CandidateNum: 164848

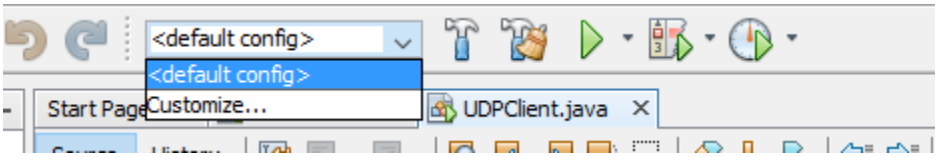
Computer Networks

Assigment2, UDP/TCP

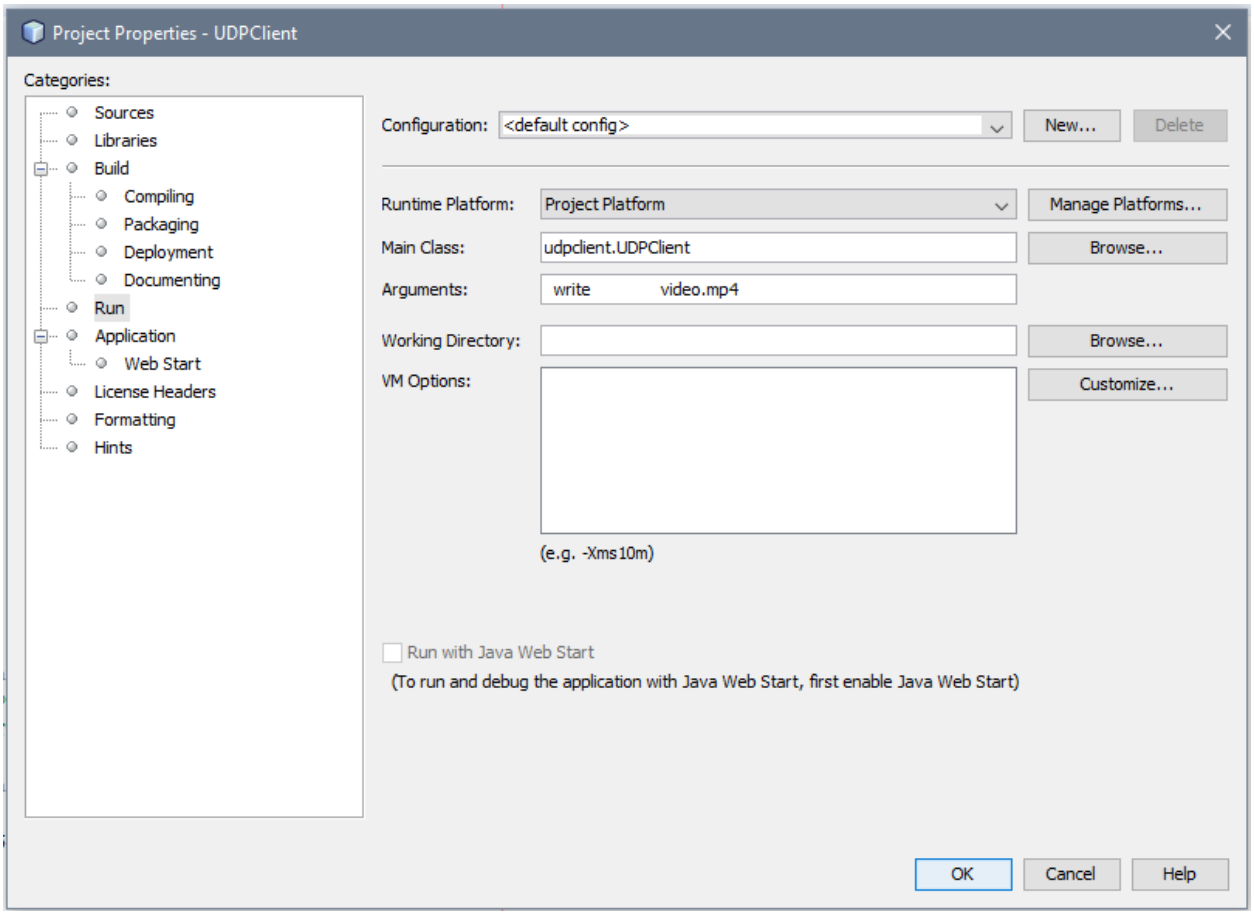
CandidateNum: 164848

How to use

After opening the project for UDPServer/Client or TCPServer/Client check the customize in both UDPClient and TCPClient. Example below



Click on the Customize method and you will see a new window where you can pass your arguments to the client. Example Below



At the section where it says Arguments you have a line where you can type in your first argument that you want the client to send for example by typing either 'read' or 'write' and after you can press space and type in the second argument which is going to be the name of the file with its type. To run the TCP you should customize it the same way as for udp. Example of argument in above Diagram.

Note (1) make sure having a space between two argument otherwise it will understand it as a one argument. (2) make sure that requirement should be 'read' or 'write' with no capitals. (3) Firstly, run the server then only rune the client so that client is able to connect with the server hat is already running.

CandidateNum: 164848

JAVA libraries used

After finishing researches through java docs, I found the exact numbers of libraries needed to combine for use and their classes that are very helpful to implement.

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.net.SocketException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;
import java.net.SocketTimeoutException;
import java.util.Random;
```

Since the approach involved working with bytes I have implemented io and nio libraries that are very necessary if I want to create bytes or array of bytes in case if I want to force any variable to be converted into bytes or transfer from bytes into a variable or a file. The main difference between io and nio libraries is that the io library is known as 'blocking' and nio known as 'non-blocking', that means when in a java program we run threads that invokes a read() or write() classes, then with the io library the thread is blocked unless there is some data to use in a field where it was implemented, on the other hand JAVA nio non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available. Both io and nio libraries are equally effective and in some cases when we deal with multithreading nio might be easier to implement while coding. However, both libraries are possible to combine and keep for further use. The .NET libraries were used to create a socket both for a client and a server, the socket is important if we want a connection between a client and a main server both for udp and tcp, and finally the last library util.random was implemented to initialize a random number of a port both in a client and in a server thread.

UDP Client

All the variables are declared for a later use to fulfill the steps as it is followed in RFC 1350 doc.

```
//the default port of amin server and default ip
private  final String TFTP_SERVER_IP = "127.0.0.1";
private  final int TFTP_DEFAULT_PORT = 1024;

//generate a random port for a client
private Random r = new Random();
private int portClient = r.nextInt(65535 - 1024 + 1) + 1024;
```

CandidateNum: 164848

The TFTP_SERVER_IP and TFTP_DEFAULT_PORT are default ip for both client and a server and default port for the main server. The reason to create a default port for main server only was because that according to the RFC the server can deal with multiple client at the time but a client can be connecting to multiple servers. Thus, we need a default port for a min server that will run threads for each client and all these threads will have different ports just like those clients but to connect to all these threads we need to send the first packet with the request to the main server which port I decided to be 1025 since I can't use any number below 1024. Review figure 2.

Figure 2

```
//the default port of amin server and default ip
private  final String TFTP_SERVER_IP = "127.0.0.1";
private  final int TFTP_DEFAULT_PORT = 1025;
```

In the following code, I have created an instance of Random object in order to give a range of 1024-65535. To be randomly chosen once the client is running, the same approach was used in a server thread once created.

Figure 3

```
//generate a random port for a client
private Random r = new Random();
private int portClient = r.nextInt(65535 - 1024 + 1) + 1024;
```

OP codes and Socket

According to RFC the packet should involve 5 types of opcodes RRQ/WRQ are requests for writing or reading always sent by a client. The ACK/DATAPACKET are the op codes used to understand if the received packet is an acknowledgment or it is a packet that involves data. Lastly, the error op code that in this specific assignment is sent in the packet in case the file is not found. The order of this op codes are scheduled following the RFC doc from 1 to 5 and initialized as bytes for later comparisons. Review figure 4.

Figure 4

```
// TFTP OP Codes
private  final byte OP_RRQ = 1;
private final byte OP_WRQ = 2;
private final byte OP_DATAPACKET = 3;
private final byte OP_ACK = 4;
private final byte OP_ERROR = 5;
```

For making connection we need a datagramSocket and inetAdress that will show the ip address the packet was sent from and also bufferByteArray that will include the data from very received packet. For sending and receiving a packet I have initialized two different instances of datagram packet one for sending a packet which is outBoundDatagramPacket,and for receiving which is inBoundDatagramPacket. According to the RFC the sent and received packets should have same block number so I have initialized a blocknum variable to later increment and check with every received packet in case the data was lost or corrupted so I should to resend it. Review figure 5.

CandidateNum: 164848

Figure 5

```
/**
 * create a datagramSocket
 * and empty instance of inetAddress to later store the address of a received
*/
private DatagramSocket datagramSocket = null;
private InetAddress inetAddress = null;
private byte[] bufferByteArray;

/**
 * create DatagramPackets for storing in the the address
 */
private DatagramPacket outBoundDatagramPacket;
private DatagramPacket inBoundDatagramPacket;
private int blockNum = 1;
```

Client Simulator-method

According to the RFC the request packet should include op code/name of the file/ and the mode code in the same order, so I have created code that will parse 2 arguments to an 'op' byte which will be either WRQ or RRQ initially, and the second argument to the variable string initialized as filename and the mode set to be as default value of 'octet', after I create an array of bytes where I append all those instructions and then I parse it to object packet as an first packet that is going to be send. Review figure 6.

Figure 6

```
//check the arguments
byte op;

if (args[1].equals("read"))
{
    op = OP_RRQ;
}
else if (args[1].equals("write"))
{
    op = OP_WRQ;
}
else{return;}

//creat the default octet and get the filename
String mode = "octet";
String fileName = args[2];

//convert the octet and filename into bytes
byte[] modeByte = mode.getBytes();
byte[] fileNameByte = fileName.getBytes();

//create an array of bites and fiill up with data for the request
ByteArrayOutputStream byteOutOS = new ByteArrayOutputStream();
byteOutOS.write(0);
byteOutOS.write(op);
byteOutOS.write(fileNameByte);
byteOutOS.write(0);
byteOutOS.write(modeByte);
byteOutOS.write(0);

//crweate a packet and fill up with the bites
byte[] packet = byteOutOS.toByteArray();
```

CandidateNum: 164848

In the following code I create a connection and parse the inetAddress and default port that I want to send the first packet, by also setting the timer once connection is made Review figure 7

Figure 7

```
//give the adress of a socket
inetAddress = InetAddress.getByName(TFTP_SERVER_IP);
datagramSocket = new DatagramSocket(portClient);

//set a timer
datagramSocket.setSoTimeout(5000);

// prepare for communication to send
outBoundDatagramPacket = new DatagramPacket(packet,
packet.length, inetAddress, TFTP_DEFAULT_PORT);

// send the request packet
datagramSocket.send(outBoundDatagramPacket);
```

And then I try to receive the respond from the server thread.

```
//create buffer array to store in
bufferByteArray = new byte[516];


// prepare for communication to recieve
inBoundDatagramPacket = new DatagramPacket(bufferByteArray,
bufferByteArray.length, inetAddress,
datagramSocket.getLocalPort());

//try to recieve
try
{
    datagramSocket.receive(inBoundDatagramPacket);
}catch(SocketException e)
{}

//DECLARE AN ARRAY FOR RECIEVING A PACKET FROM A SERVER
ByteArrayOutputStream byteInOS = new ByteArrayOutputStream();
```

CandidateNum: 164848

CHECK for ACK/DATAPACKET/ERROR

After the request is sent and the respond is received and parsed into a BufferByteArray, we start checking in order if the BufferByteArray includes an ack, datapacket or error, to do it I have separated them into 3 else if codes that will work only if there is ack, error or datapcket,

If ACK

If it is an ack that means that the request was Writing and and the server has received so the first thing I do is I check the path of the file and run file and read it all in bytes, othwwerwise I run not found exception. Review figure 8

Figure 8

```
if(bufferByteArray[1] == OP_ACK)
{

    //CHECK FOR FILE AND RUN EXCEPTION IF NOT FOUND ENDING THE METHOD
    byte[] bFile;
    try {
        Path path = Paths.get(fileName);
        bFile = Files.readAllBytes(path);
    } catch (FileNotFoundException e)
    {
        return;
    }
```

So we know that we have to be transferring packets with the size of 516 and we know that the op code and the block number should take 4 bytes in this packet not less or more. Thus, I check the length of a file and call empty integer called currentSize that will be equal to the length – 512*offset . -512 because the max amount of the data I can put in the packet has to be 512, *offset because offset is initially is set to 0 meaning of it will be after incremented based on the number of the packets sent, why because I always to know how much byte are left to send what was the last point of the derived byte

EXAMPLE:

> Let's say I have a file that is equal to 1576 bytes, I just sent the first packet that involves 512 data, that means (offset=0) * 512 = 0. Thus 0 is the last point from I derived bytes till 512 length, now I send one more packet with incremented offset then (offset=1) * 512 = 512, and then one more (offset=2)*512 =1024, so now we know that 3 packets are sent and the remaining size is (1576-1536) =40 according to the code below

```
if(bFile.length < 512*(offset+1)){
    currentSize = bFile.length - (512*offset);
    done = true;
}else{
    currentSize = 512;
}
```

After sending the first packet with data it goes to thedo while loop where it tries to receive a file otherwise it runs socke timous exception and resends the last sent packet, once receive I increment the blockunm and compare ig the received blovk num is equal to the block num in the client once they are equal I start appending in a array of op code blocknum and the data, with the same procedure and send it to the server thread which was made to deal with client specifically after it repeats the same procedure once there is nothing left to send, I voze to use do while because I want to send an empty packet event if everything was equally sent. Review figure9

Figure 9

```
//check if the block numbers are equal
if(bufferByteArray[3] == blockNumArray[1])
{
    // set to null to fill up and send again
    packetFill.reset();

    //change the blocknum
    blockNumArray = new byte[2];
    blockNumArray[1]=(byte)blockNum;

    // fill up the array
    packetFill.write(0);
    packetFill.write(OP_DATAPACKET);
    packetFill.write(blockNumArray);

    if (bFile.length < (512 * (offset + 1)))
    {
        currentSize = bFile.length - (512 * offset);
    } else
    {
        currentSize = 512;
    }

    packetFill.write(bFile, (offset * 512), currentSize);

    offset++;

    //store  them in a packet
    byte[] incPacket = packetFill.toByteArray();

    //give the adress and send
    outBoundDatagramPacket = new DatagramPacket(incPacket,
            incPacket.length, inetAddress, inBoundDatagramPacket.getPort
    try
```

CandidateNum: 164848

If Datapacket

The goal is to start receiving datapackets and store them into an array by also sending an acknowledgment for a server thread as a signal that the packet is received. Thus, I have created two array in which I will be appending a data both for acknowledgments and for received data in which I will append all the received data that will be stored and transferred into a file after receiving them all. Review figure 10

Figure 10

```
}else if(bufferByteArray[1] == OP_DATAPACKET)
{
    ByteArrayOutputStream ackFill = new ByteArrayOutputStream();
    ByteArrayOutputStream recieveed = new ByteArrayOutputStream();


    byte[] blockNumArray = new byte[2];
    blockNumArray[1]=(byte)blockNum;

    ackFill.write(0);
    ackFill.write(OP_ACK);

    try {
        ackFill.write(blockNumArray);
    } catch (IOException ex) {
        //Logger.getLogger(ServerThread.class.getName()).log(Level.SEVERE, null, ex);
    }


    //fill up the packet
    byte[] ackPack = ackFill.toByteArray();

    //prepare the connection
    outBoundDatagramPacket = new DatagramPacket(ackPack,
                ackPack.length, inBoundDatagramPacket.getAddress(), inBoundDatagramPacket.getPort());

    // send the first ack
    try
    {

        datagramSocket.send(outBoundDatagramPacket);

    } catch (SocketException e) {} catch (IOException ex) {}

    //recieve an array
    recieveed.write(bufferByteArray, 4, inBoundDatagramPacket.getLength()-4);
```

So far it is all straight forward following the same procedure, the ack is sent when the data packet is received, check if the blocknums are equal then fill up the ack packet and send, once sent repeat receive, check, send, etc. Review the figure 11

CandidateNum: 164848

Figure 11

```
do
{
    //increment the block num for every packet
    blockNum++;

    //force the in varible block num to be in bytes of 2
    blockNumArray = new byte[2];
    blockNumArray[1]=(byte)blockNum;

    //recieve the next packet
    datagramSocket.receive(inBoundDatagramPacket);

    if(bufferByteArray[3] == blockNumArray[1])
    {
        recieveed.write(bufferByteArray, 4, inBoundDatagramPacket.getLength()-4);
        try
        {
            // set to 0 before you send
            ackFill.reset();
            ackFill.write(0);
            ackFill.write(OP_ACK);
            ackFill.write(blockNumArray);

            //create an packet
            byte[] ackPacket = ackFill.toByteArray();

            //make connection
            outBoundDatagramPacket = new DatagramPacket(ackPacket,
                    ackPacket.length, inBoundDatagramPacket.getAddress(), inBoundDatagramPacket.getPort());

            // send the packet
            datagramSocket.send(outBoundDatagramPacket);

            //if(inBoundDatagramPacket.getLength() < 516){break;}
        } catch (SocketException e) {}
    }
} while (inBoundDatagramPacket.getLength() == 516);
```

If Error

If the error was sent that means the file is not found and according to the specs I clos the socket . Review figure 12.

Figure 12

```
}else if(bufferByteArray[1] == OP_ERROR)
{
    datagramSocket.close();
}
```

The datasocket is closed when the compiler finish ‘if conditions’.

UDPServer

The UDPSERVER can send packet only consisting with 3 different op codes and those op code are ‘ack’ ‘datapacket’ ‘error’, Since my goal was to make a server that can deal with multiple clients at the time, I have made an main Server with its Default port as 1025 and I am running a multithreadin on to be able to run instance of a server to would be able to deal with a client. For example once the request is sent the packet will parse to a new server thread that will have randomized number of a port in range of 1024-65535 and will start dealing with this client separately, so this way no matter how many clients connect with the server many server thread will be created and will be dealing with these client at the time. Example of multithreading on figure 13

Figure 13

```
private Server() throws SocketException
{
    // create a socket
    datagramSocket = new DatagramSocket(TFTP_DEFAULT_PORT);

    // try - create a new server thread parse the recieved packet to new server
    try {
        bufferByteArray = new byte[516];
        while (true)
        {
            inBoundDatagramPacket = new DatagramPacket(bufferByteArray,
                    bufferByteArray.length, inetAddress,
                    datagramSocket.getLocalPort());

            datagramSocket.receive(inBoundDatagramPacket);

            (new Thread(new ServerThread(inBoundDatagramPacket))).start();
        }
    } catch (SocketException e)
    {

    }catch (IOException e)
    {

    }
}
```

UPDServerThread

The main difference between my server and client is that the server has constructor that gets the parsed packet and starts checking the inetAddress with the port number of the client from which the packet was sent from and only then it starts making a socket for a communication and sets. Review the example of a constructor on figure 14.

Figure 14

```
public ServerThread(DatagramPacket packet)
{
    this.packet = packet;
}
```

CandidateNum: 164848

In the server thread, it sends an error packet if it could not find the path of the file that has the same name and type as it was written in the request packet. Review Figure 15

Figure 15

```
//get the file name chek the path and read them in bytes
try
{
    Path path = Paths.get(name);
    bFile = Files.readAllBytes(path);

}catch(FileNotFoundException e)
{} catch (NoSuchFileException ex)
{
    byte[] errorNum = new byte[2];
    ByteArrayOutputStream errorArray = new ByteArrayOutputStream();
    errorArray.write(0);
    errorArray.write(OP_ERROR);
    errorArray.write(errorNum);
    errorArray.write(errorByte);
    errorArray.write(0);
    ////////////////////////////////////////////////////////////////////////fill up the packet
    byte[] packetError = errorArray.toByteArray();
    ////////////////////////////////////////////////////////////////////////connection
    outBoundDatagramPacket = new DatagramPacket(packetError,
        packetError.length, packet.getAddress(), packet.getPort());
    ////////////////////////////////////////////////////////////////////////
    datagramSocket.send(outBoundDatagramPacket);
}
```

TCPClient/Server

As described at the first Paragraph the TCPclient can be customized and compiled the same way as UDP and. My TCP is all constructed followed the rules of TCP protocol. But I could not implement it to work without datagrampacket so the file is not derived or sen in sequence of bytes but is divided into datagram packets and sent just like in the UDP. However, this is a poor example of a TCP protocol but it still follows the full description of a TCP.

Example1:

the writing request is sent the TCP receives and starts sending multiple packets without waiting for the acks from a client.

Example2:

The reading request is sent, the server makes connection sends an ack and waits for a client to send a packet without sending acknowledgments back to a client.