Final Assignment

Book Shop

Prepared for Introduction to Network Programming class taught by Dr Zhanfang Zhao

Georgi Butev 3024421

MSc Computer Systems and Networking (CSN 1024 FT)

Faculty of ESBE, LSBU, London, UK, SE1 6LN.

butevg@lsbu.ac.uk

Date of Submission: 09 January 2012

Table of Contents

1. Introduction

This report details one possible solution to the outlined problem stated in the assignment sheet. The solution involves the development of a book shop server/client service. The server and client are separate Java applications (i.e. BookShop.class and Client.class). The inventory of books is stored in a text file (i.e. inventory.txt). There are three additional text files (i.e. temp.txt, basket.txt, and total.txt). All three are temporary and are automatically deleted when the server and client applications stop execution.

The "Introduction to Network Programming" lecture notes[*] were used as a general guideline. The "Writing the Server Side of a Socket" tutorial[**] from Oracle was used in order to construct the programming logic and flow charts for both server and client. Examples from (Horstmann, 2007, p.887-905)[***] were used to construct specific class methods.

2. Solution

The solution to the problem is separated into TCP server and TCP client. The solution to both server and client includes selected algorithms, classes' description, methods description, and flowcharts explaining the programming logic.

3. TCP Server

The TCP server class binds to the localhost, host name, and IP address via port number address. The port number address has to be in the range of 0 to 65535. It is recommended for custom network programs that the port number is between 1024 to 65535 or automatically generated and assigned by the operating system. The first 1024 port numbers are reserved for well-known applications such as HTTP (80), FTP (21), POP3 (110), and Telnet(23). This class is also used to create a socket, listen, and accept a connection from a client. Three Java API libraries were used to aid the solution:

1. java.io is used for input/output readers, buffers, exceptions.
2. java.net is used to create the server and client socket.
3. java.util scanner is used to scan strings using a delimiter.

3.1 Algorithms

The **readClientInput** method gets stream input from the client:

```
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(), true);
while((message = in.readLine()) != null){
    switch(message){
        case "I":
            sendInventory(out);
            break;
        default:
            if(message.startsWith("B")){
            String isbn = message.substring(2);
            buyBook(out, isbn);
    }
     break;
}
```


[*] http://eent3.lsbu.ac.uk/staff/zhaoza/inp/
[**] http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html
[***] Horstmann, C., 2007. Big Java. 3rd edition. Hoboken, USA: Wiley and Sons.

A decision is made using the client message as a switch. There are no other client commands (apart from Inventory and Buy) so the default case is used to buy a book. The B should be followed by the ISBN number. To get it, the first two characters would be subtracted (i.e. B and a white space).

The **sendInventory** method is invoked upon the client sending I. Before sending the inventory to the client the inventory file is checked (whether it exists, can be read, and written by the application). The inventory is read into the buffer and sent. The client would format the inventory with new lines. Also the server prints the file path and size in bytes part of log functionality.

```
File file = new File("inventory.txt");
FileReader fr = new FileReader(file);
BufferedReader br = new BufferedReader(fr);

if(file.exists() && file.canRead() && file.canWrite()){
    System.out.println(file.getAbsolutePath() + " " + file.length() + " bytes");
     while((temp = br.readLine()) != null){
         out.println(temp);
       }
}
```

The **buyBook** method searches for the ISBN number. If the number is present in the inventory.txt the following will be extracted and stored from the scanner – ISBN, book title, author name, price, and quantity. The quantity will be parsed as integer, decreased by one, and converted into string ready for writing to the temp.txt file.

The temp.txt file is used to duplicate the inventory.txt file but with a decreased value of quantity. Initially my intentions were to find and replace only the quantity value. Unfortunately this proved to be rather problematic. This is why the final solution uses three pass temp.txt file writing procedure. Another problem was that the ISBN may be in beginning, middle or end of the file. That is why the first write pass stores all tokens (if any) from the scanner into the file. Also during the first pass the client gets a response from the server. The second pass stores the ISBN, book, author, price, and quantity into the file. Finally whatever is left (if any) in the scanner is stored into the file.

1st Pass

```
File tempFile1 = new File("temp.txt");
tempFile1.createNewFile();
FileWriter fw1 = new FileWriter(tempFile1, true);
BufferedWriter firstPass = new BufferedWriter(fw1);
    while(true){
         temp[i] = scanner.next();
         firstPass.write(temp[i] + "\t");
         if (temp[i].equals(isbn)){
              found = temp[i];
              title = scanner.next();
              author = scanner.next();
              price = scanner.next();
              quantity = scanner.next();
              q = Integer.parseInt(quantity, 10) - 1;
              qForWrite = Integer.toString(q);
              out.println(found + "\t" + title + "\t" + author + "\t" + price + "\t" + q + "\t");
              break;
         }
         i++;
    }
```

2nd Pass

File tempFile2 = new File("temp.txt");
FileWriter fw2= new FileWriter(tempFile2, true);
BufferedWriter secondPass = new BufferedWriter(fw2);
 	
secondPass.write(title + "\t" + author + "\t" + price + "\t" + qForWrite + "\t");
secondPass.flush();
fw2.close();

3rd Pass

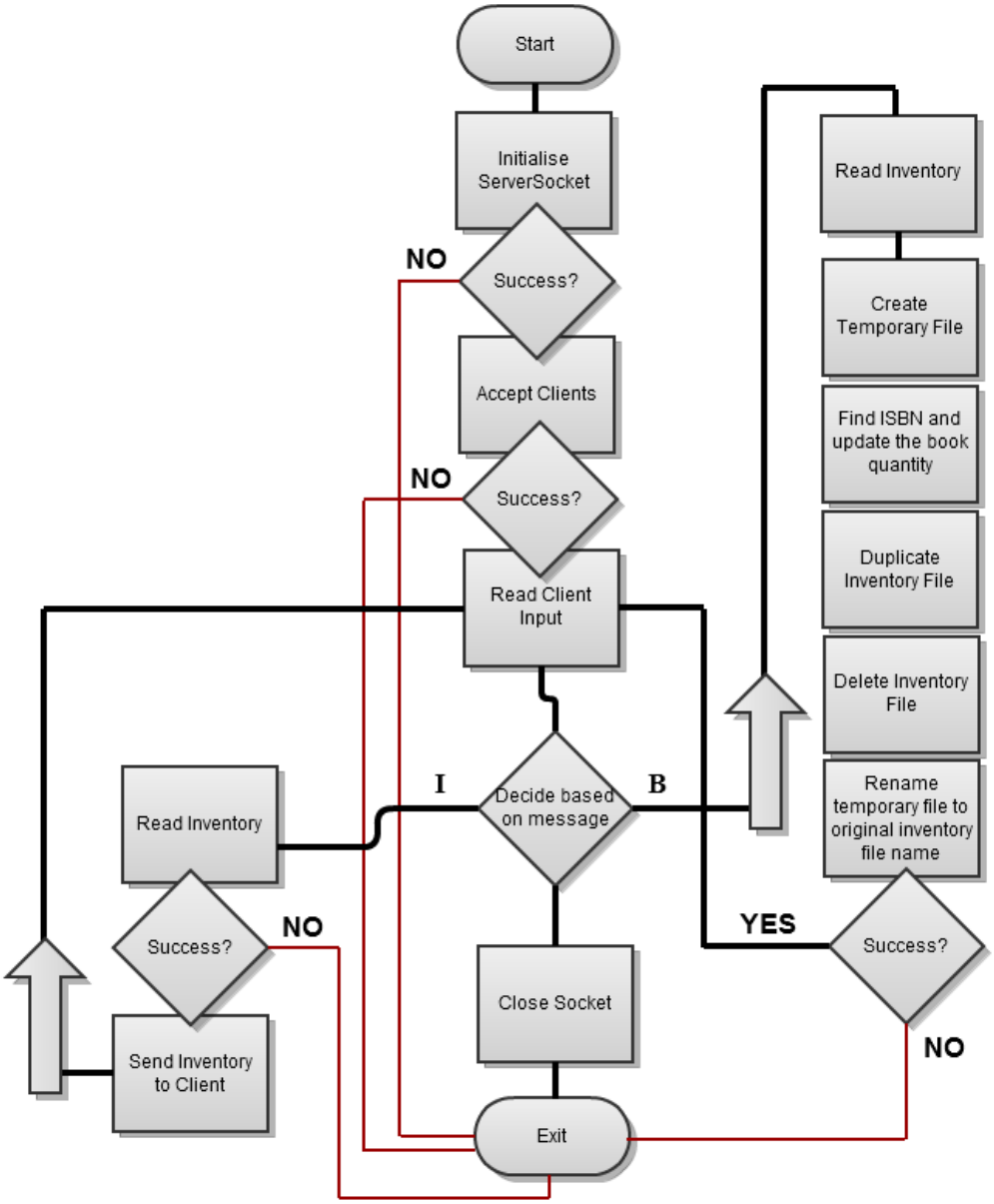
File tempFile3 = new File("temp.txt");
FileWriter fw3 = new FileWriter(tempFile3, true);
BufferedWriter thirdPass = new BufferedWriter(fw3);
 	
while(scanner.hasNext()){
thirdPass.write(scanner.next() + "\t");
}
thirdPass.flush();
fw3.close();
fr.close();
 	
file.delete();
tempFile1.renameTo(file);

After the third pass the temp file has duplicated the original inventory but with decreased quantity. The inventory file should be deleted and replaced. The temporary file should be renamed to inventory.txt because the application requires this file to read the inventory.

3.2 Class

BookShop
+ main(args: String[]): void + startServer(port: short): void + readClientInput(socket: Socket): void + sendInventory(out: PrintWriter): void + buyBook(out: PrintWriter, isbn: String): void

3.3 Flowchart



4. TCP Client

The client class uses a socket to connect to the TCP server application. The server is identified by localhost and port number address. The localhost could be manually changed to local area network host name or IP address. Such socket communication between client and server works best in a local area network where there are no routers and little delay.

If the server is located at a remote destination its IP address should be public. If the server is behind NAT (Network Address Translation) network device inside a private network with private IP addresses (e.g. 10.0.0.1, 172.16.0.1, 192.168.0.1) the client will not be able to connect to the server. This problem could be overcome using third server with a public IP address. In this case both the client and server should connect to the third server. Afterwards all incoming client data streams would be transmitted from the third server to the destination server (with a private IP address).

For better performance the client and server could use UDP (User Datagram Protocol). This protocol has very little data overhead and it is suitable for streaming media such as audio and video. That is because if some parts of a song or a live TV news cast are skipped the client would not notice any significant change. In other words the UDP does not guarantee packet delivery. Such guarantee is not essential for media broadcast.

However TCP is used for these two applications because we need safe and guaranteed delivery of data. If this was a real life application, buying a book should be precise. For example if some packets are lost the client may end-up paying a lot more for a single book. Furthermore there are no strict performance requirements for such a book shop service.

The objective of this class is to allow the client to view the inventory from the server and buy selected (by ISBN) available books. The following commands were implemented:

- **I** – requests the entire inventory.
- **B** – reminds the user that he/she should use B followed by the ISBN number.
- **B xxxxxxxxxx** – sends a request to buy the selected book. The sequence of x is a standard ten digit ISBN number.
- **basket** - displays all recently purchased books during this session.
- **quit** - the client and server stop.

Three Java API libraries were used to aid the solution:

1. java.io is used for input/output readers, buffers, exceptions.
2. java.net is used to create the server and client socket.
3. java.util scanner is used to scan strings using a delimiter.

4.1 Algorithms

Establish a connection to the server, read input data stream, and create object for output data stream.

```
final short PORT = 5555;
final String HOST = "localhost";
Socket socket = null;
socket = new Socket(HOST, PORT);

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(), true);
```

Make decision for (inventory, buying a book, shopping trolley, quit) based on user command:

```
while(true){
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String message = br.readLine();
    switch(message){
        case "I": requestInventory(out, message, in);
            break;
        case "B":
            System.out.println("Please, enter B followed by the ISBN number.");
            break;
        case "basket":
            System.out.println("This is the content of your shopping trolley: ");
            viewBasket();
            break;
```

```
        case "quit": quit(socket);
            break;
        default:
            if(message.startsWith("B")){
            buyBook(out, message, in);
    }
    break;
}
```

The **requestInventory** method sends an **I** command to the server. The server should respond with the entire inventory items. The inventory items use tab as a delimiter. The client would receive one big tabbed string of data with no new lines. To deal with this problem we need to format the string appropriately. That is why the programme iterates over all string tokes. After every fifth element the programme would insert a new line. Furthermore after every token the programme would insert a pipe symbol.

```
out.println(message);
String temp = in.readLine();
Scanner scanner = new Scanner(temp).useDelimiter("\t");
System.out.println("ISBN\t" + "Book Name\t" + "Author Name\t" + "Price\t" + "Quantity in Stock");
while(scanner.hasNext()){
    for(int i = 0; i<=4; i++){
        System.out.print(scanner.next() + " | ");
    }
    System.out.println();
}
```

The **buyBook** method stores the recently purchased book into a string. Afterwards two methods are called to deal with that string. The **storeInBasket** method creates a temporary basket.txt file and stores the previously mentioned string for future reference. The **buyBook** method is called upon “B xxxxxxxxxx” command from the client's keyboard input.

```
File file = new File("basket.txt");
FileWriter fw = new FileWriter(file, true);
BufferedWriter bw = new BufferedWriter(fw);

bw.write(book + "\n");
bw.flush(); fw.close();
```

The **storeInAccounts** method creates a temporary total.txt file, stores, and processes the previously mentioned string. This method simply extracts the price from the basket.txt file and parses the string literals into double numbers (the total price of bought books requires floating point calculation).

```
File file = new File("total.txt");
FileWriter fw = new FileWriter(file, true);
BufferedWriter bw = new BufferedWriter(fw);
Scanner scanner = new Scanner(book).useDelimiter("\t");
isbn = scanner.next();
title = scanner.next();
author = scanner.next();
temp = scanner.next();
price = Double.parseDouble(temp);
bw.write(price + "\n");
bw.flush(); fw.close();
```

The **viewBasket** method reads the temporary basket.txt file and displays the result to the client. The result should be a list of all the purchased books by the client/server for this session. It also displays the total price (two digits after the decimal point precision) of bought books, calling the **calculatePrice** method. The **viewBasket** method is called upon “basket” command from the client's keyboard input.

```
File file = new File("basket.txt");
FileReader fr = new FileReader(file);
BufferedReader br = new BufferedReader(fr);
while((temp = br.readLine()) != null){
    System.out.println(temp);
}
br.close(); fr.close();
Double total = calculatePrice();
System.out.printf("Total price for the whole transaction is: %.2f \n", total);
```

The **calculatePrice** method reads the temporary total.txt file which contains all recently purchased books' prices. Also it adds all of them to calculate the total amount. Afterwards it returns the sum to the **viewBasket** method.

```
File file = new File("total.txt");
FileReader fr = new FileReader(file);
BufferedReader br = new BufferedReader(fr);
Scanner scanner = new Scanner(br).useDelimiter("\n");
while((temp = br.readLine()) != null){
    sum += Double.parseDouble(temp);
}
br.close(); fr.close();

return sum;
```

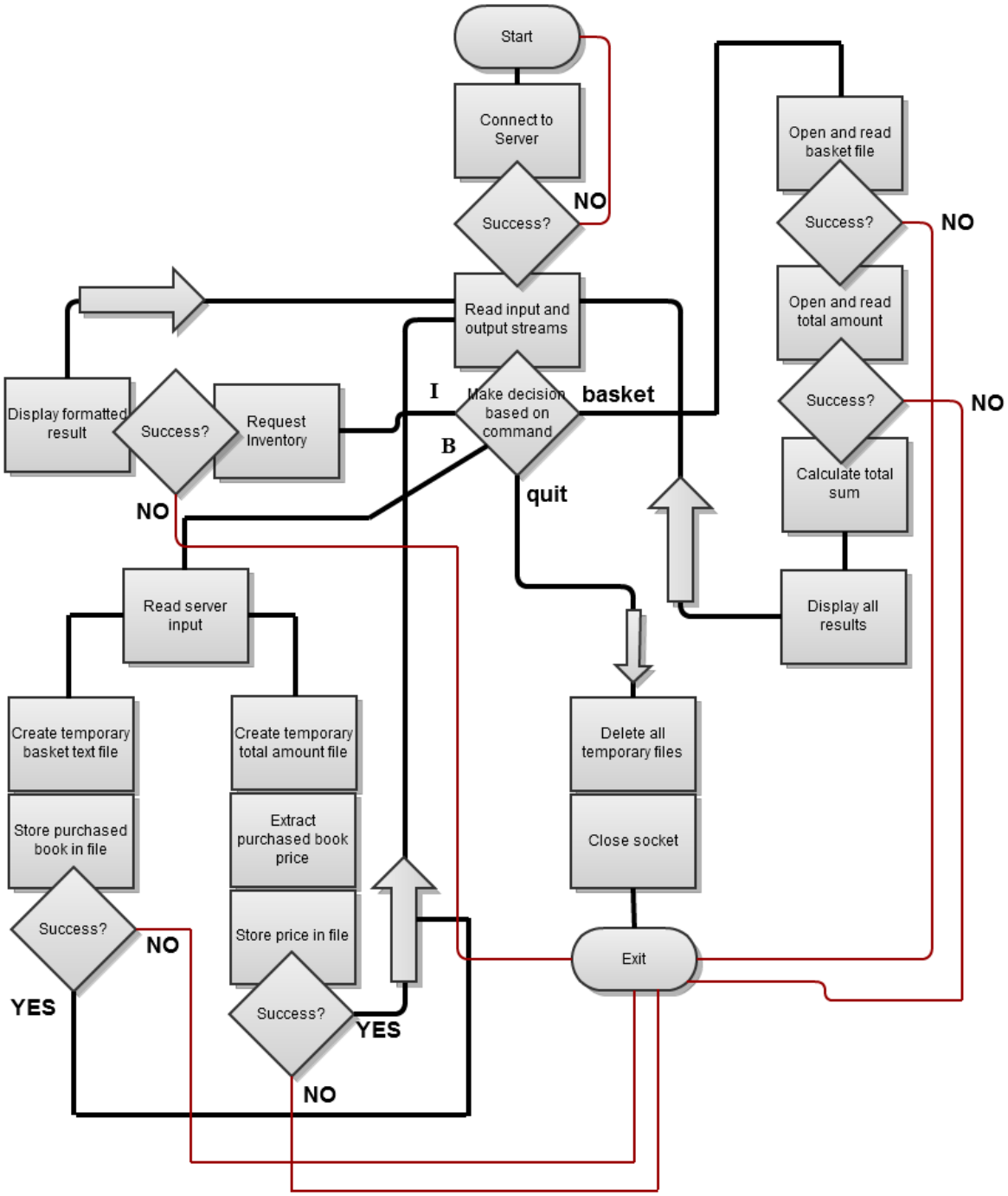
The **quit** method is called when a “quit” command is received from the keyboard input. This method deletes the temporary files basket.txt and total.txt. It also closes the socket connection.

```
File file1 = new File("basket.txt");
file1.delete();
File file2 = new File("total.txt");
file2.delete();
socket.close();
System.exit(0);
```

4.2 Class

Client
+ main(args: String[]): void + requestInventory(out: PrintWriter, message: String, in: BufferedReader): void + buyBook(out: PrintWriter, message: String, in: BufferedReader): void + storeInBasket(book: String): void + storeInAccounts(book: String): void + viewBasket(): void + calculatePrice(): double + quit(): void

4.3 Flowchart



5. Conclusion

The client is able to connect to the server using predefined port number and host name via reliable transport control protocol. All required functionality was implemented for both the server and client. The biggest challenge during development time was controlling while loops which locked a process and no other methods or statements could be reached. Alternatively a while loop would iterate properly but break abruptly. Also problematic was fine tuning the sequence of input, output, response, reply between client and server. Extracting required data from buffers and scanners did not go without problems. Opportunities for this project are the implementation of multi-threaded response, backlog queue, and graphical user interface. A possible software bug may be unexpected server malfunction during the three pass writing – this would garble the data in the original inventory file.