EE4210 Computer Network II

Assignment 1 Report

SyncPeer

File Synchronization Application

by

Luan Wenhao A0119541J

National University of Singapore

Semester 2, AY2015/2016

Summary

SyncPeer is a peer-to-peer file synchronization application with a command line interface. SyncPeer is able to synchronize files in a specified folder with another SyncPeer process running on the same or different machines through TCP connection. It is robust to network interruption and unstable Internet conditions, ensuring the integrity of data to the maximum possible extent. It is developed with Java core libraries and no external dependency is required. It fulfills the assignment requirement.

Table of Contents

Introduction

SyncPeer is designed and implemented to achieve peer-to-peer file synchronization over TCP connection. It is part of EE4210 Assignment 1 and it is developed to fulfill assignment requirements with certain pre-defined assumptions.

Requirements

1. The synchronization will result in both peers having the same files.
2. No duplicate transmission of shared files is allowed.
3. Peer should be able to specify the other peer's IP.
4. The application should be robust to abrupt interruptions.
5. Application-layer protocol is a must.

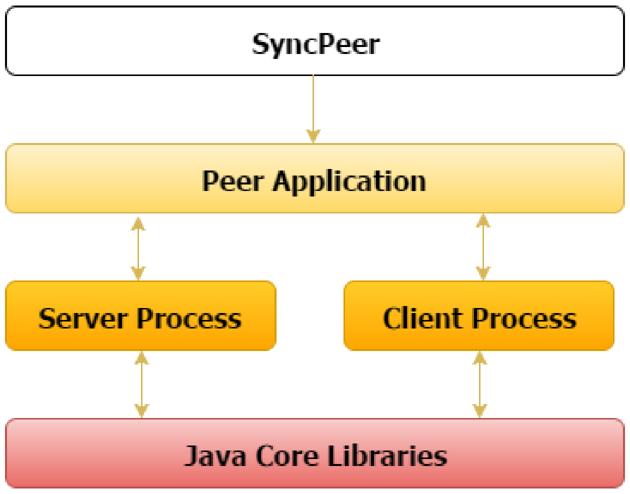
Assumptions

1. Files with the same names are considered the same.
2. One-to-One peer connection only.
3. IP address of the peer to connect is known.
4. Files in shared folders will not be deleted during the process.

Application Design and Implementation

The design and implementation of this application is discussed in following three sections, introducing the architecture, the class diagram and the sequence diagram respectively, followed by explanations on the design choices.

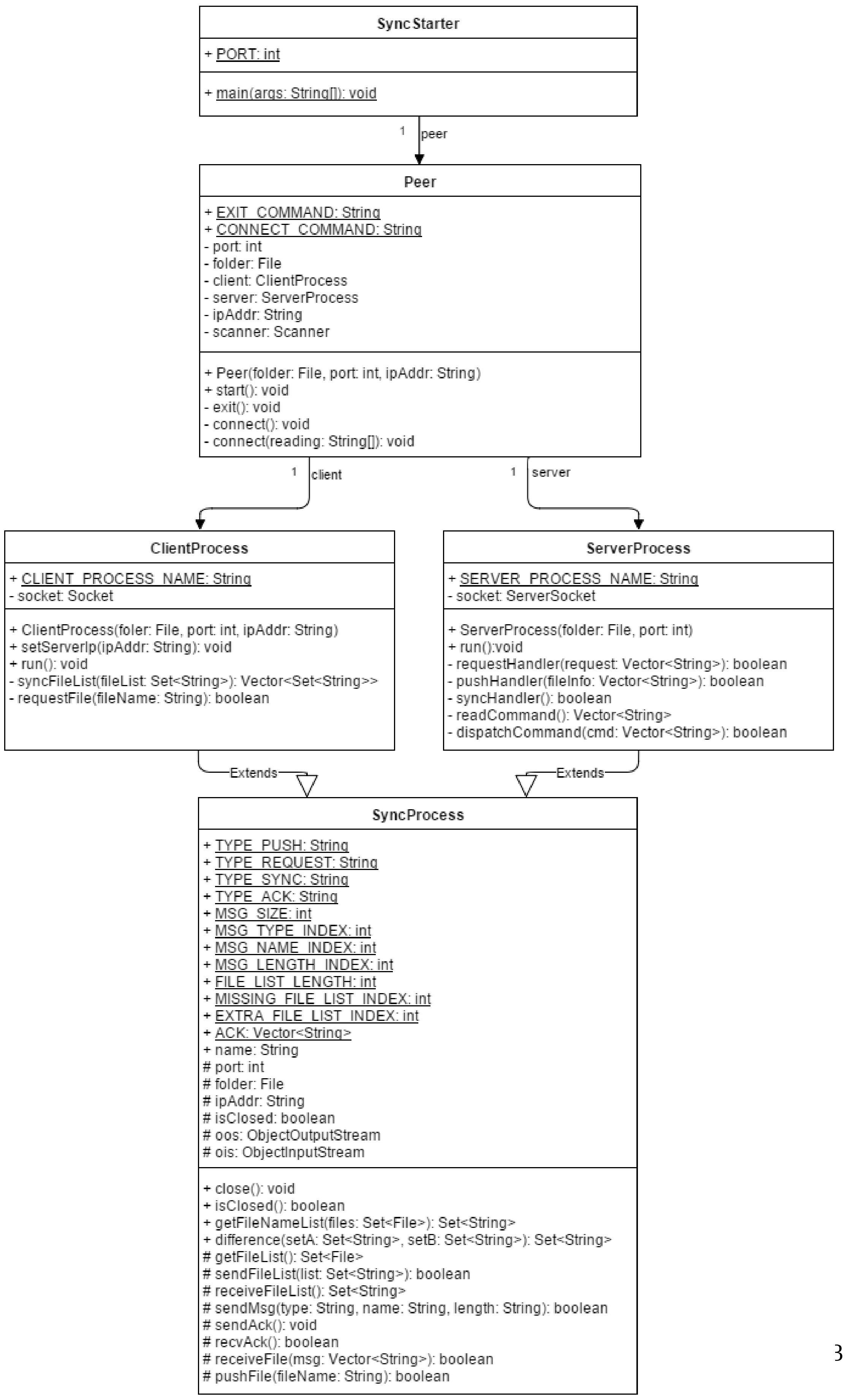
System Architecture



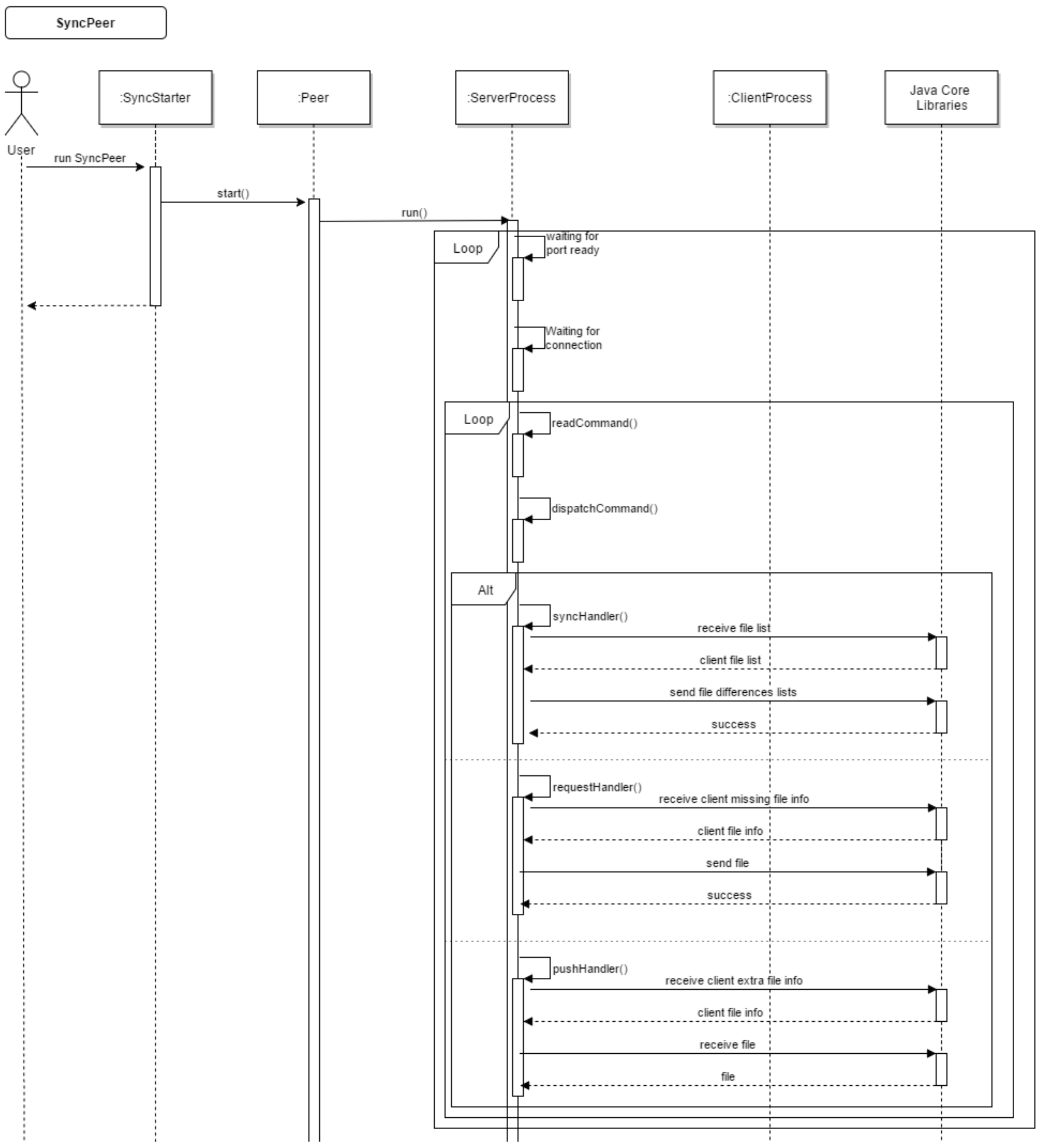
The service provided by SyncPeer is based on 4-layer architecture. SyncPeer Starter program serves as a loader and it exits after loading a peer application. The peer application keeps a server process and a client process for the actual synchronization logic. The peer application interacts with users and dispatch user commands to the two processes. The two processes are running in parallel with each other on different threads and they call Java core libraries functions to handle all the networking requests and responses.

The use of a loader to load the actual peer application is designed to add adaptability and extendibility to SyncPeer. In different circumstances users might have different requirements on the interaction with SyncPeer or different configurations so as to adapt to various platforms. Using a loader to configure and launch the actual peer application enables platform-related modifications without exposing the core of SyncPeer, minimizing its vulnerability to injection attacks. On the other hand, third-party modules can be integrated into the loader to interact with SyncPeer, making SyncPeer more extendible.

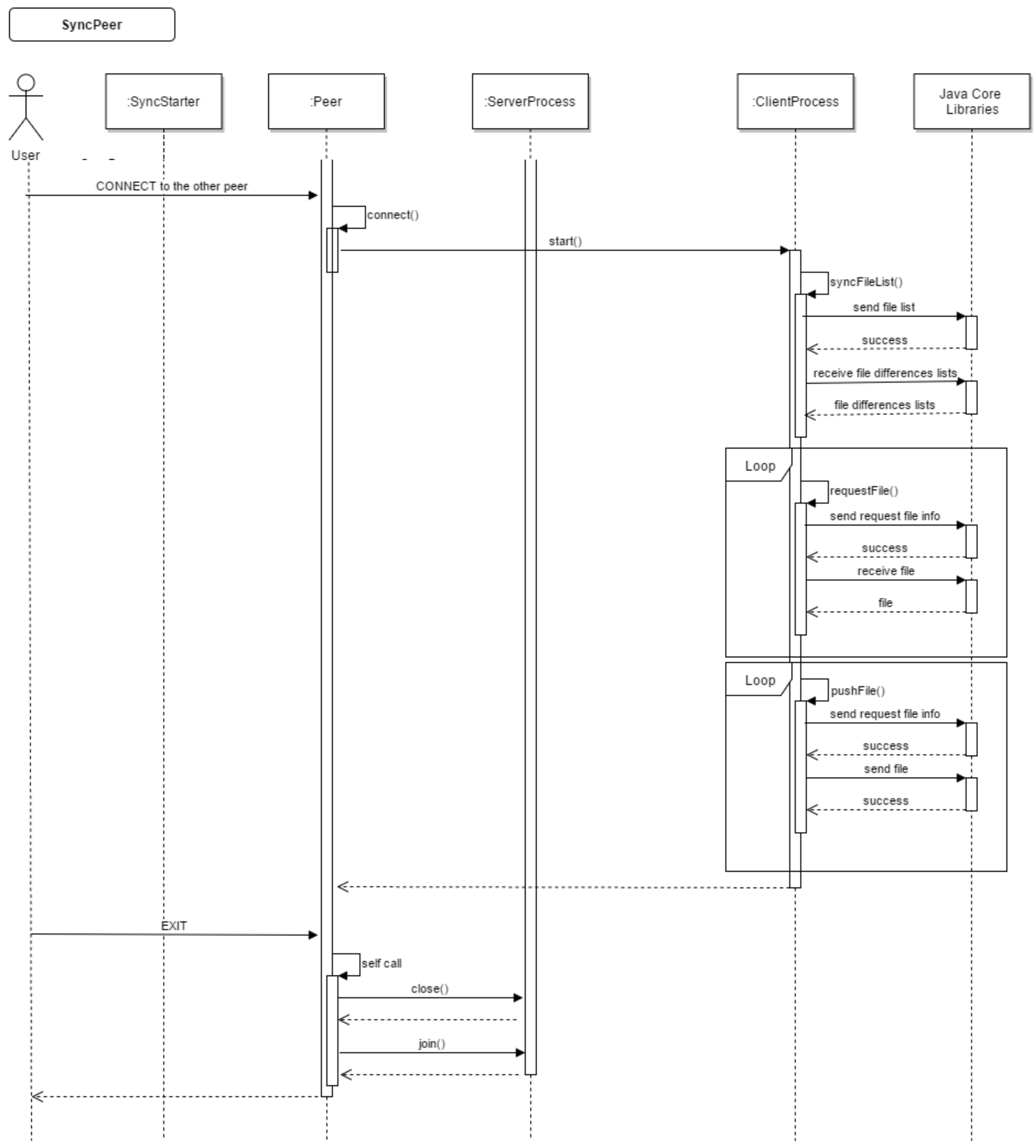
Class Diagram


SyncStarter
+ PORT: int
+ main(args: String[]): void
1
peer
Peer
+ EXIT_COMMAND: String
+ CONNECT_COMMAND: String
- port: int
- folder: File
- client: ClientProcess
- server: ServerProcess
- ipAddr: String
- scanner: Scanner
+ Peer(folder: File, port: int, ipAddr: String)
+ start(): void
- exit(): void
- connect(): void
- connect(reading: String[]): void
1
client
1
server
ClientProcess
+ CLIENT_PROCESS_NAME: String
- socket: Socket
+ ClientProcess(foler: File, port: int, ipAddr: String)
+ setServerIp(ipAddr: String): void
+ run(): void
- syncFileList(fileList: Set<String>): Vector<Set<String>>
- requestFile(fileName: String): boolean
ServerProcess
+ SERVER_PROCESS_NAME: String
- socket: ServerSocket
+ ServerProcess(folder: File, port: int)
+ run():void
- requestHandler(request: Vector<String>): boolean
- pushHandler(fileInfo: Vector<String>): boolean
- syncHandler(): boolean
- readCommand(): Vector<String>
- dispatchCommand(cmd: Vector<String>): boolean
Extends
Extends
SyncProcess
+ TYPE_PUSH: String
+ TYPE_REQUEST: String
+ TYPE_SYNC: String
+ TYPE_ACK: String
+ MSG_SIZE: int
+ MSG_TYPE_INDEX: int
+ MSG_NAME_INDEX: int
+ MSG_LENGTH_INDEX: int
+ FILE_LIST_LENGTH: int
+ MISSING_FILE_LIST_INDEX: int
+ EXTRA_FILE_LIST_INDEX: int
+ ACK: Vector<String>
+ name: String
# port: int
# folder: File
# ipAddr: String
# isClosed: boolean
# oos: ObjectOutputStream
# ois: ObjectInputStream
+ close(): void
+ isClosed(): boolean
+ getFileNameList(files: Set<File>): Set<String>
+ difference(setA: Set<String>, setB: Set<String>): Set<String>
# getFileList(): Set<File>
# sendFileList(list: Set<String>): boolean
# receiveFileList(): Set<String>
# sendMsg(type: String, name: String, length: String): boolean
# sendAck(): void
# recvAck(): boolean
# receiveFile(msg: Vector<String>): boolean
# pushFile(fileName: String): boolean

Sequence Diagram



(Continue on next page)


SyncPeer
User
:SyncStarter
:Peer
:ServerProcess
:ClientProcess
Java Core Libraries
CONNECT to the other peer
connect()
start()
syncFileList()
send file list
success
receive file differences lists
file differences lists
Loop
requestFile()
send request file info
success
receive file
file
Loop
pushFile()
send request file info
success
send file
success
EXIT
self call
close()
join()

Explanation

The loader, SyncStarter class, contains the entry to the program and it sets up a Peer. After launching Peer in a new thread, SyncStarter exits and hands over the control to Peer.

Once Peer takes control, it initializes a server thread from ServerProcess and runs it in the background. Peer starts reading user inputs and handling them accordingly. The available two commands are defined in the constants “EXIT_COMMAND” and “CONNECT_COMMAND”. Exit is done by closing all open sockets and interrupts existing synchronization, followed by joining server threads to clean up background tasks. The SyncPeer application will be terminated by the exit command. Connect command establishes connection from Peer to a remote server and starts synchronization. It is done by instantiating a new client thread and passing on the server IP address. Peer closes connection on the finish of synchronization and it will wait for user commands to either start a new synchronization session or terminate.

ServerProcess runs in background and handles client connections and synchronizations. It keeps listening to a designated port and once a client connection is established, the synchronization begins. Once the connection is established, ServerProcess starts listening to commands sent from the connected Peer, a.k.a. the remote client. Received commands will be dispatched according to their types and handlers call Java core libraries to reply or send files to the remote client.

ClientProcess is instantiated every time a new connection command is input by users. It starts synchronization by connecting to the server, exchange file lists, requesting client missing files and pushing server missing files.

Both ServerProcess and ClientProcess are child class of an abstract class SyncProcess. SyncProcess extends Thread class and implements many constants, class members and class methods that are used by the two child classes. SyncProcess also implements some methods that define the same behavior of its child classes, such as “close()” to close sockets and terminate ongoing connections and “isClosed()” to check whether the “close()” method has been called.

Communication Protocol

A communication protocol is defined to allow Peers to communicate and exchange information. Initially design of the protocol was very complicated and in the latest version, it was simplified to a great extent so as to increase human-readability and improve efficiency.

Basic Rule

Client Peer initiates requests by sending commands to server Peer. Server Peer responds with requested information or the requested file.

Blocking Rule

All commands and file transfers should be acknowledged by an acknowledgement. Process should be blocked until acknowledgements are received.

Command Format

A *Vector* of *String*, interpreted as follows:

Command Type	File Name	File Length

where "File Name" and "File Length" can be empty *String* if not needed.

Available command types are:

TYPE_PUSH – Push a file to server Peer

TYPE_REQUEST – Request a file from server Peer

TYPE_SYNC – Request exchanging file lists with server Peer

TYPE_ACK – Acknowledge upon receiving of commands or files.

Special Rule on File List Sync

The file list sync results returned from server Peer are two *Set* of *String* containing file names of client missing files and server missing files. The rule states that server Peer sends the client missing file lists first, followed by server missing file lists. The two transfers are viewed as two individual file transfers.

Problem-Solving

Port-Binding on a Same Machine

ServerProcess is incorporated with port binding collision avoidance system. If, during the process of port-binding, the designated port is occupied by another process, or other Peers running on the same machine, it will keep trying instead of prompting errors so as to allow ClientProcess to function normally. It enables testing and file synchronization on a same machine through localhost.

Duplicate Codes on Client and Server

In early versions, ClientProcess and ServerProcess were independently developed. It resulted in two complicated classes with plenty of duplicate codes on communication and file transfer. It was solved by creating an abstract class, SyncProcess, to set up an abstraction over the two independent classes.

Choice of Socket Streams

All communications and file transfers are implemented with Java socket ObjectInputStream and ObjectOutputStream. Although lower level streams, such as ByteArrayInputStream and ByteArrayOutStream, do exist and might also work well, object streams tend to be more convenient in handling both commands and files. Object translation is encapsulated in the stream, which also makes the use of object streams safer, compared to using lower level streams and doing manual casting.

Safe Termination of Threads

When users decide to exit the program, safe termination of server thread and client thread is important and should be handled with care. After exploring many different options, the latest version is designed to use socket timeout to close connections and terminate thread safely. It is done by setting a small timeout value at the beginning and catching the timeout exception at the end.

Error Handling

Most of the methods in SyncPeer are integrated with internal exception handling and they only return a Boolean value to callers telling whether the execution succeeds or runs into errors. Any error will lead to interruption of the current synchronization session. It prevents SyncPeer from sudden failure as well as reducing ambiguity in multilevel error handling.

Stream Flushing

In early versions, client and server could very often hang without any reason. After setting breakpoints and debugging the codes step by step, it was noticed that streams did not send out the message that had been written in. The problem was solved by calling the *flush()* method of streams after each write.

Bugs and Limitations

Non-General Communication Protocol

The current communication protocol serves SyncPeer application well but it is not general. It includes a special rule to handle file list transfers, which should be standardized into basic rules. It might cause confusion if the application is going to be further developed by a team.

Error Recovery

Currently SyncPeer application will terminate ongoing synchronization session on occurrence of any error. However, if time permits, an error recovery system should be implemented instead of using a forceful termination approach. It will improve user experience and make file transfers more efficient.

Blocking Acknowledgement

SyncPeer requires acknowledgements to be blocking. However, it greatly undermines efficiency of file transfers. A better acknowledgement system should be designed to allow unblocking acknowledgement and utilize the aforementioned error recovery system to recover from unacknowledged commands or file transfers.

Exception Handling System

The current exception handling system is focused on internal exception handling, ensuring that callers can tell if execution is successful from the returned Boolean value. However, it turns out that many functions are unnecessarily long and complicated due to integration of exception handling codes. A better choice might be implementing an appropriate multi-level exception handling system where functions selectively throw out exceptions and exceptions are caught at an appropriate level for recovery or termination.

User Guide

Compilation

1. Navigate to root directory of SyncPeer.
2. Compile the programming by typing :

   ```
   javac syncpeer/SyncStarter.java
   ```

   or doing

   ```
   javac `find . -name '*.java'
   ```

   for a clean build.

Usage

Launch

```
java syncpeer.SyncStarter <Sync Folder> [<IP Address>]
```

<Sync Folder>: Relative path to the folder to be synchronized

<IP Address>: (Optional) The server IP address.

When <IP Address> is present, SyncPeer starts in client mode and tries to connect to the server represented by the IP address. Otherwise SyncPeer starts in server mode and waits for user input or client connections.

Commands

1. Connect to a server with an IP address and start synchronization:

   ```
   CONNECT <IP Address>
   ```

   The connection will be automatically closed after synchronization and SyncStarter will switch back to server mode, listening to connections.

2. Exit the application:

   ```
   EXIT
   ```

   All connections will be closed and ongoing synchronization sessions will be terminated immediately. Please exit the application after synchronization is finished to prevent from loss of file data.

- Note that all commands are NOT case-sensitive.

Demonstration

Two test folders are included and they are located in the root directory of SyncPeer, named “SyncFolderA” and “SyncFolderB” respectively. SyncFolderA contains three files - “a.txt”, “b.txt” and “c.txt”. SyncFolderB contains two files – “x” and “y”.

Follow the instructions below to try a demo on your local machine.

Compile the codes:

```
javac syncpeer/SyncStarter.java
```

Launch Peer 1:

```
java syncpeer.SyncStarter SyncFolderA
```

Open up another terminal and launch Peer 2:

```
java syncpeer.SyncStarter SyncFolderB 127.0.0.1
```

You will notice that Peer 1 sends “a.txt”, “b.txt” and “c.txt” to Peer 2 and Peer 2 sends “x” and “y” to Peer 1. Now try making Peer 1 the client and Peer 2 the server. Switch to Peer 2 terminal. Now try exiting Peer 2:

```
EXIT
```

Peer 2 will exit safely. Now switch to Peer 1 terminal and try:

```
EXIT
```

Peer 1 will exit safely as well.

Compare the files in SyncFolderA and SyncFolderB. You will see they are exactly the same, meaning the file synchronization is successful.

Happy playing with SyncPeer!