed in its own status file. The sending service on the server side compares the status files. If these files are not equal, the sending service sends the files to the clients, so that the status files at the end becomes equal. On the other side the client receiving service receives what the server sends. Following the saving process, the receiving service writes the processed command in the client status file and informs the server to write this command as a processed command. Eventually, all clients will be identical.

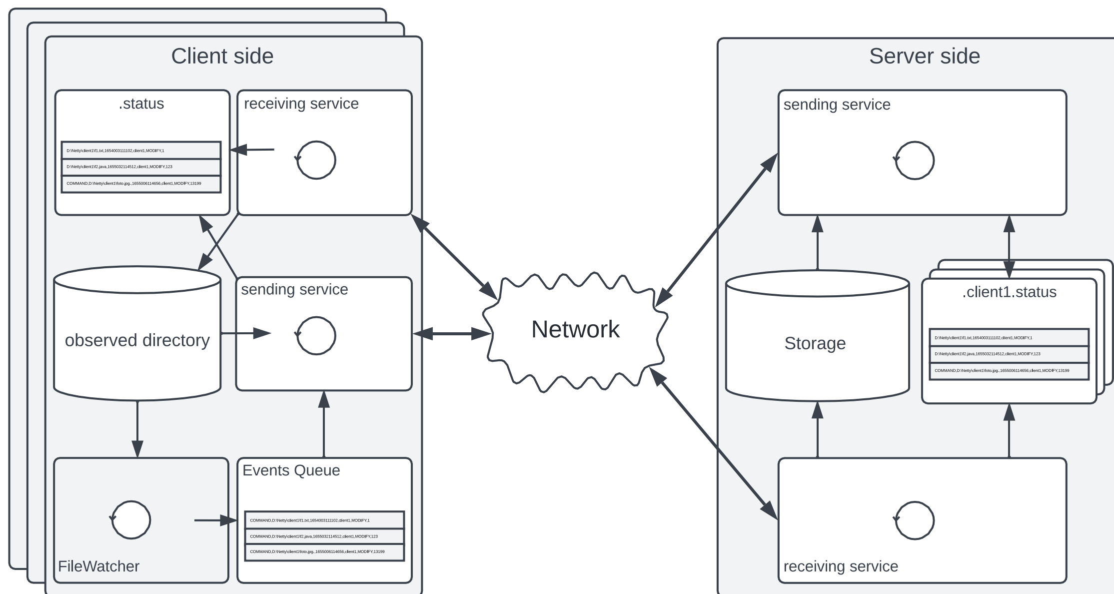


Figure 6.1: System architecture of the server-based file sync system

6.1.2. State diagrams

The state diagrams for server-side and client-side of the server-based file sync system are presented in this section.

State diagram of the server:

Figure 6.2 illustrates the states, in which the server can be found. When the server goes online, it waits for client connection requests and remains in this state as long as it is online. When a connection request is accepted, the server goes at the same time to the receiving and sending phases. In the receiving phase, the server can be found in one of the following states:

WELCOME state: In this state, the server welcomes the receiving channel of the newly connected client. The client in turn welcomes the server. If the welcome messages are correctly exchanged, the server switches with this client to the `COMMANDS` state.

COMMANDS state: In this state, the server is ready to receive commands from the client. These commands contain all required information, that the server needs to process them. We distinguish between three types of commands:

1. *Create file*: This command is sent by the client when he creates a file in his observed directory. The server creates an empty version of this file in its storage.

2. *Delete file*: This command is sent by the client when he deletes a file from his observed directory. The server redirects the command to all connected clients to delete this file from their observed directories. However, the server keeps at least one copy (the last one) of the file.
3. *Modify file*: This command is sent by the client when he modifies the content of a file in his observed directory. If the file does not exist in the server storage, the server creates an empty version of it and notifies the client that it is ready to receive the file data. At this point, the server switches with this client to the `DATA` state.

DATA state: This state allows the server to receive file data over the network from the client. During receiving data chunks, the server must be able to distinguish between four scenarios:

1. *First scenario:* receiving a number of bytes less than the actual file size. In this case, the client must continue sending and the server must continue receiving. If the client stops sending for a certain period (e.g., 10s) although he has not completed sending file data, the server reports a timeout error and switches back to the `COMMANDS` state.
2. *Second scenario:* receiving a number of bytes more than the actual file size. In this case, the server reports an error and switches back to the `COMMADS` state.
3. *Third scenario:* receiving a number of bytes equals to the actual file size. In this case, the server notifies the client, that the desired file is stored. Now the server switches back with this client to the `COMMANDS` state.
4. *Fourth scenario:* receiving no data for a specific time. In this case a timeout error occurs. The server switches back to the `COMMANDS` state.

In the sending phase, the server can be found in one of the following states:

WELCOME state: In this state, the server welcomes the sending channel of the newly connected client. The client in turn welcomes the server. If the welcome messages are correctly exchanged, the server switches to the `SENDING_FILES` state.

SENDING_FILES state: in this state, the server periodically compares the status files of all clients. In the case that they are not identical, the server sends the corresponding

commands and unsynchronized files (data) to the clients. The final result of this state is keeping all clients identical.

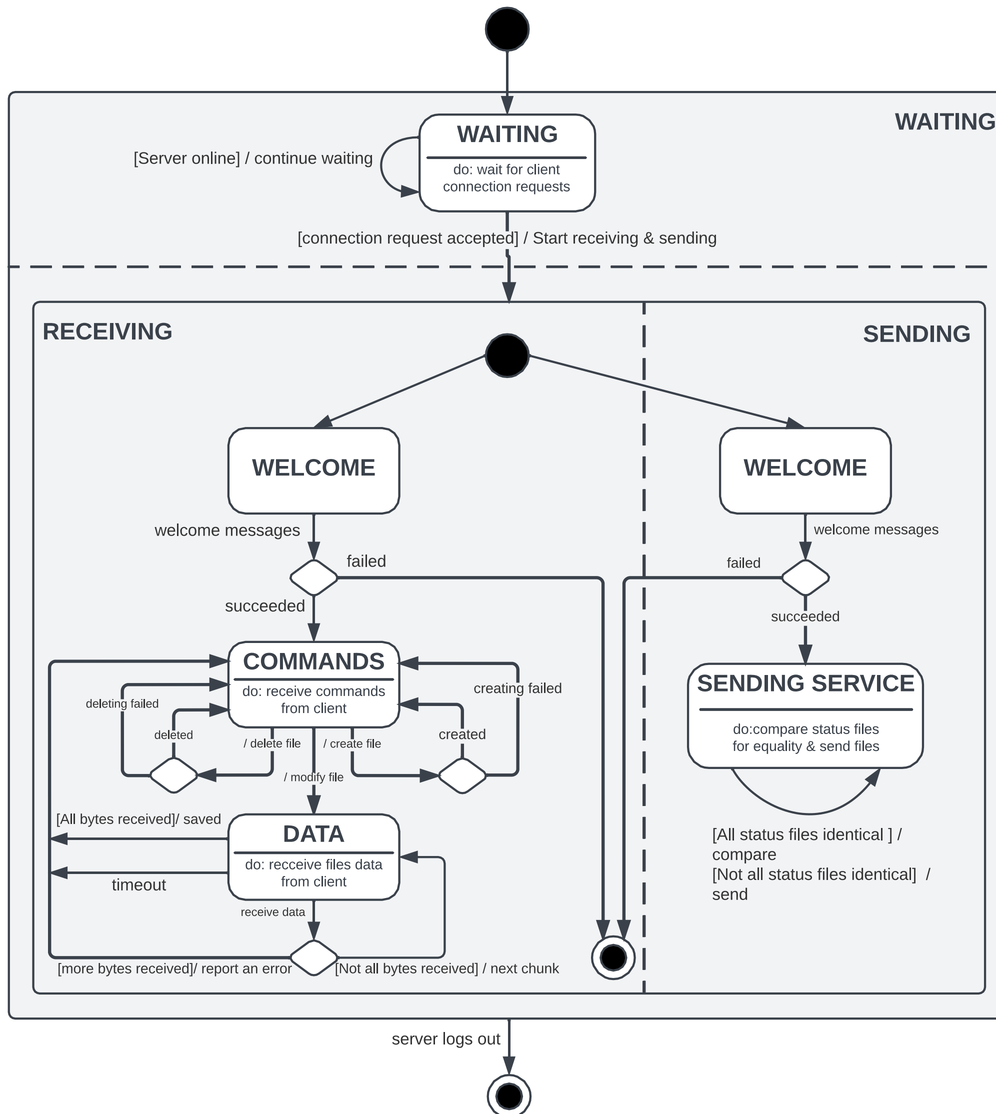


Figure 6.2: State diagram of the server

State diagram of client:

Figure 6.3 illustrates the states, in which the client can be found. When the client starts, it sends a connection request to the server. If the connection fails, the client goes to the final state, otherwise the client switches at the same time to the receiving and sending phases. In the receiving phase, the client can be found in one of the following states:

WELCOME state: In this state, the client receives a welcome message from the receiving channel of the server and responds with a welcome message. If the process is succeeded, the client switches to the `COMMANDS` state, otherwise, the client goes to the final state.

COMMANDS state: In this state, the client receives commands (create, modify, delete) from the server. In case the commands (create file, delete file) the client goes back to the `COMMANDS` state if the command is successfully handled or if it failed. In case the command (modify file), the client switches to the `DATA` state.

DATA state: In this state the client receives the modified files data from the server, analogous to the server side. After processing all command types, the client switches back to the `COMMANDS` state.

In the sending phase the client can be found in one of the following states:

WELCOME state: In this state, the client receives a welcome message from the sending channel of the server and responds with a welcome message. If the process is succeeded, the client switches at the same time to the `WATCHING` and `SENDING_FILES` states.

WATCHING state: In this state the client `FileWatcher` is started to monitor the registered client directory for changes and events. When changes (events) occur in this observed directory, the `FileWatcher` adds these events to the events queue.

SENDING_FILES state: In this state, the periodically running `CommandSender` sends the commands from the events queue to the server. When the server responds that it is ready to receive file data, the `SendingHandler` of the client sends the data to the server.

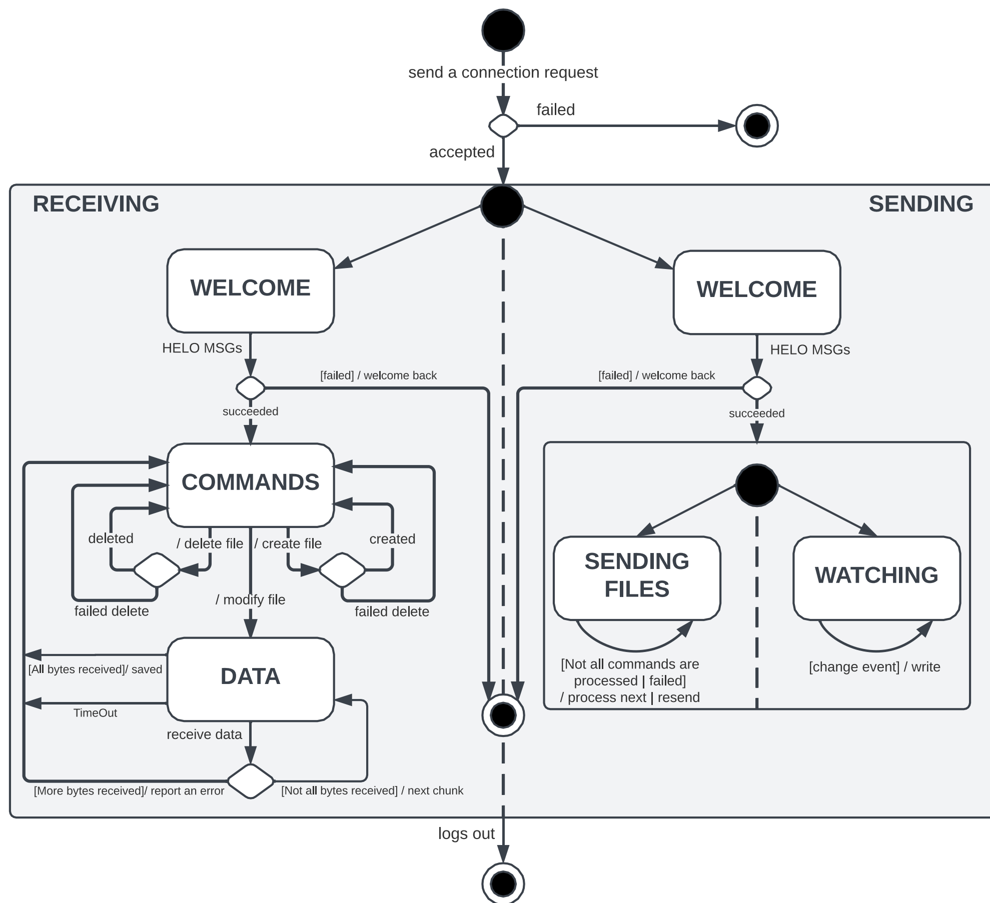


Figure 6.3: State diagram of the client

6.2. Implementation

In this section, the implementation of the server-based file sync system using the Netty framework will be explained. The implementation meets the requirements and the design presented in the previous sections. The system consists of two main programs, the server program, and the client program. We will start with the main system components and their roles in the system.

6.2.1. Main system components

In this section, the main components (classes, enumerations, methods, files) of the whole system are listed and illustrated.

Server components

Enumeration ProcessingState

In Listing 6.1 the states are defined, in which the server can be found. `WELCOME_RECEIVER` state is to welcome the receiving channel of the newly connected client. `WELCOME_SENDER`

state is to welcome the sending channel of the newly connected client. COMMANDS state is to receive and handle the client commands. DATA state, in which the server receives data (files content) from the client. SENDING_FILES state, in which the server sends commands and files data to the connected clients.

```
1   public enum ProcessingState {
2     WELCOME_RECEIVER, WELCOME_SENDER, COMMANDS, DATA, SENDING_FILES;
3   }
```

Listing 6.1: Processing modes in which the server can be found

Class ReceivingTimeoutHandler & class SendingTimeoutHandler

These classes extend class ReadTimeoutHandler. The overridden method readTimedOut is invoked, when there is no data to read from the client for a specific time (passed to the constructor of the class). Listing 6.2 and Listing 6.3 illustrate the implementation of these classes. We check first if the client channels are still active (9-12). After that we extract the ReceivingInfo ri (14) and the SendingInfo si (14) objects from the corresponding maps. Now for the receiving if the server is in DATA state, it sends to the client a TIMEOUT_SAVE error, and switches to COMMANDS state. But if the server is in WELCOME_RECIVER state, that means the welcome messages are incorrectly exchanged and, in this case both client channels are closed (the same logic for WELCOME_SENDER state).

```
1   private class ReceivingTimeoutHandler extends ReadTimeoutHandler {
2
3     public ReceivingTimeoutHandler(int timeoutSeconds) {
4       super(timeoutSeconds);
5     }
6
7     @Override
8     protected void readTimedOut(ChannelHandlerContext ctx) throws Exception {
9       if (!ctx.channel().isActive()) {
10        Out.println("SERVER ERROR: the client is no longer online");
11        return;
12      }
13
14      ReceivingInfo ri = receivingInfoMap.getReceivingInfo(ctx.channel());
15
16      if (rci.getReceivingState().equals(ProcessingState.DATA)) {
17        Out.println("SERVER ERROR: Timeout while receiving");
18        sendMsg(ctx.channel(), TIMEOUT_SAVE + " " + rci.getFilePathInServer());
19        rci.setReceivingState(ProcessingState.COMMANDS);
20      }
21      if (rci.getReceivingState().equals(ProcessingState.WELCOME_RECEIVER)) {
22        Out.println("SERVER ERROR: Welcome messages are not correctly exchanged");
23        ReceivingHandler.terminate();
24        SendingHandler.terminate();
25      }
26    }
27  }
```

Listing 6.2: Class ReceivingTimeoutHandler

```
1   private class SendingTimeoutHandler extends ReadTimeoutHandler {
2
3     public SendingTimeoutHandler(int timeoutSeconds) {
4       super(timeoutSeconds);
5     }
6
7     @Override
8     protected void readTimedOut(ChannelHandlerContext ctx) throws Exception {
9       if (!ctx.channel().isActive()) {
10        Out.println("SERVER ERROR: the client is no longer online");
11        return;
12      }
13
14      SendingInfo si = sendingInfoMap.getSendingInfo(ctx.channel());
15
16      if (sci.getSendingState().equals(ProcessingState.WELCOME_SENDER)) {
17        Out.println("SERVER ERROR: Welcome messages are not correctly exchanged");
18        ReceivingHandler.terminate();
19        SendingHandler.terminate();
20      }
21    }
22  }
```

Listing 6.3: Class SendingTimeoutHandler

Class Event

This class is used as a command descriptor which has the commands information sent between the server and the clients. Listing 6.4 illustrates the class fields and Table 6.2 gives a short description of them.

```
1   public class Event {
2
3     private Path filePath;
4     private Long lastModify;
5     private String client;
6     private EventType eventType;
7     private Long fileSize;
8     . . .
9   }
```

Listing 6.4: Class Event for events sent between server and client

Event content	Description
COMMAND	The keyword to know that the message is a command.
filePath	The path of the modified file.
lastModify	The last modification date in milliseconds.
client	The name of the client, that has modified the file.
eventType	The type of the change which is done on this file (CREATE, DELETE, MODIFY)
fileSize	The size of the file changed(in bytes)

Table 6.1: Event information (commands sent between server and clients)

Example of an event (command):

"COMMAND,D:\Netty\client1\f1.txt,1654003111102,client1,MODIFY,1"

Status files

For each client there is a file on the server. In these files, the events processed by the server are stored, as can be seen in Figure 6.4. These files are automatically created using the client names, which are sent with the welcome messages in the welcome states when the clients connect for the first time. The name of these files has the signature (.client name.status).

```
.client1.status
1  D:\Netty\client1\New Text Document.txt,1663230535065,client1,MODIFY,1
2  D:\Netty\client1\New Text Document.txt,1663230548890,client1,MODIFY,2
3  D:\Netty\client1\New Text Document.txt,1663230557762,client1,MODIFY,3
4  D:\Netty\client1\file1.txt,1654003111102,client1,MODIFY,1
5  D:\Netty\client1\New Text Document.txt,1663230589509,client1,DELETE,0
6  D:\Netty\client1\renamed.txt,1663230557762,client1,MODIFY,3
7
```

Figure 6.4: Status file of client1 on the server-side

Class ReceivingInfo

This class contains the most useful receiving information, that the server needs to know at any time, what is received form the client and the server makes the right decision with this client accordingly. Listing 6.5 illustrate the class fields and Table 6.3 contains a short description of these fields respectively.

```
1   public class ReceivingInfo {
2
3     private final Channel channel;
4     private ProcessingState processingState;
5     private long numReceivedBytes;
6     private Path historyPath;
7     private Path filePathInServer;
8     private long fileSize;
9     private ByteBuf data;
10    private String name;
11    private Event event;
12    . . .
13  }
```

Listing 6.5: Fields of the class `ReceivingInfo`

Field	Description
channel	Client sending channel, through which the server receives data from the client.
processingState	The state in which the server found with this client.
numReceivedBytes	The number of bytes received from this client so far.
historyPath	The path of the status file of this client.
filePathInServer	The processed file path in the server.
fileSize	The size of the processed file (Bytes must be received from the client)

data	The data packet received from the client (Netty ByteBuf)
name	The name of the client that the server deals with.
event	Client command information explained in the previous section.

Table 6.2: Receiving information needed by the server

Class SendingInfo

This class contains the most useful sending information, that the server needs to know at any time, what is sent to the client and the server makes the right decision with this client accordingly. Listing 6.6 illustrate the class fields and Table 6.4 contains a short description of these fields respectively.

```
1   public class SendingInfo {
2     private final Channel channel;
3     private Path historyPath;
4     private Path filePath;
5     private ByteBuf data;
6     private String name;
7     . . .
8   }
```

Listing 6.6: Fields of the class `SendingInfo`

Field	Description
channel	Client receiving channel, through which the server sends data to the client
historyPath	The path of the status file of this client
filePath	The processed file path in the server
data	Data packet sent to the client
name	The name of the client

Table 6.3: Sending information needed by the server

ReceivingHandler class

This Class is one of the handlers added to the `ChannelPipeline` of the first server channel (receiving data channel). As its name says, the main task of this handler class is to process the received data from clients and to forward it to the next handler in the `ChannelPipeline`. It extends the `ChannelInboundHandlerAdapter` abstract class in Netty API to implement the most important methods as can be seen in the following listings. In Listing 6.7 method `channelActive` is overridden. This method is invoked once the client connection request is accepted, and his sending channel gets active. This channel is stored in the `ReceivingInfo` class. Each client has an entry in the `ReceivingInfoMap`. The key to this map is the sending client channel, and the value is the `ReceivingInfo`

instance. The server sends a welcome message in this method over the receiving client channel.

```
1   private class ReceivingHandler extends ChannelInboundHandlerAdapter {
2     @Override
3     public void channelActive(ChannelHandlerContext ctx) throws Exception {
4       ReceivingInfo ri = new ReceivingInfo(ctx.channel());
5       receivingInfoMap.addReceivingInfo(ctx.channel(), rci);
6       sendMsg(ctx.channel(), HELO_MSG + " " + SERVER);
7     }
8     . . .
9   }
```

Listing 6.7: Method `channelActive` in Server class `ReceivingHandler`

The second important method is `channelRead`. This method is invoked once a message is received from the client. Listing 6.8 illustrates the implementation of this method. First, the received message object is tested, whether it is an instance of Netty `ByteBuf`. If it is not the case, the client channel is closed. After that, the current receiving information object is extracted from the `ReceivingInfoMap`. Then, the current message is stored. At the end, the handling method `handleClientMessage` is called, where all client messages are handled.

```
1   private class ReceivingHandler extends ChannelInboundHandlerAdapter {
2     . . .
3     @Override
4     public void channelRead(ChannelHandlerContext ctx, Object msg) throws
5     IOException, InterruptedException {
6       if (!(msg instanceof ByteBuf)) {
7         Out.println("FATAL ERROR: invalid client message format");
8         ctx.channel().close();
9         return;
10      }
11      ReceivingInfo ri = receivingInfoMap.getReceivingInfo(ctx.channel());
12      ri.setData((ByteBuf) msg);
13      handleClientMessage(ri);
14    }
15    . . .
16  }
```

Listing 6.8: Method `channelRead` in Server class `ReceivingHandler`

Method `handleClientMessage` illustrated in Listing 6.9 is used to handle all messages during receiving process depending on the server state. As a parameter `ReceivingInfo` instance is passed to this method. Which allows the server to know the receiving information with the sender client.

If the server is in `DATA` state, it receives data chunks from client. These chunks are written into the corresponding file. if the number of received bytes are equal to the size of the file has to be received, the server sends `MODIFIED` message to the client to inform him that the

sent file is stored. Then, the server switches to the COMMANDS state to receive new commands from this client. If the number of received bytes greater than the file size, the server sends FAILED_MODIFY message to the client and switches to the COMMADNS state to receive new commands from the client.

If the server is in COMMANDS state, it receives commands from the client to process them. These commands are (CREATE, DELETE, MODIFY) file and they have the form we saw in class EVENT. They have all important information the server needs to process them. If the commands CREATE and DELETE are correctly processed the server informs the client with CREATED and DELETED messages and stays in the COMMANDS state. Otherwise, it informs the client with FAILED_CREATE and FAILED_DELETE messages and stays in COMMANDS state. For the MODIFY command the server calculates the path of the desired file on the server. Then, it sets the FileOutputStream with this file. After that the server switches to the DATA state. At the end it informs the client with READY_TO_RECEIVE message that it is ready now to receive data of the modified file.

If the server is in WELCOME_RECEIVER state, it receives a welcome message from the client. This message contains the client name, which is used by the server to create the client status file if it is not exist. At the end if the welcome message was correctly received, the server switches to the COMMANDS state. Otherwise, the client channels are closed.

```
1   private void handleClientMessage(ReceivingInfo ri) throws IOException,
2   InterruptedException {
3       Path filePathInServer = null;
4       String newFileName = null;
5       if (ri.getReceivingState().equals(ProcessingState.DATA)) {
6         try {
7           byte[] chunk = new byte[ri.getData().readableBytes()];
8           ri.addReceivedBytes(ri.getData().readableBytes());
9           ri.getData().readBytes(chunk);
10          fos.write(chunk);
11          fos.flush();
12          if (ri.getNumReceivedBytes() == ri.getFileSize()) {
13            fos.close();
14            sendMsg(ri.getChannel(), MODIFIED);
15            ri.writeToClientTable(event.toString());
16            ri.setReceivingState(ProcessingState.COMMANDS);
17          } else if (ri.getNumReceivedBytes() > ri.getFileSize()) {
18            ri.setNumReceivedBytes(0);
19            sendMsg(ri.getChannel(), FAILED_MODIFY);
20              ri.setReceivingState(ProcessingState.COMMANDS);
21          }
22        } catch (Exception e) {
23            Out.println("SERVER ERROR: error while receiving data");
24            ri.setNumReceivedBytes(0);
25            sendMsg(ri.getChannel(), FAILED_MODIFY);
26            ri.setReceivingState(ProcessingState.COMMANDS);
27        }
```

```
28      } else if (ri.getReceivingState().equals(ProcessingState.COMMANDS)) {
29          if (ri.getData().toString(CharsetUtil.UTF_8).startsWith(COMMAND)) {
30            Out.println("<--- " + ri.getData().toString(CharsetUtil.UTF_8));
31            eventInfos = ri.getData().toString(CharsetUtil.UTF_8).split(",");
32            event = new Event(Paths.get(eventInfos[1]), Long.valueOf(eventInfos[2]),
33            eventInfos[3], getEventType(eventInfos[4]),Long.parseLong(eventInfos[5]));
34            ri.setEvent(event);
35            boolean isDir = Boolean.valueOf(Files.isDirectory(event.getFilePath()));
36            switch (event.getEventType()) {
37            case CREATE:
38            try {
39                filePathInServer = ri.getFilePathInServer();
40                if (isDir && !Files.exists(filePathInServer)) {
41                  Files.createDirectory(filePathInServer);
42                  sendMsg(ri.getChannel(), CREATED);
43                  ri.writeToClientTable(event.toString());
44                }
45                break;
46            } catch (Exception e) {
47                Out.println("SERVER ERROR: error while creating file");
48                sendMsg(ri.getChannel(), FAILED_CREATE);
49                break;
50            }
51            case DELETE:
52              try {
53                filePathInServer = ri.getFilePathInServer();
54                ri.writeToClientTable(event.toString());
55                sendMsg(ri.getChannel(), DELETED);
56                break;
57              } catch (Exception e) {
58                Out.println("SERVER ERROR: error while deleting file");
59                sendMsg(ri.getChannel(), FAILED_DELETE);
60                break;
61              }
62            case MODIFY:
63              try {
64                filePathInServer = ri.getFilePathInServer();
65                String oldFileName = filePathInServer.getFileName().toString();
66                newFileName = oldFileName.split("\\.")[0] + "_" +
67                ri.getEvent().getLastModify() + "_" + ri.getName() + "." +
68                oldFileName.split("\\.")[1];
69                Path newfilePathInServer = Paths.get(filePathInServer.toString()
70                .replace(filePathInServer.getFileName().toString(), newFileName));
71                if (!Files.exists(newfilePathInServer)) {
72                  Files.createFile(newfilePathInServer);
73                }
74                ri.startReceiving(ri.getEvent().getFileSize());
75                fos = new FileOutputStream(newfilePathInServer.toFile());
76                ri.setReceivingState(ProcessingState.DATA);
77                sendMsg(ri.getChannel(), READY_TO_RECEIVE);
78              } catch (Exception e) {
79                Out.println("SERVER ERROR: error while modifying file");
80                sendMsg(ri.getChannel(), FAILED_MODIFY);
81                break;
82              }
83              break;
84            default:
85              break;
86            }
87          } else {
88            sendMsg(ri.getChannel(), INVALID_COMMAND);
89          }
90        } else if (ri.getReceivingState().equals(ProcessingState.WELCOME_RECEIVER)) {
91          Out.println("<-- from client_Hello: " + ri.getData().toString());
92          if (ri.getData().toString(CharsetUtil.UTF_8).startsWith(HELO_MSG)) {
93            Out.println("<--- " + ri.getData().toString(CharsetUtil.UTF_8));
94            String[] helloInfos = ri.getData().toString(CharsetUtil.UTF_8).split(" ");
```

```
95          ri.setName(helloInfos[1]);
96          ri.setHistoryPath();
97          if (!Files.exists(ri.getHistoryPath())) {
98            Files.createFile(ri.getHistoryPath());
99          }
100         statusFiles.putIfAbsent(ri.getName(), ri.getHistoryPath());
101         ri.setReceivingState(ProcessingState.COMMANDS);
102       } else {
103         Out.println("SERVER ERROR: welcome message is not correct in receiver");
104         SendingHandler.terminate();
105         terminate();
106       }
107     }
108     if (ReferenceCountUtil.refCnt(ri.getData()) > 0) {
109       ri.getData().release();
110     }
111 }
```

Listing 6.9: Method `handleClientMessage` on server receiving channel

Class SendingHandler

This handler is added to the `ChannelPipeline` of the second server channel (sending channel). It extends the `SimpleChannelInboundHandler<ByteBuf>` abstract class in Netty API. The main task of this class is to make all clients identical after any change. This process is performed by sending any modified file to all clients over their receiving channels. These channels and all sending information are stored in class `SendingInfo`. Method `channelActive` is overridden in class `SendingHandler` as can be seen in Listing 6.10. This method is invoked once the client connection request (receiving channel) is accepted. At this point, the client receiving channel is passed to `SendingInfo` constructor. All clients with sending information are stored in the map `sendingInfoMap <Channel, SendingInfo>`. The server welcomes the receiving channel of the newly connected client.

```
1   private class SendingHandler extends SimpleChannelInboundHandler<ByteBuf> {
2     @Override
3     public void channelActive(ChannelHandlerContext ctx) throws Exception {
4       SendingInfo si = new SendingInfo(ctx.channel());
5       sendingInfoMap.addSendingInfo(ctx.channel(), si);
6       sendMsg(ctx.channel(), HELO_MSG + " " + SERVER);
7     }
8     . . .
9   }
```

Listing 6.10: Overriding method `channelActive`

Method `channelRead0` illustrated in Listing 6.11 is invoked once a message is received from the receiving client channel. First, we check if the received `msg` is an instance of `ByteBuf`. If it is not the case the client channel is closed. Otherwise, the `SendingInfo` object is extracted to be passed to the method `handleClientMessage` in Listing 6.12 (see next).

```
1   private class SendingHandler extends SimpleChannelInboundHandler<ByteBuf> {
2     . . .
3     @Override
4     public void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
5     Exception {
6       if (!(msg instanceof ByteBuf)) {
7         Out.println("FATAL ERROR: invalid client message format");
8         ctx.channel().close();
9         return;
10      }
11      SendingInfo si = sendingInfoMap.getSendingInfo(ctx.channel());
12      si.setData((ByteBuf) msg);
13      handleClientMessage(si);
14    }
15    . . .
16  }
```

Listing 6.11: Overriding method channelRead

Method handleClientMessage in Listing 6.12 is used to handle all messages during sending process. These messages start with one of these keywords:

In SENDING_FILES state the keywords are:

- CREATED, MODIFIED, DELETED: to notify the server that the processed file is created, modified, or deleted on the client-side. After that the processed command is removed from the event queue and is written in the status file. CommandSender is now allowed to take the next existing command from the queue.
- FAILED_CREATE, FAILED_MODIFY, FAILED_DELETE: these messages mean the process failed. The command would be resent to the client if he is still active. otherwise, the server continues handling the rest of commands in the events queue.
- READY_TO_RECEIVE: is a message to notify the server that the client is ready to receive file data. Which are sent chunk by chunk by the server.

In WELCOME_SENDER state the keyword is:

- HELO_MSG: is a message to welcome the server once the client connection request is accepted. The client name is also received with this message and saved in sendingInfo. The status file path is created using the client name.

```
1   private void handleClientMessage(SendingInfo si) throws Exception {
2
3     byte[] chunkedfile = new byte[1024 * 1024];
4     ByteBuf buffer = null;
5     String[] clientMsg = si.getData().toString(CharsetUtil.UTF_8).split(" ");
6     if (si.getSendingState().equals(ProcessingState.SENDING_FILES)) {
7       switch (clientMsg[0]) {
8       case CREATED, MODIFIED, DELETED:
9         Out.println("<--- " + si.getData().toString(CharsetUtil.UTF_8));
```

```
10          wlock.lock();
11          try {
12            Pair<Channel, Event> pair = events.remove();
13            si.writeToClientTable(pair.getValue().toString());
14          } finally {
15            wlock.unlock();
16          }
17          commandSender.setNextCommand(true);
18          break;
19        case FAILED_CREATE, FAILED_MODIFY, FAILED_DELETE:
20          Out.println("<--- " + clientMsg[0]);
21          if (si.getChannel().isActive()) {
22            commandSender.setNextCommand(true);
23          } else {
24            Out.println("SERVER INFO: client [" + si.getName()
25            + "] is no longer online next command will be handled");
26            sendNextCommand();
27          }
28          break;
29        case READY_TO_RECEIVE:
30          Out.println("<--- " + si.getData().toString(CharsetUtil.UTF_8));
31          if (getFis() != null) {
32            Channel receiverChannel = null;
33            int r = getFis().read(chunkedfile);
34            while (r >= 0) {
35              buffer = Unpooled.copiedBuffer(chunkedfile, 0, r);
36              receiverChannel = si.getChannel();
37              if (receiverChannel.isActive()) {
38                send(si.getChannel(), buffer);
39                r = getFis().read(chunkedfile);
40              } else {
41                Out.println("SERVER INFO: client [" + si.getName()
42                + "] is no longer online next command will be handled");
43                sendNextCommand();
44              }
45            }
46              getFis().close();
47            }
48            break;
49          default:
50            sendNextCommand();
51            break;
52          }
53        } else if (si.getSendingState().equals(ProcessingState.WELCOME_SENDER)) {
54            if (clientMsg[0].startsWith(HELO_MSG)) {
55              Out.println("Update <--- " + si.getData().toString(CharsetUtil.UTF_8));
56              String[] helloInfos = si.getData().toString(CharsetUtil.UTF_8).split(" ");
57              si.setName(helloInfos[1]);
58              si.setHistoryPath();
59              sendingChannelsMap.put(helloInfos[1], si.getChannel());
60              si.setSendingState(ProcessingState.SENDING_FILES);
61            } else {
62              Out.println("SERVER ERROR: welcome message is not correct in sender");
63              ReceivingHandler.terminate();
64              terminate();
65            }
66        }
67  }
```

Listing 6.12: Handle client messages during the sending process

Class CommandSender

To synchronize files between clients, the server needs to send events (commands) to these clients. These events are queued in the events queue as pairs of <Channel, Event>.

```
final Queue<Pair<Channel, Event>> events = new ConcurrentLinkedQueue<>();
```

The pair is used to specify which event must be sent to which channel (client) at any time. `CommandSender` is a runnable class. Its run method is periodically executed at fixed rate, using `scheduleAtFixedRate` java class.

```
static final ScheduledExecutorService executor = Executors.newScheduledThreadPool(8);
```

This can be seen in Listing 6.13 in method `startSendingCommands`.

The main task of this class is to send the queued events to the corresponding clients. If the boolean variable `next_command` is set to true and the events queue is not empty, the first `Pair<Channel, Event>` is peeked from the queue. The client receiving channel and the event to be sent are specified. The keyword `COMMAND` is added to the event as a prefix. If the file is not a directory, the file path on the server-side is calculated and an InputStream `fis` is built on it. The help function `normalizeFilePathForClient` gets two parameters `filePath` on client-side and `clientName`. It returns the path of the same file on the server-side. At the end we check if the channel, through it the event should be sent, is still active the event is sent. Otherwise, this event is deleted, and the next event will be sent.

```
1   private class CommandSender {
2
3       public boolean next_command = true;
4
5       public void startSendingCommands() {
6         Runnable task = getRunnable();
7         executor.scheduleAtFixedRate(task, 0, 10, TimeUnit.SECONDS);
8       }
9
10      public CommandSendingTask getRunnable() {
11        return new CommandSendingTask();
12      }
13
14      public void setNextCommand(boolean next) {
15        rlock.lock();
16        try {
17          next_command = next;
18        } finally {
19          rlock.unlock();
20        }
21      }
22      public boolean getNextCommand() {
23        wlock.lock();
24        try {
25          return next_command;
26        } finally {
27          wlock.unlock();
28        }
29      }
30
31      private class CommandSendingTask implements Runnable {
32        @Override
33        public void run() {
34          Pair<Channel, Event> pair = null;
35          if (getNextCommand() && events.size() > 0) {
```

```
36          setNextCommand(false);
37          pair = events.peek();
38          Channel receiverChannel = pair.getKey();
39          Event event = pair.getValue();
40          Path filePath = event.getFilePath();
41          Long lastModificationTime = event.getLastModify();
42          String clientName = event.getClient();
43          String command = COMMAND + "," + event.toString();
44          String oldFileName = filePath.getFileName().toString();
45          boolean isDir = Boolean.valueOf(Files.isDirectory(filePath));
46          if (!isDir) {
47            String[] fnArr = oldFileName.split("\\.");
48            if (fnArr[0].contains("conflict")) {
49              fnArr[0] = fnArr[0].split("_")[0];
50            }
51            String newFileName = fnArr[0] + "_" + lastModificationTime + "_" +
52            clientName + "." + fnArr[fnArr.length - 1];
53            Path serverFilePath = normalizeFilePathForClient(filePath, clientName);
54            serverFilePath = Paths.get(serverFilePath.toString()
55                .replace(serverFilePath.getFileName().toString(), newFileName));
56            if (Files.exists(serverFilePath) && !isDir) {
57              while (true) {
58                try {
59                  InputStream fis = Files.newInputStream(serverFilePath);
60                  SendingHandler.setFis(fis);
61                  break;
62                } catch (IOException e) {
63                  try {
64                    Thread.sleep(1000);
65                    continue;
66                  } catch (InterruptedException e1) {
67                    Out.println("SERVER ERROR: ");
68                  }
69                }
70              }
71            }
72          }
73          if (receiverChannel.isActive()) {
74            sendMsg(receiverChannel, command);
75          } else {
76            Out.println("SERVER INFO: client [" + clientName
77            + "] is no longer online next command will be handled");
78            sendNextCommand();
79          }
80        }
81      }
82    }
83    private Path normalizeFilePathForClient(Path clientFilePath,String clientName) {
84      StringBuilder sb = new StringBuilder();
85      String fileSeparator = Pattern.quote(System.getProperty("file.separator"));
86      String[] serverPathItems = clientFilePath.toString().split(fileSeparator);
87      for (int i = 0; i < serverPathItems.length - 1; i++) {
88        if (serverPathItems[i].equals(clientName)) {
89          for (int j = i + 1; j < serverPathItems.length; j++) {
90            sb.append(serverPathItems[j]);
91            if (j != serverPathItems.length - 1) {
92              sb.append(SEP);
93            }
94          }
95        }
96      }
97      Path filePathInServer = Server.SAVING_DIR.resolve(Paths.get(sb.toString()));
98      return filePathInServer;
99    }
100 }
```

Listing 6.13: Class CommandSender on server-side

Class DeleteOldFiles

The main task of this class is to delete the old file copies of all files saved in the server (e.g., older than 7 days). This task is runnable. It is periodically executed using `scheduleAtFixedRate` java class as can be seen in method `startDeleting` in Listing 6.14. Iterating through all directories in the server is done in the run method. For each directory, `DeleteOldFiles` method is called. This method takes the directory path as a parameter and iterates through all files in it. All old file versions, but the last one, are deleted. Note: the last copy of all files should still be stored on the server, even if they are older than the specified time.

```
1 private class DeleteOldFiles {
2
3     public void startDeleting() {
4       executor.scheduleAtFixedRate(new Task(), 0, 7, TimeUnit.DAYS);
5     }
6
7     private class Task implements Runnable {
8       @Override
9       public void run() {
10          Out.println("Start deleting...");
11          try (Stream<Path> paths = Files.walk(SAVING_DIR)) {
12            paths.forEach(p -> {
13              if (Files.isDirectory(p)) {
14                deleteOldFiles(p);
15              }
16            });
17          } catch (IOException e) {
18            Out.println("SERVER ERROR: while iterating through server dir...");
19          }
20      }
21      . . .
22  }
```

Listing 6.14: Class `DeleteOldFiles` in the server

Class StatusFilesComparer

The main task of this class is the pairwise comparison of status files on the server-side. This process allows the server to know which events or in other words which files are not synchronized. This task is periodically executed as can be seen in Listing 6.15 in method `startComparison`. The desired result is achieving the identical state of the status files on the server-side. That means the clients should have the same synchronized file system at the end, except when a conflict occurs between them. When two or more clients have modified the same file, the server does not know which version of the file should be synchronized with other clients.

Solution of file conflict:

First, the server should recognize the conflict case and show an information message to the user. Let's assume that the file "f" is synchronized between clients c1 and c2. That means, c1 and c2 have the same version of the file "f". Both clients modify "f". That means there are now 3 versions of "f", the original one in the server and the other two, let's call them "c1_f" and "c2_f" are also stored in the server. The server has now two versions of the same file, which are not synchronized. The server creates two files "c1_f", and "c2_f" and sends them as new files to the clients so that c1 will have at the end the files "c1_f" and "c2_f". That means the versions "c1_f" will not be overridden by "c2_f" and vice versa for c2. Listing 6.15 illustrates the implementation of class `StatusFilesComparer`. First, the server makes sure that all clients are ready for the synchronization process. That means, all status files are in a stable state and there is no client in the mode of sending or receiving. If that is the case and the number of status files is greater than one, the pairwise comparison process between all status files starts in the server. The main idea here is that the comparison takes place in two stages.

The first phase: compares the status files of the two clients and extracts the list of commands (files) that must be synchronized with each of them.

The second phase: is to compare the results of the first phase. If two not synchronized versions of the same file are detected, a conflict situation is reported. The server creates two new files (e.g., conflict in file `f.txt`, then the file `"f_confilct_client1.txt"` is sent to client2, `"f_confilct_client2.txt"` is sent to client1). At the end, the result is queued in the events queue to be sent by the `CommandSender` that we have seen in the previous section.

```
1   private class StatusFilesComparer {
2
3       public void startComparison() {
4         executor.scheduleAtFixedRate(new Task(), 0, 15, TimeUnit.SECONDS);
5       }
6
7       private class Task implements Runnable {
8
9         @Override
10        public void run() {
11          try {
12            if (events.size() == 0) {
13              Out.println("STATUS FILES ARE COMPARED NOW");
14              List<String> clientNames = statusFiles.keySet().stream().toList();
15
16              for (String clientName1 : clientNames) {
17                Out.println(clientName1);
```

```
18              Set<Event> eventList1 = new TreeSet<>();
19              List<String> lastChanges1 = new ArrayList<>();
20              List<String> statusAllLines1 = new ArrayList<>();
21              Set<Event> toSyncEvents = new TreeSet<>();
22              Path f1 = statusFiles.get(clientName1);
23              statusAllLines1 = Files.readAllLines(f1);
24              lastChanges1 = getLastChanges(f1);
25
26              for (String clientName2 : clientNames) {
27                if (!clientName1.equals(clientName2)) {
28                  Out.println("----" + clientName2);
29                  Set<Event> eventList2 = new TreeSet<>();
30                  List<String> lastChanges2 = new ArrayList<>();
31                  List<String> statusAllLines2 = new ArrayList<>();
32                  Path f2 = statusFiles.get(clientName2);
33                  statusAllLines2 = Files.readAllLines(f2);
34                  lastChanges2 = getLastChanges(f2);
35
36                  for (String lg2 : lastChanges2) {
37                    String[] ls = lg2.split(",");
38                    Event e = new Event(Paths.get(ls[0]), Long.valueOf(ls[1]), ls[2],
39                        getEventType(ls[3]), Long.parseLong(ls[4]));
40                    if (!statusAllLines1.contains(lg2)) {
41                      eventList1.add(e); // what is missing from client i
42                    }
43                  }
44
45                  for (String lg1 : lastChanges1) {
46                    String[] ls = lg1.split(",");
47                    Event e = new Event(Paths.get(ls[0]), Long.valueOf(ls[1]), ls[2],
48                        getEventType(ls[3]), Long.parseLong(ls[4]));
49                    if (!statusAllLines2.contains(lg1)) {
50                      eventList2.add(e); // what is missing from client j
51                    }
52                  }
53
54                  if (eventList1.size() == 0 && eventList2.size() == 0) {
55                    Out.println("---- NO CHANGES between " + clientName1 + " and " +
56                    clientName2 + " ----");
57                  }
58
59                  // compare the results for conflicts
60                  for (Event e1 : eventList1) {
61                    Event e1Copy = (Event) e1.clone();
62                    for (Event e2 : eventList2) {
63                      if (e1.getFilePath().getFileName().equals(e2.getFilePath()
64                      .getFileName())) {
65                        Out.println("CONFLICT: between client " + clientName1 + "
66                        and "+ clientName2 + " there are more than one version of
67                        file " + e1.getFilePath());
68                          String fn1 = e1.getFilePath().getFileName().toString();
69                          String[] fnarr1 = e1.getFilePath().getFileName().toString()
70                          .split("\\.");
71                          Path p1 = Paths.get(e1.getFilePath().toString().replace(fn1,
72                          fnarr1[0] + "_conflict_" + clientName2 + "." + fnarr1[1]));
73                          e1Copy.setFilePath(p1);
74                      }
75                    }
76                    toSyncEvents.add(e1Copy);
77                  }
78                }
79              } // end comparing client i with all clients
80
81              if (toSyncEvents.size() > 0) {
82                Out.println("NEXT FILES MUST BE SENT TO: " + clientName1);
83                toSyncEvents.forEach(p -> Out.println("command:      " + p));
84              }
```

```
85
86                // events to be sent to client i
87                if (clientName1 != null && sendingChannelsMap.get(clientName1) != null
88                && sendingChannelsMap.get(clientName1).isActive()) {
89                  wlock.lock();
90                  try {
91                    for (Event e : toSyncEvents) {
92                      Pair<Channel, Event> p = new
93                      Pair<>(sendingChannelsMap.get(clientName1), e);
94                      if (!events.contains(p)) {
95                        events.add(p);
96                      }
97                    }
98                    if (events.size() > 0) {
99                      Out.println("number of files to be sent: " + events.size());
100                   }
101                 } finally {
102                   wlock.unlock();
103                 }
104               }
105             }
106           }
107         } catch (Exception e) {
108           Out.println("SERVER ERROR: during compairing the status files");
109         }
110       }
111     }
112   }
```

Listing 6.15: Class `StatusFileComparer`

Client components

In this section, we will see the most important client components.

Enumeration ClientState

The client has the same server states, see Listing 6.16. `WELCOME_RECEIVER` state is to welcome the receiving channel of the server. `WELCOME_SENDER` state is to welcome the sending channel of the server. `COMMANDS` state is to receive commands from the server and to organize the syncing process. `DATA` state is to receive files data. `SENDING_FILES` state is to send commands and files to the server.

```
1   public enum ClientState {
2       WELCOME_RECEIVER, WELCOME_SENDER, COMMANDS, DATA, SENDING_FILES;
3   }
```

Listing 6.16: Modes in which the client can be found

Class History

This class is used by the client to save his status. In other words, the changes have been performed on his files. This class needs just the status file path of the client, which is passed to the constructor method. Listing 6.17 illustrates class fields, `statusFilePath`

represents the path of the status file of the client, and `lock` is used to handle thread scheduling while writing into the status file.

```
1   public class History {
2
3     private final Path statusFilePath;
4     private final Lock lock;
5
6     public History(Path statusFilePath) {
7       this.statusFilePath = statusFilePath;
8       this.lock = new ReentrantLock();
9     }
10    . . .
11  }
```

Listing 6.17: Class `History` to manage the client status file

Class ReceivingHandler and class SendingHandler

Since the client should be able to receive from and send files to the server. It is necessary to implement these two handler classes the same way like for the server-side. The underlying ideas for these two handler classes are the same ones implemented in the server. There are some differences. So, the received files on the client-side must not be synchronized with all other clients. The client receives data only from the server and the server receives data from all clients.

Class FileWatcher

This class is used by the client to monitor his folder registered to be synchronized. When the client creates, modifies, or deletes any file stored in his monitored directory, the `FileWatcher` creates an appropriate event. This event is pushed to the events queue to be sent to the server by the client `CommandSender`. Listing 6.18 illustrates the implementation of class `FileWatcher`. Let's begin with the class fields:

`terminate`: is a boolean variable used to terminate the watching process when it is set to true.

`watcher`: is a java watch service that watches registered objects for changes and events [8]. The watcher is set in the constructor.

`key`*:* is a java watch key created when a watchable object is registered with a watch service. When an event is detected, the key is signaled and queued so that it can be retrieved by invoking the watch service's `poll` or `take` methods [8].

`fileFromServer`: used to distinguish between files changed by the client or sent by the server in the synchronization process, which don't have to be watched.

`fileTimeStamps`: is a `Map` of <`Path`, `Long`>. It stores the last modification time for each processed path as a long value. It is used to guarantee that at any time only one event will be watched.

Method `start` is used to execute method `watch` in a new thread. Method `terminate` is used to terminate `FileWatcher` by setting field `terminate` to true and closing the `watcher`. Method `setServerFile` is used to set `fileFromServer` when the client monitored directory is changed by the server. Method `watch` is used to watch the client directory registered with the `watcher`. Map `keysMap` stores the keys of all registered inner directories in the main client folder. In line (59) the `watcher` waits for changes. Method `pollEvents` retrieves and removes all pending events for the watch key, returning a list of the events that were retrieved. When registering an object with the watch service, you specify the types of events that you want to monitor [8]. The standard event types are `ENTRY_CREATE, ENTRY_DELETE, and ENTRY_MODIFY`. In all cases, a new event is created and pushed into the events queue to be sent to the server by the client `CommandSender`. In the `finally` block the watch key is reset.

```
1   private class FileWatcher {
2
3       private static volatile boolean terminate;
4       private WatchKey key;
5       private WatchService watcher;
6       private volatile boolean fileFromServer = false;
7       private Map<Path, Long> fileTimeStamps;
8
9       public FileWatcher() throws IOException {
10        this.watcher = FileSystems.getDefault().newWatchService();
11        this.fileTimeStamps = new ConcurrentHashMap<>();
12      }
13
14      public void startWatching() {
15        new Thread(() -> {
16          try {
17            watch();
18          } catch (InterruptedException | IOException e) {
19            Out.println("CLIENT ERROR: while watching client file system");
20          }
21        }).start();
22      }
23
24      public void terminate() throws IOException {
25        terminate = true;
26        watcher.close();
27        Thread.currentThread().interrupt();
28      }
29
```

```
30      public void setServerFile(boolean fileFromServer) {
31        rlock.lock();
32        try {
33          this.fileFromServer = fileFromServer;
34        } finally {
35          rlock.unlock();
36        }
37      }
38
39      public boolean isServerFile() {
40        wlock.lock();
41        try {
42          return fileFromServer;
43        } finally {
44          wlock.unlock();
45        }
46      }
47
48      @SuppressWarnings({ "unchecked" })
49      private void watch() throws InterruptedException, IOException {
50        key = folderPath.register(watcher, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
51        Map<Path, WatchKey> keysMap = new HashMap<>();
52        keysMap.put(folderPath, key);
53        while (!terminate) {
54          try {
55            Event command = null;
56            boolean dir = false;
57            long oldFileModifiedTimeStamp = 0L;
58            long newFileModifiedTimeStamp = 1L;
59            key = watcher.take(); // blocking
60            List<WatchEvent<?>> watchedEvents = key.pollEvents();
61            for (int i = 0; i < watchedEvents.size(); i++) {
62              WatchEvent<Path> pEvent = (WatchEvent<Path>) watchedEvents.get(i);
63              Path relPath = pEvent.context();
64              Path dirPath = (Path) key.watchable();
65              Path filePath = dirPath.resolve(relPath);
66              if (!filePath.getFileName().toString().equals(".status")) {
67                dir = filePath.toFile().isDirectory() ? true : false;
68                oldFileModifiedTimeStamp = fileTimeStamps.get(filePath) == null ? 0
69                : fileTimeStamps.get(filePath);
70                newFileModifiedTimeStamp =
71                Files.getLastModifiedTime(filePath).toMillis();
72                if (!isServerFile() && newFileModifiedTimeStamp >
73                oldFileModifiedTimeStamp) {
74                  Out.println("filePath = " + filePath);
75                  Out.println("oldFileModifiedTimeStamp = " + oldFileModifiedTimeStamp
76                  + "\nnewFileModifiedTimeStamp = " + newFileModifiedTimeStamp);
77                  PathMatcher javaMatcher =
78                  FileSystems.getDefault().getPathMatcher("glob:**.{*}");
79                  if (pEvent.kind() == ENTRY_CREATE) {
80                    if (javaMatcher.matches(filePath.getFileName())) {
81                      command = new Event(filePath,
82                      Files.getLastModifiedTime(filePath).toMillis(),
83                      name, EventType.MODIFY, Files.size(filePath));
84                      wlock.lock();
85                      try {
86                        if (!events.contains(command) && Files.size(filePath) > 0) {
87                          events.add(command);
88                        }
89                      } finally {
90                        wlock.unlock();
91                      }
92                    } else if (pEvent.kind() == ENTRY_CREATE && dir) {
93                      WatchKey k = filePath.register(watcher, ENTRY_CREATE,
94                      ENTRY_DELETE, ENTRY_MODIFY);
95                      keysMap.put(filePath, k);
96                      command = new Event(filePath,
```

```
97                    Files.getLastModifiedTime(filePath).toMillis(),
98                    name, EventType.CREATE, Files.size(filePath));
99                    wlock.lock();
100                   try {
101                     if (!events.contains(command)) {
102                       events.add(command);
103                     }
104                   } finally {
105                     wlock.unlock();
106                   }
107                 }
108               } else if (pEvent.kind() == ENTRY_DELETE) {
109                 for (Path p : keysMap.keySet()) {
110                   if (!Files.exists(p)) {
111                     keysMap.get(p).cancel();
112                   }
113                 }
114                 newFileModifiedTimeStamp = System.currentTimeMillis();
115                 command = new Event(filePath, newFileModifiedTimeStamp, name,
116                 EventType.DELETE, 0L);
117                 wlock.lock();
118                 try {
119                   if (!events.contains(command)) {
120                     events.add(command);
121                   }
122                 } finally {
123                   wlock.unlock();
124                 }
125               } else if (pEvent.kind() == ENTRY_MODIFY) {
126                 if (!Files.isDirectory(filePath)) {
127                   command = new Event(filePath,
128                   Files.getLastModifiedTime(filePath).toMillis(), name,
129                   EventType.MODIFY, Files.size(filePath));
130                   wlock.lock();
131                   try {
132                     if (!events.contains(command)) {
133                       events.add(command);
134                     }
135                   } finally {
136                     wlock.unlock();
137                   }
138                 }
139               }
140               fileTimeStamps.put(filePath,
141               Files.getLastModifiedTime(filePath).toMillis());
142             }
143           } else {
144             resetKey();
145           }
146         }
147       } catch (Exception cwse) {
148         terminate();
149         break;
150       } finally {
151         resetKey();
152       }
153     }
154   }
155
156   public void resetKey() {
157     if (key != null) {
158       key.reset();
159     }
160   }
161 }
```

Listing 6.18: Class `FileWatcher` on client-side

6.2.2. Server program

In this section, the server program implementation is illustrated. All server components mentioned above are put together. This is done in method `start` in Listing 6.19. The process starts with invoking all periodical tasks. Method `startSendingCommands` in class `CommandSender`, which is responsible for sending the events that were pushed in the event queue. Method `startComparison` in class `StatusFilesComparer`, which is responsible for comparing the status files on the server. Method `startDeleting` in class `DeleteOldFiles`, which is responsible for deleting the old version of synchronized files. After that, the `EventLoopGroups` are created. Two server channels are created and initialized. `receivingChannel` is responsible for receiving events and data from clients. `sendingChannel` is responsible for sending events and data to the clients. The `ChannelPipeline` of newly connected clients (child channels) is initialized with classes `ReceivingTimeoutHandler` and `ReceivingHandler`. At the end, the `receivingChannel` is bind with the port (`PORT1: 5000`) as can be seen (20). Method `sync` is called for waiting until the binding process is completed. Method `channel` returns the desired `receivingChannel` of the server. For the `sendingChannel,` the same steps are done. But with `SendingTimeoutHandler` and `SendingHandler`. This channel is bind with the port (`PORT2: 5001`). The program blocks (37). By typing any key, the program is terminated, and the server channels are closed. In the `finally` block the `EventLoopGroups` are shutdown.

```
1   private void start() throws Exception {
2       statusFilesComparer.startComparison();
3       commandSender.startSendingCommands();
4       deleteOldFiles.startDeleting();
5       EventLoopGroup bossGroup = new NioEventLoopGroup(1);
6       EventLoopGroup workerGroup = new NioEventLoopGroup();
7       try {
8         Channel receivingChannel = new ServerBootstrap()
9           .group(bossGroup, workerGroup)
10          .channel(NioServerSocketChannel.class)
11          .childHandler(new ChannelInitializer<SocketChannel>() {
12              @Override
13              protected void initChannel(SocketChannel clientChannel1) throws
14              Exception {
15                clientChannel1.pipeline().addLast("receivingTimeoutHandler",
16                new ReceivingTimeoutHandler(5));
17                clientChannel1.pipeline().addLast("messageHandler", new
18                ReceivingHandler());
19              }
20            }).bind(PORT1).sync().channel();
21
22        Channel sendingChannel = new ServerBootstrap()
23          .group(bossGroup, workerGroup)
24          .channel(NioServerSocketChannel.class)
```

```
25          .childHandler(new ChannelInitializer<SocketChannel>() {
26              @Override
27              protected void initChannel(SocketChannel clientChannel2) throws
28              Exception {
29                clientChannel2.pipeline().addLast("sendingTimeoutHandler", new
30                SendingTimeoutHandler(5));
31                clientChannel2.pipeline().addLast("sendingHandler", new
32                SendingHandler());
33              }
34            }).bind(PORT2).sync().channel();
35
36          Out.println("Server terminate by typing any key...");
37          In.readLine();
38          terminate();
39          receivingChannel.close().syncUninterruptibly();
40          sendingChannel.close().syncUninterruptibly();
41
42        } finally {
43          bossGroup.shutdownGracefully();
44          workerGroup.shutdownGracefully();
45        }
46      }
```

Listing 6.19: Server program

6.2.3. Client program

In this section, the client program implementation is illustrated. All client components mentioned above are put together. This is done in method `start` in Listing 6.20. The process starts with invoking all periodical tasks. Method `startSending` in class `CommandSender`, which is responsible for sending the events that were pushed in the event queue. Method `startWatching` in class `FileWatcher`, which is responsible for monitoring the client file system. After that, an `EventLoopGroup` is created. Two client channels are then created and initialized. `sendingChannel` is responsible for sending events and data to the server. `receivingChannel` is responsible for receiving events and data from the server. The `ChannelPipeline` of server channels is initialized with classes `SendingHandler` and `ReceivingHandler`. At the end, the `sendingChannel` is connected with the port (`PORT1: 5000`). For the `receivingChannel` the same steps are done. But it is initialized with the `ReceivingHandler`. This channel is connected to the port (`PORT2: 5001`). The main thread of the program blocks in line (29). By typing any key, the program is terminated, and `CommandSender` and `FileWatcher` are closed. In `finally` block the `EventLoopGroup` is shutdown.

```
1   private void start() throws Exception {
2       SendingHandler sendingHandler = new SendingHandler();
3       ReceivingHandler receivingHandler = new ReceivingHandler();
4       EventLoopGroup group = new NioEventLoopGroup();
5       try {
6         Channel sendingChannel = new Bootstrap()
7           .group(group)
```

```
8           .channel(NioSocketChannel.class)
9           .handler(new ChannelInitializer<SocketChannel>() {
10            @Override
11            protected void initChannel(SocketChannel clientChannel) throws Exception
12            {
13                clientChannel.pipeline().addLast(sendingHandler);
14              }
15            }).connect(HOST, PORT1).sync().channel();
16
17        Channel receivingChannel = new Bootstrap()
18          .group(group)
19          .channel(NioSocketChannel.class)
20          .handler(new ChannelInitializer<SocketChannel>() {
21              @Override
22              protected void initChannel(SocketChannel clientChannel) throws Exception
23              {
24                clientChannel.pipeline().addLast(receivingHandler);
25              }
26            }).connect(HOST, PORT2).sync().channel();
27
28        Out.println("Client application terminate with typing any key...");
29        In.readLine();
30        commandSender.terminate();
31        fileWatcher.terminate();
32        sendingChannel.close().syncUninterruptibly();
33        receivingChannel.close().syncUninterruptibly();
34
35      } finally {
36        group.shutdownGracefully();
37      }
38    }
```

Listing 6.20: Client program

7 Conclusion

The synchronous blocking approach is not effective for client-server models due to waiting times for responses, thus resulting in waste of resources and reduction of performance. In contrast, the asynchronous non-blocking approach is more efficient, especially for request/response models, where the response time, the performance, and the utilization of the resources are very important.

Synchronization of files and keeping a history of changes are critical parts of any cloud-based file synchronization system. This thesis has described an implementation of a simple file synchronization system based on the Netty framework. Netty is a Java framework based on Java NIO network communication protocol. Which allows an asynchronous non-blocking input and output. The Netty framework is used for developing of maintainable high-performance client-server systems. It provides an architecture, which separates the business logic from the network logic of applications. Netty also provides the possibility to separate the tasks of processing incoming and outgoing data into several stages, which contributes significantly and maximizes code reusability.

The main goal of the thesis was the analysis and evaluation of the Netty non-blocking approach for the implementation of client-server systems. As a case study for evaluation, a file synchronization system has been implemented. This system is a client-server (request-response) model. The files changed by any client are stored on the server storage. After that, the server synchronizes these files with the rest of the clients. This system has the ability to recognize the conflict cases, that occurs when two or more versions of the same file have to be synchronized at the same time. The case study has shown that Netty is an effective and convenient Java framework for the rapid development of client-server systems.

References

[1] “Khan, MD Ibrahim, et al. "Using blockchain technology for file synchronization." IOP Conference Series: Materials Science and Engineering. Vol. 561. No. 1. IOP Publishing, 2019”.

[2] “Tridgell, Andrew, and Paul Mackerras. "The rsync algorithm." (1996)”.

[3] “Z. Li, Z. Zhang and Y. Dai, "Coarse-grained cloud synchronization mechanism design may lead to severe traffic overuse," in Tsinghua Science and Technology, vol. 18, no. 3, pp. 286-297, June 2013, doi: 10.1109/TST.2013.6522587”.

[4] “Dropbox,” [Online]. Available: https://www.dropbox.com/features/sync.

[5] “S. Li, Q. Zhang, Z. Yang and Y. Dai, "Understanding and Surpassing Dropbox: Efficient Incremental Synchronization in Cloud Storage Services," 2015 IEEE Global Communications Conference (GLOBECOM), 2015, pp. 1-7, doi: 10.1109/GLOCOM.2015.7417235”.

[6] “Netty project,” [Online]. Available: https://netty.io/.

[7] Norman Maurer and Marvin Allen Wolfthal. Netty in Action. Manning Publications, 2015. ISBN 9781617291470.

[8] “https://docs.oracle.com/javase/7/docs/api/java,” [Online].

[9] “TutorialPoint,” [Online]. Available: https://www.tutorialspoint.com/software_engineering/software_requirements.htm.

[10] “Mendix,” [Online]. Available: https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/.