Computer Networks

Lab #3 - Networking Programming Using TCP Sockets

Summary

- Client/Server Model with TCP
- TCP Sockets
- Java Example
- Exercise: File Transfer over TCP

Client/Server Model ¶

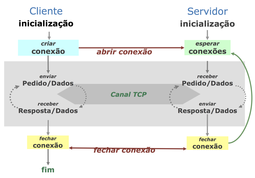
Two autonomous components

- Server - first to run and usually always running
- Client - usually started by the user to request a service

In previous labs we have seen how to build a distributed application where components (processes) coordinate using message exchange by the way of exchanging UDP Datagrams. We used the client / server pattern to structure our network application. We build several instances of a file copy program and a project based on UDP.

In this lab we will see how to use TCP Sockets to more easily build the same application.

Client/Server Model with TCP Channels



TCP Logical Channels or Connections (or Streams)

- A TCP connection is a logical two-way reliable channel among two processess
- The connection is open by the client, directed towards the server IP address and port,
- The server IP address and port identifies the other extreme of the connection
- It supports two independent, reliable and ordered flow of bytes — one in each direction
- It can be closed at any moment by any of the two communicating processess
- Before any communication can take place, both sides must agree that they want to establish the communicating TCP channel among them

TCP Sockets

- A TCP connection is established among two TCP Sockets, one in each extreme of the channel
- A client TCP Socket *"opens"* a connection to the server side TCP Socket - the first *opens* the connection, the second one *accepts* it
- A server creates a TCP Socket to accept incoming connections; this socket has a server port and the server IP address
- A client opens or creates the connection by requesting the creation of a local TCP Socket connected to the server TCP Socket

Note: UDP and TCP port numbers are similar concepts but their ranges are independent

Example (ECHO Server and Client)

The client creates a TCP Socket by connecting it to the server TCP Socket; the server Socket is identified by the server address and the socket port. Then, the client reads lines from its consoleand sends them to the server. The server reads the bytes sent by the client and echoes them back to the client.

Java Server Code

The code of the server is very simple. It just creates a Socket to accept incoming connections in the previously agreed port. Then it accepts client request to establish a connection.

In [ ]:
```
import java.io.*;
import java.net.*;

public class EchoServer {

    public static final int PORT = 8000 ;

    public static void main(String args[] ) throws Exception {

        // creates a server socket to wait for connections
        try(ServerSocket serverSocket = new ServerSocket( PORT )) {
            for(;;) {
                // waits for a new connection from a client
                try(Socket clientSocket = serverSocket.accept()) {
                    // handle the connection...
                    new ConnectionHandler().handle( clientSocket );
                } catch( IOException x ) {
                    x.printStackTrace();
                }
            }
        }
    }
}
```

When the connection is established, the handler simply continously reads bytes and writes them back to the other side while the connection is not closed.

In [ ]:
```
import java.io.*;
import java.net.*;

public class ConnectionHandler {
    private static final int TMP_BUF_SIZE = 16;

    public void handle( Socket cs ) throws IOException {

        InputStream is = cs.getInputStream();

        for(;;) {
            // implements the data ECHO, by reading and writing
            // while the connection is not closed

            int n ;
            byte[] buf = new byte[ TMP_BUF_SIZE] ;
            while( (n = is.read(buf)) > 0 )
                os.write( buf, 0, n );
        }
    }
}
```

Note that after the connection is established, it can be seen as a read / write stream/pipe.

Java Client

The client starts by processing the parameters and opening a connection to the server.

When the connection is open, it starts using it as a read / write stream/pipe.

In [ ]:
```
import java.io.*;
import java.net.*;
import java.util.*;

public class EchoClient {
    private static final int PORT = 8000;
    public static void main(String[] args ) throws Exception {
        if( args.length != 1 ) {
            System.out.println("usage: java EchoClient server") ;
            System.exit(0);
        }
        String server = args[0] ;
        // Creating a connection to the server
        try(Socket socket = new Socket( server, PORT )) {
            OutputStream os = socket.getOutputStream();
            InputStream is = socket.getInputStream();

            //...
        }
    }
}
```

Once the connection is established, the client prepares a `Scanner` to read bytes from the console (`System.in`).

Enters a loop where it reads a line, sends it to the server, gets the echo and prints it to the console, until it receives `!end`.

In [ ]:
```
String echoRequest;
    Scanner in = new Scanner( System.in ) ;
    do {
        echoRequest = in.nextLine() + "\n";
        os.write( echoRequest.getBytes() ) ;
        String echoReply = new Scanner( is ).nextLine();
        System.out.printf("Server replied: \"%s\"\n", echoReply ) ;
    } while( ! echoRequest.equals("!end\n") ) ;
```

Echo Java Files

You can download the client and the server as complete examples. The server code is available here and the client here

Recipes

Class ServerSocket

In [ ]:
```
try( ServerSocket ss = new ServerSocket( PORT ) ) {
    ...
        cs = ss.accept();
    ...
}
```

Class Socket

In [ ]:
```
try( Socket ss = new ServerSocket( server, PORT ) ) {
    ...
    InputStream is = ss.getInputStream();
    OutputStream os = ss.getOutputStream();
    ...
}
```

Sending and receiving (multiple) bytes

In [ ]:
```
int n;
byte buf = new byte[TMP_BUF_SIZE];
while( (n = is.read( buf )) > 0 )
    os.write( buf, 0, n)
```

Reading a single byte at each time (slow)

In [ ]:
```
InputStream is = cs.getInputStream();
int b = is.read();
```

WARNING: Anti-Pattern

InputStream.available() works with FileInputStream, but ***does not work*** with streams that are backed by Sockets.

In [ ]:
```
Socket cs = new Socket( server, port );
InputStream is = cs.getInputStream();
while( is.available() ) {

};
```

Exercise

TCP File Transfer

Given this simple TCP server:

1. Program your own client to send a file to this server. After the connection is established, a byte array terminated with `byte \0`, containing the name of the file, is sent to the server, followed by the file contents.
2. Next, transform your iterative server, into a concurrent one. Use threads to make it capable of receiving several files in parallel.

   - A docker image with the server ready to be used can be launched using:

     ```
     docker run -t -p 8000:8002 smduarte/rc18-tcpfileserver
     ```

     The server listens at port 8000...

Java Tips

Working with threads...

Threads + Lambda Expression

In [ ]:
```
new Thread( () -> {

    // place here code to execute in new thread...

}).start();
```

Threads + Helper class

Helper class implements interface Runnable

Main thread calls:

In [ ]:
```
new Thread( new HelperClass( args )).start();
```

Child thread executes in run(), receives args in constructor...

In [ ]:
```
class HelperClass implements Runnable {
    HelperClass( ... ) {
        // Constructor receives any args the helper class needs to run...
    }
    public void run() {
        // place here code to execute in new thread...
    }
}
```

- Helper class extends Thread
  - Cannot be used if helper class already extends another class...

Main thread calls:

In [ ]:
```
new HelperClass( args ).start();
```

Child thread executes in run(), receives args in constructor...

In [ ]:
```
class HelperClass extends Thread {
    HelperClass( ... ) {
        // Constructor receives any args the helper class needs to run...
    }
    public void run() {
        // place here code to execute in new thread...
    }
}
```