Inter Process Communication

Renee Paxson

CEG 4350

17 July 2024

Project Overview

This project covered the various methods of inter process communications. The goal was to use different IPC methods to transfer 100 pieces of random data between two processes. I used FIFO and Sockets to achieve this goal.

Inter-Process Communication: Named Pipe / FIFO

System: OSx

Language: C/C++

Run Instructions

To run, the user must build an executable from fifo1.cpp and fifo2.cpp in their local systems and run both at the same time.

Summary

The goal of this IPC method is to write 100 pieces of data to a named pipe, called a "fifo" in OSx, and read the data on the other side of the pipe in a separate process.

In my implementation, fifo1.cpp writes 100 random integers to the fifo `myfifo`, and fifo2.cpp reads the integers from `myfifo`. The input from `fifo1.cpp` prints out to fifo-intput.txt and the data read from the pipe in `fifo2.cpp` prints out to fifo-output.txt. These files are exactly identical, proving that the communication was successful. Specific documentation on the individual steps taken to achieve that goal are written in the comments of the code seen below.

`fifo1.cpp` code and comments:

```
13  int main() {
14      // if the input file already exists, delete it
15      std::remove("./fifo-input.txt");
16
17      // reset the default umask bits so that
18      // it doesn't mess with the permissions on the fifo
19      umask(0000);
20
21      // int to refer to the number in the file descriptor table
22      // most people just call this variable 'fd'
23      int fileDescriptor = -1;
24
```

```cpp
    // FIFOs are stored as files
    // This is the file path to where we wish to store our FIFO
    // and how we wish to name it
    char * myfifo = "./myfifo";

    // make fifo with RW permissions for all user, group, and others
    mkfifo(myfifo, 0666);

    // open fifo for write only
    // program will hold here until the read end of the pipe is open in fifo2
    while (fileDescriptor == -1) fileDescriptor = open(myfifo, O_WRONLY, 0666);

    // buffer to hold the data to send through the pipe
    int buff[100];
    // seed rand to make it rand
    srand(time(NULL));

    string filepath = "./fifo-input.txt";
    ofstream file;
    file.open(filepath);

    if (file.is_open()){
        for (int i = 0; i < 100; i++){
            // fill buffer with random numbers
            buff[i] = rand();
            file << buff[i] << std::endl;
        }

        // write buff to the fifo
        write(fileDescriptor, buff, sizeof(buff));

        // close fifo
        close(fileDescriptor);
    }

    // close file
    file.close();

    return 0;
}
```

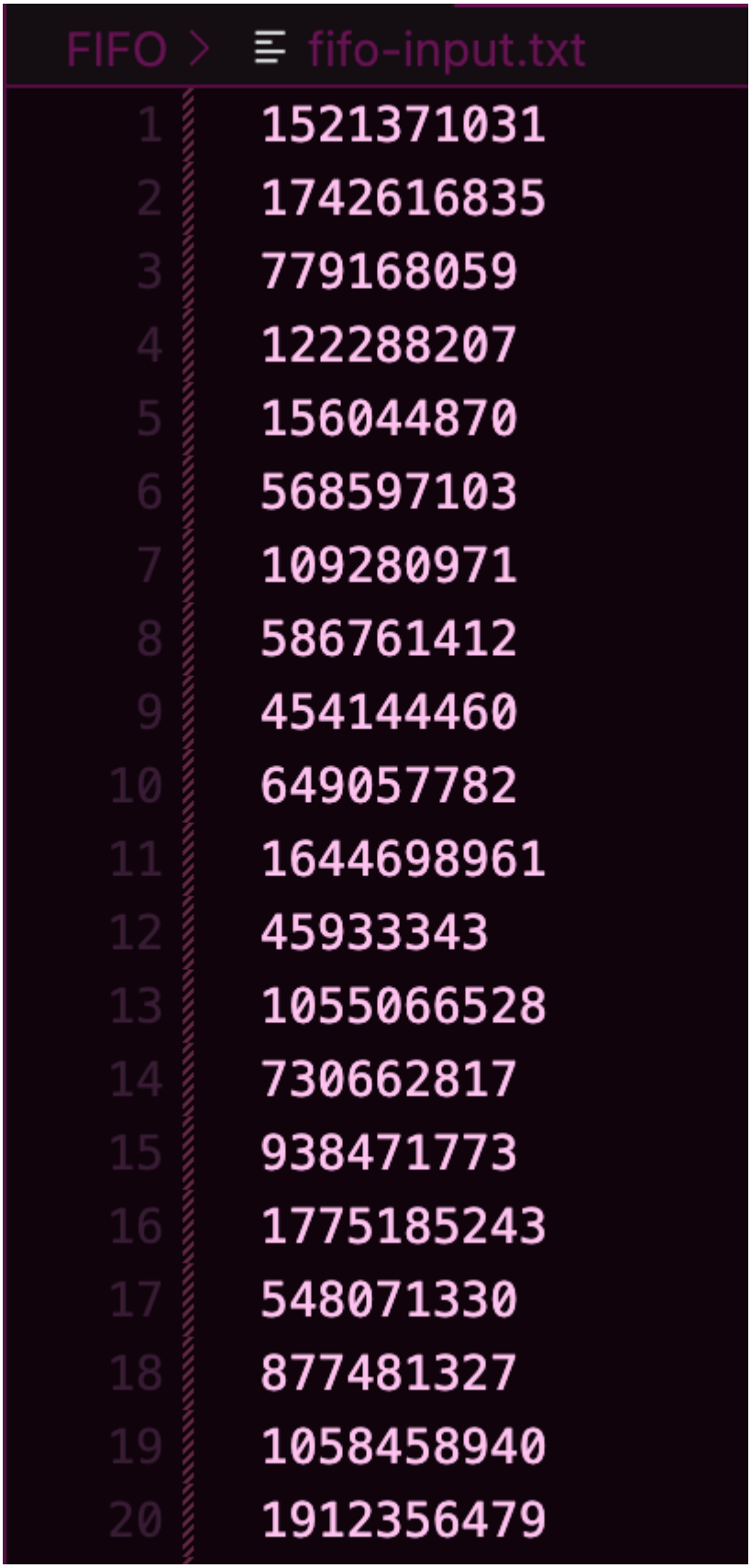
`fifo2.cpp` code and comments:

```
FIFO > fifo2.cpp > main()
int main() {
    // if the input file already exists, delete it
    std::remove("./fifo-output.txt");

    // reset the default umask bits so that
    // it doesn't mess with the permissions on the fifo
    umask(0000);
```

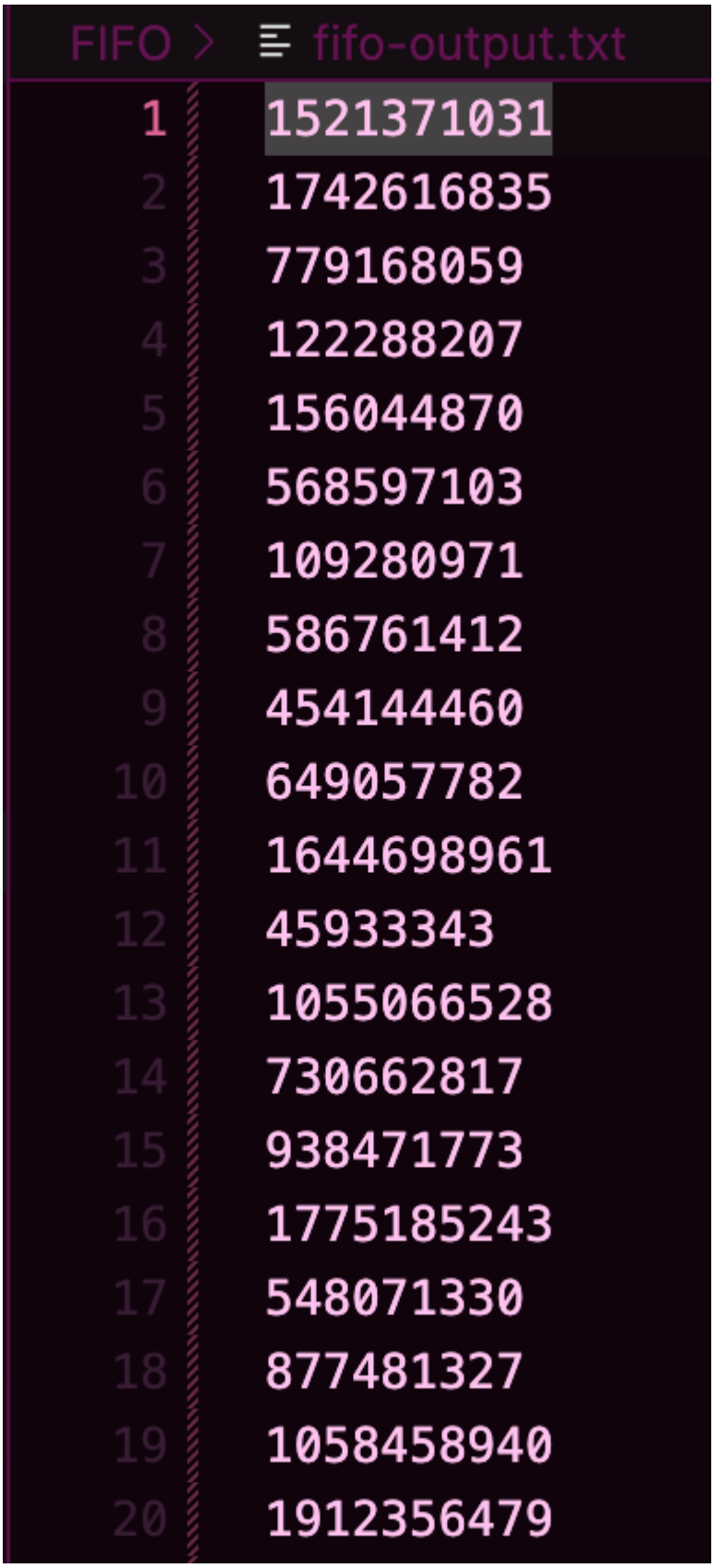
```cpp
    // int to refer to the number in the file descriptor table
    // most people just call this variable 'fd'
    int fileDescriptor = -1;

    // FIFOs are stored as files
    // This is the file path to where we wish to store our FIFO
    // and how we wish to name it
    char * myfifo = "./myfifo";

    // make fifo with RW permissions for all
    mkfifo(myfifo, 0666);

    // open fifo for read only
    // program will hold here until the write end of the pipe is open in fifo1
    while (fileDescriptor == -1) fileDescriptor = open(myfifo, O_RDONLY, 0666);

    // buffer to hold the data sent through the pipe
    // size 1 since we read one entry at a time
    int buff[1];

    string filepath = "./fifo-output.txt";
    ofstream file;
    file.open(filepath);

    // `read` will return -1 if there's an error
    // and 0 if the buffer is empty
    // and holds indefinitely if the buffer hasn't been written to yet
    while (read(fileDescriptor, buff, sizeof(buff)) > 0) {
        if (file.is_open()){
            file << buff[0] << std::endl;
        }
    }

    // close fifo
    close(fileDescriptor);

    // close file
    file.close();

    return 0;
}
```

The data written to the fifo will differ with each run, but the first 20 lines of data collected from my last run in each file are shown below

`fifo-input.txt`:



`fifo-output.txt`:



System Calls Used

- `remove()`: used to remove the input/output files at the beginning of the program if they already exist
- `mkfifo()`: the method used to create a fifo
- `open(fifo, O_WRONLY | RDONLY, permission_set)`: used to open the fifo for write only or read only
- `file.open(filepath)`: used to open the text file for writing

- `write(fileDescriptor, buff, sizeof(buff))`: used to write to the fifo
- `close(fileDescriptor)`: used to close the pipe
- `file.close()`: used to close the text file

Challenges

One of the biggest challenges with writing the fifo IPC was figuring out how to set the proper file permissions for the fifo. The intention was to set the permissions allow read and write to the user, group, and others, but running `mkfifo(myfifo, 0666)` resulted in permissions that were shifted in some way, like read and *execute* for all instead of write. After some sleuthing, I found out that the default umask bits were modifying my permission set. To solve this, I reset the umask to 0000 at the beginning of both `fifo1.cpp` and `fifo2.cpp`

Inter-Process Communication: Sockets

System: OSx

Language: Java

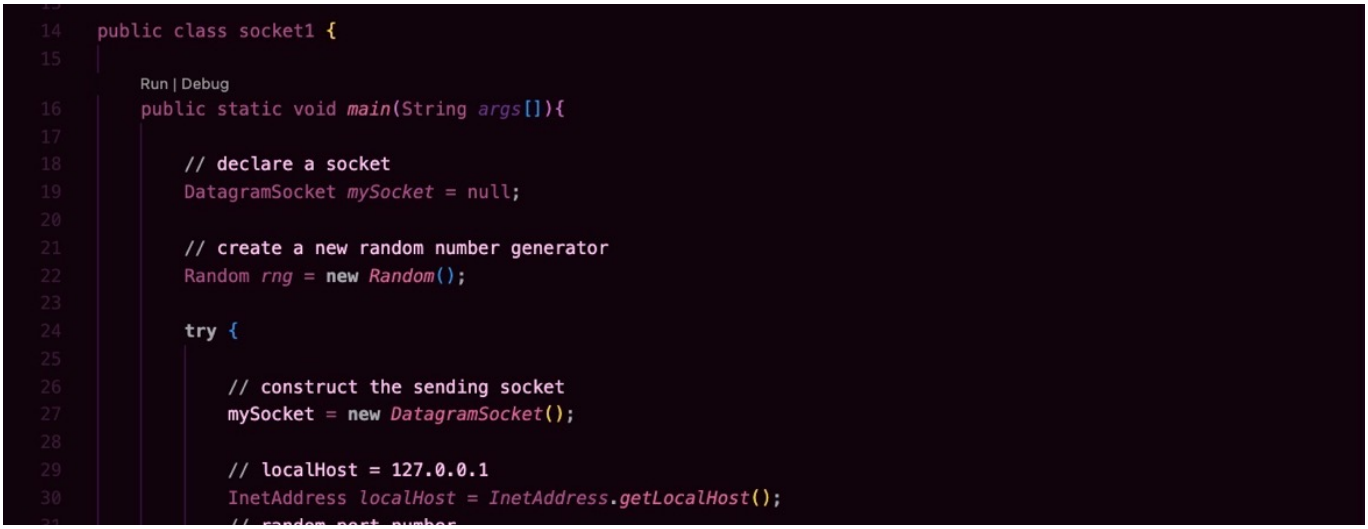
Run Instructions

To run, the user must build an executable from socket1.java and socket2.java in their local system and run them at the same time by running `socket2` first then `socket1`.

Summary

Here we are sending 100 randomly generated integers as byte arrays from `socket1.java` to `socket2.java` on our local address (LocalHost = 127.0.0.1) and the specified port to which the receiving process `socket2.java` is bound - I chose port 1234, but it doesn't matter so long as the port number is outside of the reserved range (>1023) and it is not being used by any other receiving processes. The messages are sent one at a time through Java classes `DatagramPacket` and `DatagramSocket`. The input from `socket1.java` writes to the file socketInputData.txt, and the data read from `socket2.java` writes to the file socketOutputData.txt. The contents of both files are exactly identical, proving the success of the communication. Specific documentation on the individual steps taken to achieve that goal are written in the comments of the code seen below.

`socket1.java` code and comments:



```java
            // random port number
            int serverPort = 1234;
            // variable to store the random number on
            int data;
            // random number generator seed to make the output truly random
            //int seed = (int) System.currentTimeMillis();

            // loop until 100 messages have been sent
            for (int i = 0; i < 100; i++){
                // generate the random number
                data = rng.nextInt();
                // convert integer to byte array
                // all ints have a maximum length of 4 bytes, so our byte array capacity is 4
                byte [] message = ByteBuffer.allocate(capacity:4).putInt(data).array();

                // create datagramPacket
                DatagramPacket datagramMessage = new DatagramPacket(message, message.length, localHost, serverPort);
                mySocket.send(datagramMessage);


                // I want to save each generated value as an int to a text file
                try {

                    // define filepath to save the outgoing data on
                    String pathname = "./socketInputData.txt";
                    // create a File instance using the path name
                    File file = new File(pathname);

                    // create the file writer with parameter "true" to allow it to append
                    FileWriter filewriter = new FileWriter(pathname, append:true);

                    // if file doesn't exist, create it
                    if (file.createNewFile()){}

                    // if file exists but this is the start of the loop (i=0)
                    else if (i==0){
                        // delete the file
                        Files.deleteIfExists(file.toPath());
                        // and recreate it
                        file.createNewFile();
                        // redefine filewriter
                        filewriter = new FileWriter(pathname, append:true);
                    }

                    filewriter.write(data + "\n");
                    filewriter.close();

                } catch (IOException e){
                    System.out.println(x:"an error occurred");
                    e.printStackTrace();
                }
            }
        } catch(SocketException e){System.out.println("Socket: "+ e.getMessage());
        } catch (IOException e){System.out.println("IO: " + e.getMessage());
        } finally { if (mySocket != null) mySocket.close();}
    }
}
```

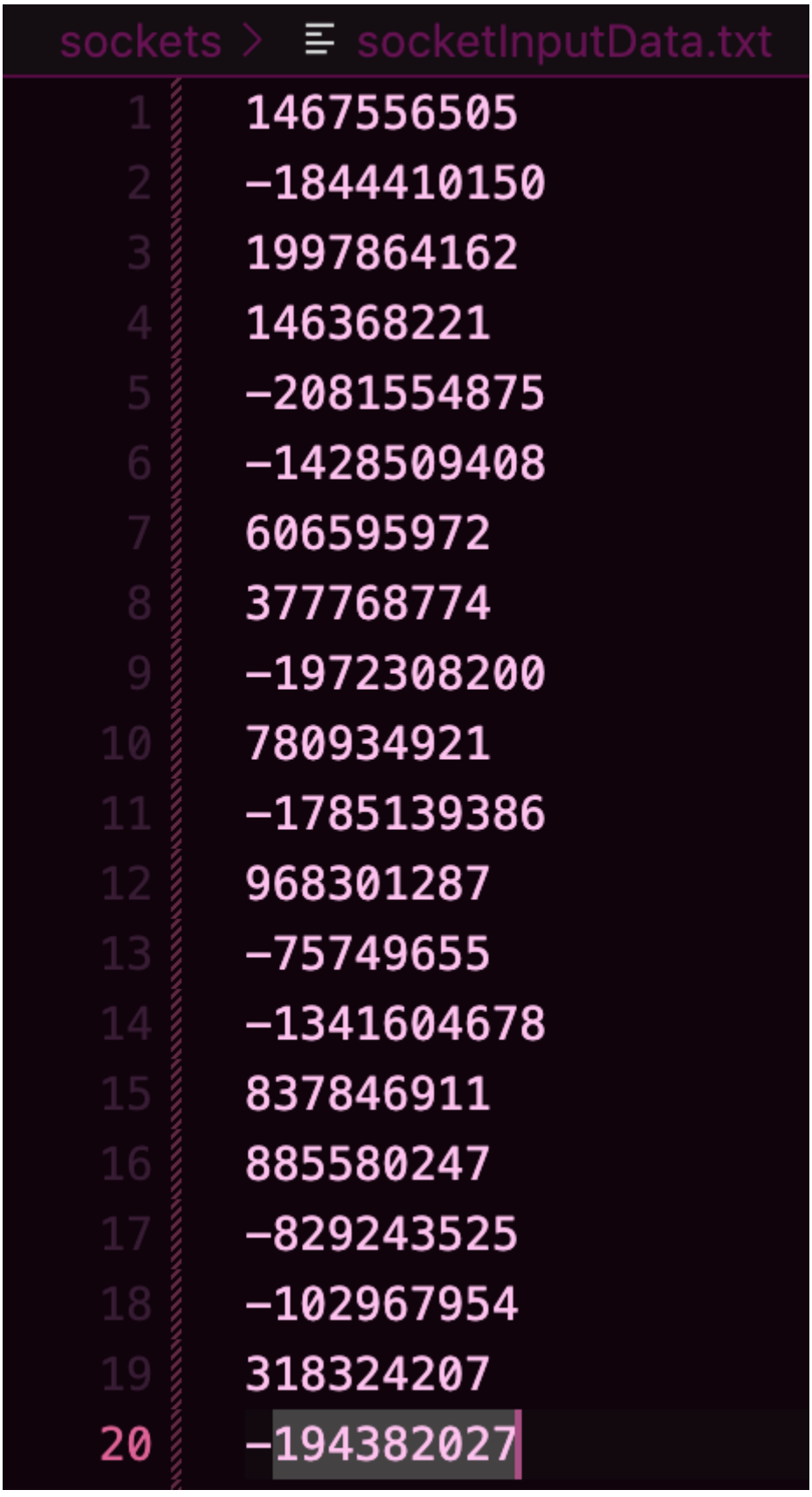
`socket2.java` code and comments:

```java
public class socket2 {
    Run | Debug
    public static void main (String args[]){

        // declare socket
        DatagramSocket mySocket = null;

        // define the filepath to save the received data on
        String pathname = "./socketOutputData.txt";
        // create a File instance using the path name
        File file = new File(pathname);
```

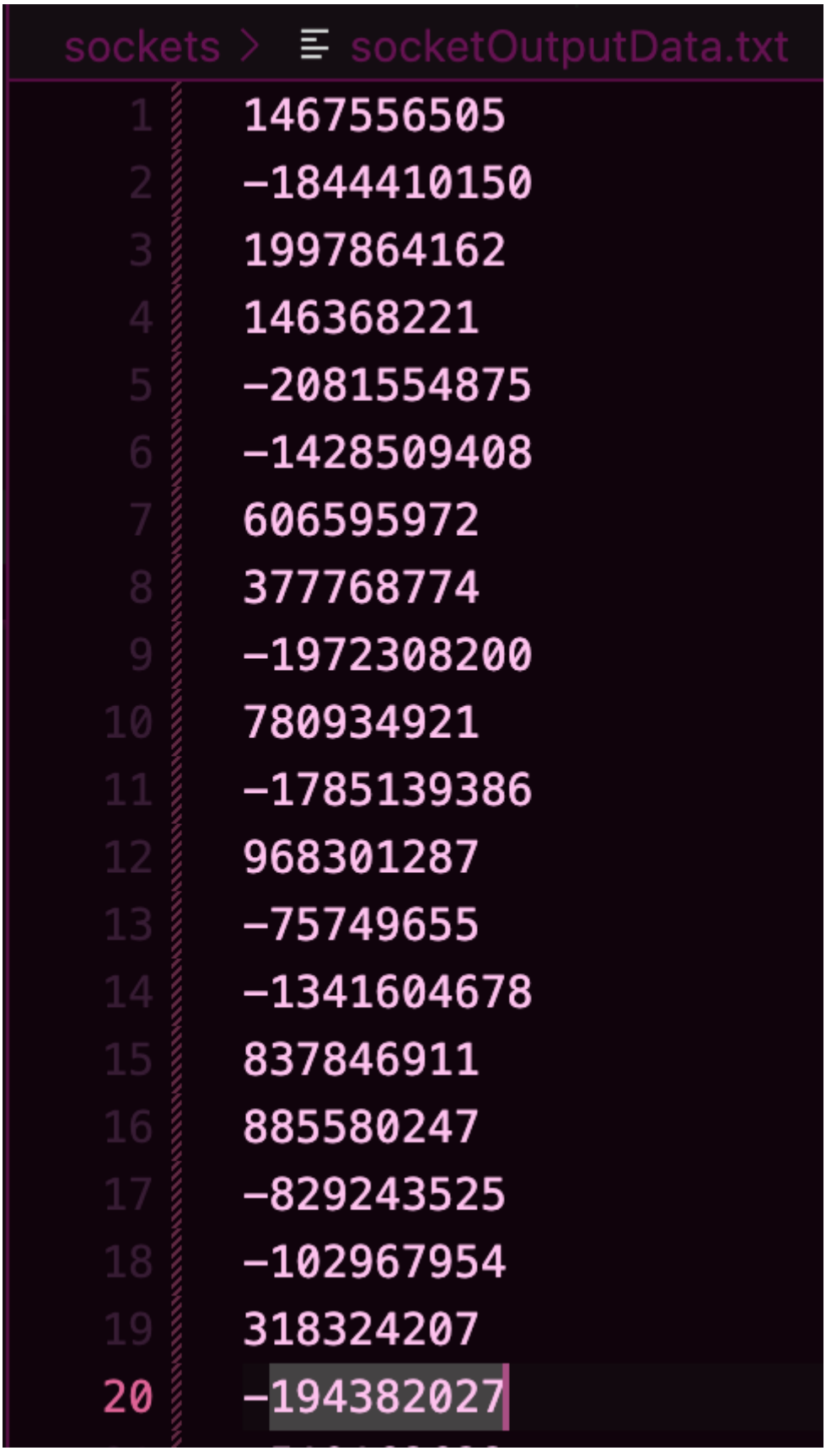
```
        File file = new File(pathname);

        try {
            // before we enter the loop, determine if the received data
            // file exists already and delete if it does
            Files.deleteIfExists(file.toPath());

            // construct the receiving socket and bind it to port 1234
            mySocket = new DatagramSocket(port:1234);

            // large byte array to receive a potentially large message
            byte[] message = new byte[1000];

            int i = 0;

            while(true){

                // create empty datagramPacket to receive the message into
                DatagramPacket request = new DatagramPacket(message, message.length);

                mySocket.receive(request);

                // convert message from byte array to int
                ByteBuffer wrapped = ByteBuffer.wrap(message);
                int data = wrapped.getInt();

                // the "write to file" section
                try {
                    //create filewriter with parameter "true" to allow append
                    FileWriter filewriter = new FileWriter(pathname, append:true);

                    // if file doesn't exist, create it
                    if (file.createNewFile()){}

                    filewriter.write(data + "\n");
                    filewriter.close();

                } catch (IOException e){
                    System.out.println(x:"an error occurred");
                    e.printStackTrace();
                }
                i++;
                if (i > 99){ break;}
            }
        } catch (SocketException e) { System.out.println("Socket: " + e.getMessage());
        } catch (IOException e){ System.out.println("IO: " + e.getMessage());
        } finally { if (mySocket != null) mySocket.close();}
    }
}
```

The data communicated will differ with each run, but the first 20 lines of data collected from my last run in each file are shown below

`socketInputData.txt`:



`socketOutputData.txt`:



System Calls Used

- `mySocket.send(datagramMessage)`: used to send the Datagram Message to the other process
- `filewriter.write(data + "\n")`: used to write the input / output data to a text file
- `filewriter.close()`: used to close the text file

The most integral methods to this implementation of socket communication are associated with the Java classes `DatagramPacket` and `DatagramSocket`.

- `DatagramPacket`: this method constructs a packet that holds the message being communicated and the length of the message. In the case of a sending process, the packet also contains the internet address of the destination process and the port number the destination is bound to
- `DatagramSocket`: this method constructs a socket to be used to transfer Datagram Packets

Challenges

For me, the biggest challenge of writing this Java-based socket IPC method was remembering how to use Java. I haven't used Java in a while - recently I've been leaning more towards C++ because I used it more in my classes. I spent the most time in this part figuring out how to handle the input and output files, such as the Java commands for deleting a file if it already exists and using Java's FileWriter method. An additional issue associated with this was learning how to convert the message from a byte array to integers and vice versa, which was achieved by a helpful Java class called `ByteBuffer`. In the end, Java was a very fun language to write in, and I'm looking forward to using it again in the future.