Balcueva, J. Escalona, J.M.

Fadrigo, J.A.M. Fortiz, P.R.

NSCOM01

TFTP Client - Program Design

Program Specifications

1. Program Language: Java
2. Interface: GUI
3. Target Features:
    a. Key Features
        i. GUI or a command line-based user interface are acceptable.
        ii. The user is allowed to specify the server IP address.
        iii. Support for both upload and download of binary files.
        iv. When uploading, the program can send any file on the computer to the TFTP server as long as the file is accessible to the user using his / her OS privileges.
        v. When downloading, the program must allow the user to provide the filename to use when saving the downloaded file to the client's computer.
        vi. Proper error handling at the minimum should include the following:
            1. Timeout for unresponsive server
            2. Handling of duplicate ACK
            3. User prompt for file not found, access violation, and disk full errors
    b. Optional Features
        i. Support for option negotiation will merit additional points if correctly implemented
            1. Option to specify the transfer block size
            2. Communicate transfer size to a server when uploading
        ii. To allow the user to manually ping the target host prior to transmission.

Features implemented:

1. TFTP protocol-based features
    a. Uploading and downloading files (at unlimited file sizes)
    b. Error detection and handling
    c. Blocksize modification
    d. Options recognition and compliance (blksize and tsize only)
2. Non-TFTP protocol-based features
    a. Internal timeouts (3 seconds)
    b. Network verification before transmission

Limitations:

1. Does not use the official TFTP timeout option
2. Cannot verify ACKs beyond a block value of 65535, thus cannot implement duplicate ACK handling. However, Wireshark packet analysis shows that the block number segment of the packet cycles back to 0 once it reaches beyond 65535 which could be used to augment the succeeding values beyond 65535 but was not implemented accordingly due to potential reliability issues on the rest of the application.
3. Since the application is single-threaded the UX might feel sluggish, especially if files are big which can take a while to be sent/received.
4. TFTPd64 may show a limited file size of 2147483647 bytes (~2.1GB), but it can still accept transmission by the client exceeding that displayed file size (i.e., progress shows beyond 100% at the limit file size). This can be attributed to the integer value limit of 2.1B.

Key Classes List

1. Client - Conducts the file reading/writing and transmission of TFTP packets.
2. TFTP - Builds the TFTP packets that will be used in TFTP transmission. The class also includes methods for checking and validating packet parameters that can affect the packet's overall validity and usability in the TFTP protocol.

TFTP Packet Diagram

The design of the packets was heavily referenced from the samples from the RFC documents and actual TFTP packets using TFTPd64 and Wireshark. The resulting packet tables are as follows. The table shows both ways of appending a padding byte for strings where if the string (as per ASCII specification) does have a 0\, then it does not need a padding byte. The opposite goes if it does have \0.

1. REQUEST (READ/WRITE, W/O OPTVALS)

Request (w/o OptVals)						
Length	2Bytes		Length of String	1Byte	Length of String	1Byte
Segment	Padding	Type (1/2)	Filename	Padding	Mode	Padding

2. REQUEST (READ/WRITE, W OPTVALS)

Request (w/o OptVals)											
Length	2Bytes		Length of String	1Byte	Length of String	1Byte	Length of String	Length of String	…	Length of String	Length of String
Segment	Padding	Type (1/2)	Filename	Padding	Mode	Padding	Opt1 (ends in \0)	Val1 (ends in \0)	…	OptN (ends in \0)	ValN (ends in \0)

3. DATA

Data				
Length	2Bytes		2Bytes	n Bytes
Segment	Padding	3	Block#	Data

4. ACK

ACK			
Length	2Bytes		2Bytes
Segment	Padding	4	Block#

5. OACK

OACK							
Length	2Bytes		Length of String	Length of String	…	Length of String	Length of String
Segment	Padding	6	Opt1 (ends in \0)	Val1 (ends in \0)	…	OptN (ends in \0)	ValN (ends in \0)

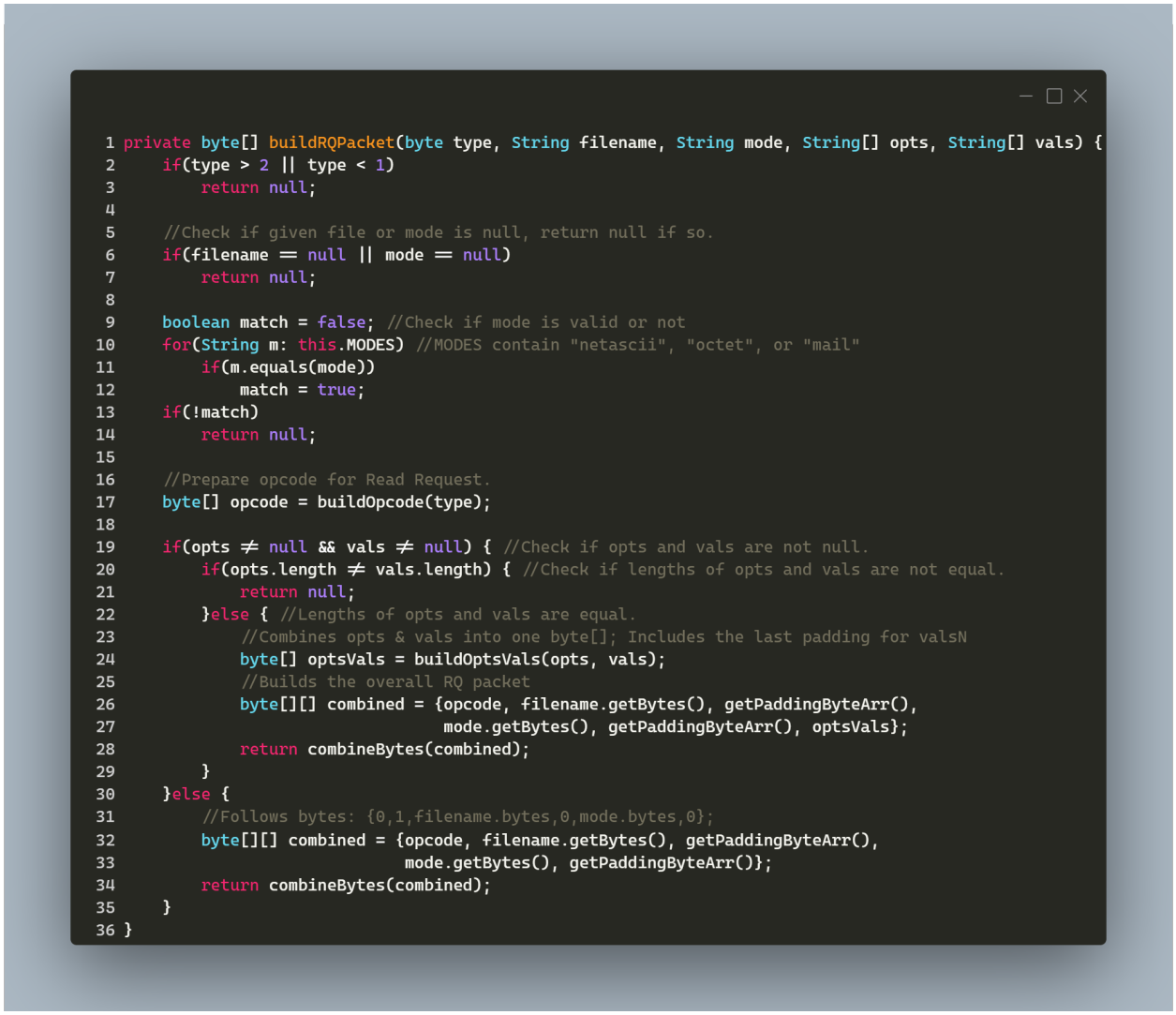
6. ERROR PACKET

Error					
Length	2Bytes		2Bytes	Length of String	1Byte
Segment	Padding	5	Error Code	ErrMsg	Padding

TFTP Packet Assembly Code

This segment contains the functions used in assembling TFTP packets that will be used in the TFTP transmission. Some of the functions are abstracted from the code shown. The contents of the abstracted methods/functions are found in the appendix below. The method parameters are also pre-validated before methods are called.

1. Request Packet (w/ or w/o OptVals, and whether Read=1 or Write=2)



```java
private byte[] buildRQPacket(byte type, String filename, String mode, String[] opts, String[] vals) {
    if(type > 2 || type < 1)
        return null;

    //Check if given file or mode is null, return null if so.
    if(filename == null || mode == null)
        return null;

    boolean match = false; //Check if mode is valid or not
    for(String m: this.MODES) //MODES contain "netascii", "octet", or "mail"
        if(m.equals(mode))
            match = true;
    if(!match)
        return null;

    //Prepare opcode for Read Request.
    byte[] opcode = buildOpcode(type);

    if(opts != null && vals != null) { //Check if opts and vals are not null.
        if(opts.length != vals.length) { //Check if lengths of opts and vals are not equal.
            return null;
        }else { //Lengths of opts and vals are equal.
            //Combines opts & vals into one byte[]; Includes the last padding for valsN
            byte[] optsVals = buildOptsVals(opts, vals);
            //Builds the overall RQ packet
            byte[][] combined = {opcode, filename.getBytes(), getPaddingByteArr(),
                                 mode.getBytes(), getPaddingByteArr(), optsVals};
            return combineBytes(combined);
        }
    }else {
        //Follows bytes: {0,1,filename.bytes,0,mode.bytes,0};
        byte[][] combined = {opcode, filename.getBytes(), getPaddingByteArr(),
                             mode.getBytes(), getPaddingByteArr()};
        return combineBytes(combined);
    }
}
```

2. OACK Packet

```java
private byte[] buildOACKPacket(String[] opts, String[] vals) {
    if(opts == null || vals == null)
        return null;
    if(opts.length != vals.length)
        return null;
    byte opcodeVal = 6;
    byte[] optCode = buildOpcode(opcodeVal);
    byte[] combinedOptsVals = buildOptsVals(opts, vals);
    byte[][] combined = {optCode, combinedOptsVals};
    return combineBytes(combined);
}
```

3. ACK Packet

```java
private byte[] buildACKPacket(Short block) {
    if(block < 0)
        return null;
    byte opcode = 4;
    byte[][] combined = {buildOpcode(opcode), u.shortToByteArr(block)};
    byte[] ack = combineBytes(combined);
    return ack;
}
```

4. Data Packet

```java
private byte[] buildDataPacket(Integer block, byte[] data) {
    if(block < 0)
        return null;
    if(data == null)
        return null;
    byte opcodeVal = 3;
    Short blockShort = block.shortValue();
    byte[] opcode = buildOpcode(opcodeVal), blockNum = u.shortToByteArr(blockShort);
    byte[][] preDataPacket = {opcode,blockNum,data};
    return combineBytes(preDataPacket);
}
```

5. ERROR Packet

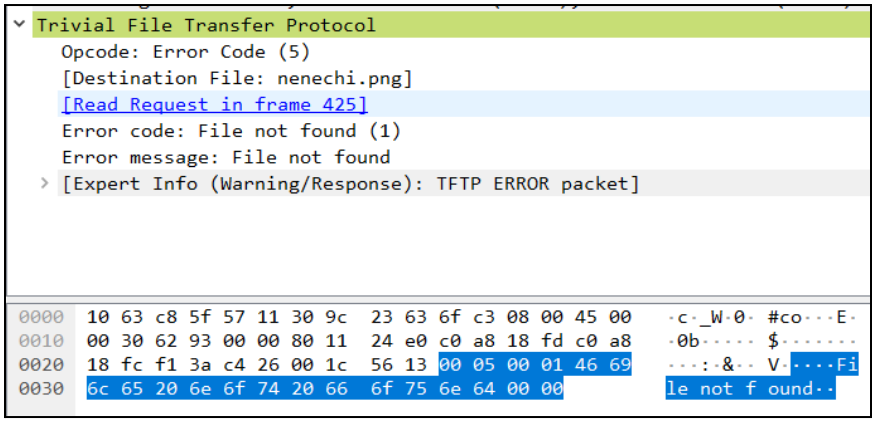
```java
private byte[] buildErrPacket(Integer err, String emsg) {
    //Error Packet
    if(!validErrCode(err))
        return null;
    byte opcodeVal = 5;
    byte[] opcode = buildOpcode(opcodeVal);
    byte[] errcode = {getPaddingByte(), getPaddingByte(), getPaddingByte(),  err.byteValue()};
    byte[] errMsg = emsg.getBytes();
    byte[][] combined = {getPaddingByteArr(), getPaddingByteArr(), opcode, errcode, errMsg,
                         getPaddingByteArr(), getPaddingByteArr()};
    return combineBytes(combined);
}
```

TFTP Packet Assembly Results and Reference

From the packet assembly code shown above, the resulting outputs (parsed to hex and bits) alongside the Wireshark packet reference are as follows.

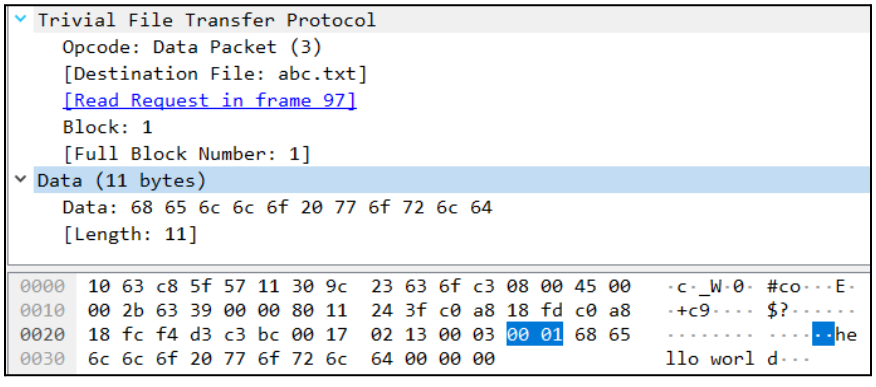
1. Error Packet

```
Error Packet
System:
System Hex from Processed Byte: 0005000146696c65206e6f7420666f756e640000
System Bits: 00000000 00000101 00000000 00000001 01000110 01101001 01101100 01100101 00100
000 01101110 01101111 01110100 00100000 01100110 01101111 01110101 01101110 01100100 00000
000 00000000
Wireshark:
Wireshark Hex Raw: 0005000146696c65206e6f7420666f756e640000
Wireshark Bits: 00000000 00000101 00000000 00000001 01000110 01101001 01101100 01100101 00
100000 01101110 01101111 01110100 00100000 01100110 01101111 01110101 01101110 01100100 00
000000 00000000
isError: true
Extract Error: 1 = File not found
```



2. Data Packet

```
Data Packet
System:
System Hex from Processed Byte: 030168656c6c6f20776f726c64
System Bits: 00000011 00000001 01101000 01100101 01101100 01101100 01101111 00100000 01110
111 01101111 01110010 01101100 01100100
Wireshark:
Wireshark Hex Raw: 0003000168656c6c6f20776f726c64
Wireshark Bits: 00000000 00000011 00000000 00000001 01101000 01100101 01101100 01101100 01
101111 00100000 01110111 01101111 01110010 01101100 01100100
getOpCode: 3
(2022/06/16 10:13:15) TFTP.extractData(): 01101000 01100101 01101100 01101100 01101111 001
00000 01110111 01101111 01110010 01101100 01100100
Extract Data: 01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 0111
0010 01101100 01100100
```



3. ACK Packet

```
ACK Packet
System:
System Hex from Processed Byte: 04054
System Bits: 00000100 00000101
Wireshark:
Wireshark Hex Raw: 00040054
Wireshark Bits: 00000000 00000100 00000000 01010100
isACK: true
extractACK: Block 84
```

```
Trivial File Transfer Protocol
    Opcode: Acknowledgement (4)
    [Destination File: tote_tilt.jpg]
    [Write Request in frame 563]
    Block: 84
    [Full Block Number: 84]
```

```
0000  10 63 c8 5f 57 11 30 9c  23 63 6f c3 08 00 45 00   ·c·_W·0· #co···E·
0010  00 20 62 e9 00 00 80 11  24 9a c0 a8 18 fd c0 a8   · b····· $·······
0020  18 fc c1 1a df 53 00 0c  ab c5 00 04 00 54 00 00   ·····S·· ·····T··
0030  00 00 00 00 00 00 00 00  00 00 00 00              ········ ····
```

4. OACK Packet

```
OACK Packet
System:
System Hex from Processed Byte: 067473697a65038313936370
System Bits: 00000110 01110100 01110011 01101001 01111010 01100101 00000011 10000011 00010
011 10010011 01100011 01110000
Wireshark:
Wireshark Hex Raw: 00067473697a6500383139363700
Wireshark Bits: 00000000 00000110 01110100 01110011 01101001 01111010 01100101 00000000 00
111000 00110001 00111001 00110110 00110111 00000000
extractOACK: {tsize}, {81967}
```

```
Trivial File Transfer Protocol
    Opcode: Option Acknowledgement (6)
    [Destination File: tote_tilt.jpg]
    [Write Request in frame 563]
  > Option: tsize = 81967
```

```
0000  10 63 c8 5f 57 11 30 9c  23 63 6f c3 08 00 45 00   ·c·_W·0· #co···E·
0010  00 2a 62 95 00 00 80 11  24 e4 c0 a8 18 fd c0 a8   ·*b····· $·······
0020  18 fc c1 1a df 53 00 16  c0 ad 00 06 74 73 69 7a   ·····S·· ····tsiz
0030  65 00 38 31 39 36 37 00  00 00 00 00              e·81967· ····
```

5. Read Request (With and Without OptsVals)

```
RRQ Packet
System:
System Hex from Processed Byte: 016e656e656368692e706e6706f6374657407473697a650300
System Bits: 00000001 01101110 01100101 01101110 01100101 01100011 01101000 01101001 00101
110 01110000 01101110 01100111 00000110 11110110 00110111 01000110 01010111 01000000 01110
100 01110011 01101001 01111010 01100101 00000011 00000000
Wireshark:
Wireshark Hex Raw: 00016e656e656368692e706e67006f6374657407473697a65003000
Wireshark Bits: 00000000 00000001 01101110 01100101 01101110 01100101 01100011 01101000 01
101001 00101110 01110000 01101110 01100111 00000000 01101111 01100011 01110100 01100101 01
110100 00000000 01110100 01110011 01101001 01111010 01100101 00000000 00110000 00000000
RQHasOACK: true
extractOACKFromRQ: {tsize}, {0}
```

```
RRQ Packet Without Opts & Vals
System:
System Hex from Processed Byte: 016e656e656368692e706e6706f637465740
System Bits: 00000001 01101110 01100101 01101110 01100101 01100011 01101000 01101001 00101
110 01110000 01101110 01100111 00000110 11110110 00110111 01000110 01010111 01000000
Wireshark:
Wireshark Hex Raw: 00016e656e656368692e706e67006f6374657400
RQHasOACK: false
Wireshark Bits: 00000000 00000001 01101110 01100101 01101110 01100101 01100011 01101000 01
101001 00101110 01110000 01101110 01100111 00000000 01101111 01100011 01110100 01100101 01
110100 00000000
```

```
▾ Trivial File Transfer Protocol
    Opcode: Read Request (1)
    Source File: nenechi.png
    Type: octet
  › Option: tsize = 0
```

```
0000  30 9c 23 63 6f c3 10 63  c8 5f 57 11 08 00 45 00   0·#co··c ·_W···E·
0010  00 38 12 e1 00 00 80 11  74 8a c0 a8 18 fc c0 a8   ·8····· t·······
0020  18 fd c4 26 00 45 00 24  3c 67 00 01 6e 65 6e 65   ···&·E·$ <g··nene
0030  63 68 69 2e 70 6e 67 00  6f 63 74 65 74 00 74 73   chi.png· octet·ts
0040  69 7a 65 00 30 00                                 ize·0·
```

6. Write Request (With and Without OptsVals)

```
WRQ Packet
System:
System Hex from Processed Byte: 02746573742e706e6706f63746574074734697a65038313936370
System Bits: 00000010 01110100 01100101 01110011 01110100 00101110 01110000 01101110 01100
111 00000110 11110110 00110111 01000110 01010111 01000000 01110100 01110011 01101001 01111
010 01100101 00000011 10000011 00010011 10010011 01100011 01110000
Wireshark:
Wireshark Hex Raw: 0002746f74655f74696c742e6a7067006f63746574007473697a6500383139363700
RQHasOACK: true
extractOACKFromRQ: {tsize}, {81967}
Wireshark Bits: 00000000 00000010 01110100 01101111 01110100 01100101 01011111 01110100 01
101001 01101100 01110100 00101110 01101010 01110000 01100111 00000000 01101111 01100011 01
110100 01100101 01110100 00000000 01110100 01110011 01101001 01111010 01100101 00000000 00
111000 00110001 00111001 00110110 00110111 00000000
```

```
WRQ Packet Without Opts & Vals
System:
System Hex from Processed Byte: 02746573742e706e6706f637465740
System Bits: 00000010 01110100 01100101 01110011 01110100 00101110 01110000 01101110 01100
111 00000110 11110110 00110111 01000110 01010111 01000000
Wireshark:
Wireshark Hex Raw: 0002746f74655f74696c742e6a7067006f6374657400
RQHasOACK: false
Wireshark Bits: 00000000 00000010 01110100 01101111 01110100 01100101 01011111 01110100 01
101001 01101100 01110100 00101110 01101010 01110000 01100111 00000000 01101111 01100011 01
110100 01100101 01110100 00000000
```

```
▾ Trivial File Transfer Protocol
    Opcode: Write Request (2)
    Destination File: tote_tilt.jpg
    Type: octet
  › Option: tsize = 81967
```

```
0000  30 9c 23 63 6f c3 10 63  c8 5f 57 11 08 00 45 00   0·#co··c ·_W···E·
0010  00 3e 12 e5 00 00 80 11  74 80 c0 a8 18 fc c0 a8   ·>····· t·······
0020  18 fd df 53 00 45 00 2a  32 41 00 02 74 6f 74 65   ···S·E·* 2A··tote
0030  5f 74 69 6c 74 2e 6a 70  67 00 6f 63 74 65 74 00   _tilt.jp g·octet·
0040  74 73 69 7a 65 00 38 31  39 36 37 00              tsize·81 967·
```

7. Data Packet (wrong output)
   The output is possibly wrong due to issues in decoding in and out of byte[] and Hex in Java. The function for decoding byte to Hex was done through Integer.parseInt(<Str>,16) & Integer.toHexString(<byte>) which has issues with overflow from Hex values of 80 and above as shown in the code below.

```
Data Packet
System:
System Hex from Processed Byte: 030ffffffa1ffffff9452ffffffd1409452d140ffffff9452ffffffd14
09452d140ffffff9452ffffffd1409452d140ffffff9452ffffffd1409452d140ffffff9452ffffffd1409452d
141fffffffffffffd9
System Bits: 00000011 00001111 11111111 11111111 11111010 00011111 11111111 11111111 11111
001 01000101 00101111 11111111 11111111 11111101 00010100 00001001 01000101 00101101 00010
100 00001111 11111111 11111111 11111001 01000101 00101111 11111111 11111111 11111101 00010
100 00001001 01000101 00101101 00010100 00001111 11111111 11111111 11111001 01000101 00101
111 11111111 11111111 11111101 00010100 00001001 01000101 00101101 00010100 00001111 11111
111 11111111 11111001 01000101 00101111 11111111 11111111 11111101 00010100 00001001 01000
101 00101101 00010100 00001111 11111111 11111111 11111001 01000101 00101111 11111111 11111
111 11111101 00010100 00001001 01000101 00101101 00010100 00011111 11111111 11111111 11111
111 11111111 11111111 11111111 11111101
Wireshark:
Wireshark Hex Raw: 000300a19452d14009452d14009452d14009452d14009452d14009452d14009452d1400
9452d14009452d14009452d1401ffd9
Wireshark Bits: 00000000 00000011 00000000 10100001 10010100 01010010 11010001 01000000 00
001001 01000101 00101101 00010100 00000000 10010100 01010010 11010001 01000000 00001001 01
000101 00101101 00010100 00000000 10010100 01010010 11010001 01000000 00001001 01000101 00
101101 00010100 00000000 10010100 01010010 11010001 01000000 00001001 01000101 00101101 00
010100 00000000 10010100 01010010 11010001 01000000 00001001 01000101 00101101 00010100 00
000001 11111111 11011001
```

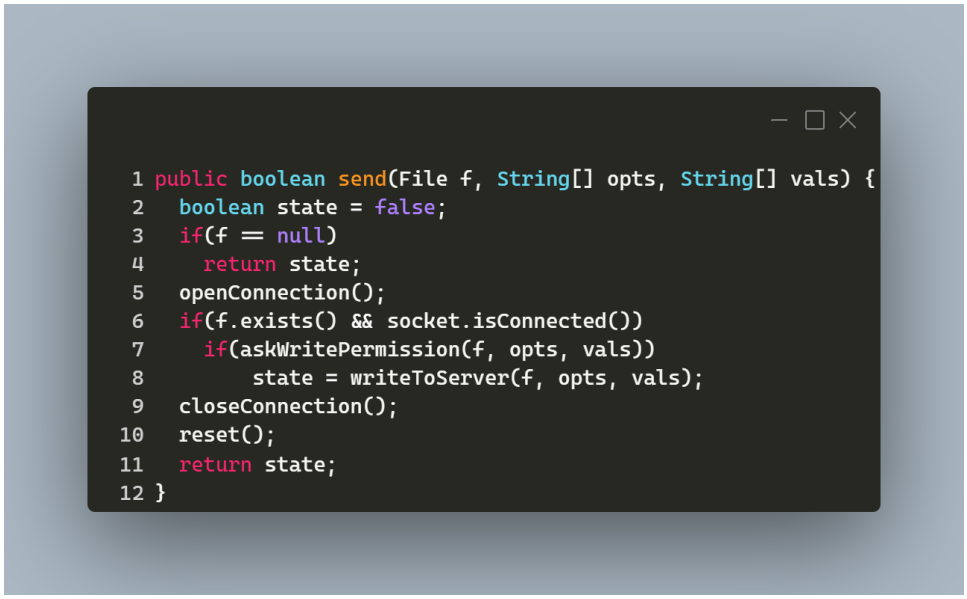
```java
public static void main(String[] args) {
  String src = "0123456789abcdef";
  System.out.println("Raw: byte, hexString");
  for(int i = 0; i < src.length(); i++){
    for(int j = 0; j < src.length(); j++){
      String raw = src.charAt(i) + "" +src.charAt(j);
      Integer hex = Integer.parseInt(raw,16); //String to Hex's Integer Equivalent
      byte b = hex.byteValue(); //Byte value of Hex
      String hexString = Integer.toHexString(b); //Hex string of Hex's byte value
      System.out.println(raw + ": " + b + ", " + hexString);

      //SAME THING GOES IF hex IS IN SHORT DATA TYPE.
    }
  }
}
```

```
78: 120, 78
79: 121, 79
7a: 122, 7a
7b: 123, 7b
7c: 124, 7c
7d: 125, 7d
7e: 126, 7e
7f: 127, 7f
80: -128, ffffff80
81: -127, ffffff81
82: -126, ffffff82
83: -125, ffffff83
84: -124, ffffff84
85: -123, ffffff85
86: -122, ffffff86
87: -121, ffffff87
88: -120, ffffff88
```

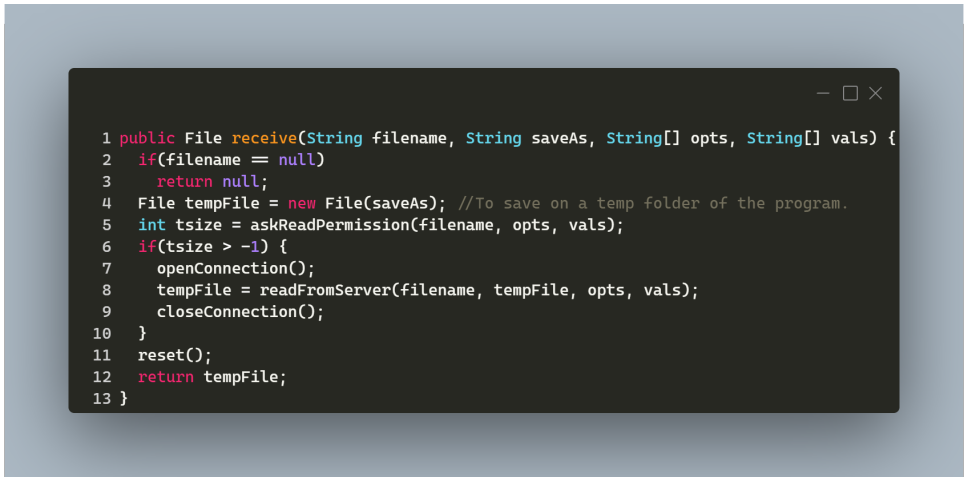
Network Sequence Code

The function call for sending and receiving is abstracted publicly into two functions which delegate the entire network-related processes of the TFTP connection prior to the actual transmission.

The sequences contain the opening and closing of the socket connection as it is based on the understanding that TFTP only requires a connection if the user has an intent of conducting a TFTP transmission. It also handles the permission calls and assessment as well as the call for reading or writing to a server if permission is granted. The connection configuration is specified from the function's object instantiator.



```java
public boolean send(File f, String[] opts, String[] vals) {
  boolean state = false;
  if(f == null)
    return state;
  openConnection();
  if(f.exists() && socket.isConnected())
    if(askWritePermission(f, opts, vals))
        state = writeToServer(f, opts, vals);
  closeConnection();
  reset();
  return state;
}
```



```java
public File receive(String filename, String saveAs, String[] opts, String[] vals) {
  if(filename == null)
    return null;
  File tempFile = new File(saveAs); //To save on a temp folder of the program.
  int tsize = askReadPermission(filename, opts, vals);
  if(tsize > -1) {
    openConnection();
    tempFile = readFromServer(filename, tempFile, opts, vals);
    closeConnection();
  }
  reset();
  return tempFile;
}
```

TFTP Sequence Diagrams and Code

For the sequence diagram and code, the following already follows the code mentioned in the Network Sequence which calls for askWritePermission() and askReadPermission() as well as the writeToServer() and readFromServer(). The sequence that follows shows the equivalent pseudocode.

```
private File readFromServer(String filename, File tempFile, String[] opts, String[] vals) {
    String methodName = "readFromServer()";

    if(!socket.isConnected())
        return null;

    if(!tempFile.exists())
        try {
            tempFile.createNewFile();
        } catch (Exception e1) {
            gui.popDialog("Error creating new file for the download!", "Error",
                          JOptionPane.ERROR_MESSAGE);
            return null;
        }

    OutputStream outputStream = null; //BYTE STREAM FILE WRITING
    int tsize = this.TSIZE, blocksize = this.BUFFER_SIZE + 4, timeout = -1; //FOR CONFIGURATION

    int subtotal = 0;
    try {
        outputStream = new FileOutputStream(tempFile);

        int ctr = 0;
        boolean validBlock = false, error = false;
        do {
            do {
                validBlock = false;
                byte[] ackbyte = tftp.getACK(ctr);

                //RECOMPUTE BLKCSIZE IF COMPUTED REMAINING BYTES OF
                //FILE IS LESS THAN BUFFER/BLKSIZE; DO NOT MOVE
                if(tsize-subtotal < blocksize) {
                    blocksize = tsize-subtotal + 4;
                }

                //SEND AN ACK FIRST
                if(ctr == 0) { //TO SYNC WITH FUTURE DATAPACKETS THAT START AT 1
                    packet = new DatagramPacket(ackbyte, ackbyte.length);
                    socket.send(packet);
                    ctr++;
                }

                //AWAIT FOR DATA RESPONSE
                byte[] buffer = new byte[blocksize];
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);

                boolean isData = tftp.isData(packet.getData());
                boolean isError = tftp.isError(packet.getData());
                if(isData) {
                    //SAVE BUFFER TO FILE
                    byte[] data = tftp.extractData(packet);
                    int block = tftp.extractBlockNumber(packet);
                    subtotal += data.length;
                    if(true/*block == ctr*/) { //CLIENT AND SERVER ARE IN SYNC TODO
                        validBlock = true;
                        int bytesRead = data.length; //BYTE LENGTH OF PACKET'S DATA SEGMENT
                        outputStream.write(data, 0, bytesRead);
                    }if(block > ctr) { //SERVER IS ADVANCED THAN CLIENT
                        //
                    }else{ // SERVER IS LATE THAN CLIENT
                        packet = new DatagramPacket(ackbyte, ackbyte.length);
                        socket.send(packet);
                    }
                }else if(isError){
                    String[] err = tftp.extractError(packet.getData());
                    //HANDLE ERRORS HERE
                    tempFile.delete();
                }else{
                    //UNEXPECTED PACKET
                }
            }while(!validBlock && !error);
            byte[] ackbyte = tftp.getACK(ctr);
            packet = new DatagramPacket(ackbyte, ackbyte.length);
            socket.send(packet);
            ctr++;
        }while(subtotal < tsize);
        outputStream.close();
    } catch (Exception e) {
        gui.popDialog("Exception occured:\n" + e.getLocalizedMessage(), "Error",
                      JOptionPane.ERROR_MESSAGE);
        return null;
        //FOR ANY CATCH(EXCEPTION) THAT OCCURS: SEND AN ERR PACKET, DELETE TEMPFILE, RETURN NULL
        //Timeout localizedMessage value is "Receive timed out"
    }
    return tempFile;
}
```

```java
private boolean writeToServer(File f, String[] opts, String[] vals) {
    String methodName = "writeToServer(f,opts,vals)";

    if(!socket.isConnected())
        return false;

    try {
        Integer SIZE = (int)Files.size(f.toPath()); //SIZE OF FILE
        Integer bytesRead = -1; //FOR FILE STREAMING

        InputStream inputStream = new FileInputStream(f.getAbsolutePath()); //FILE STREAMING

        int tsize = this.TSIZE, blocksize = this.BUFFER_SIZE, timeout = -1; //FOR CONFIGURATION

        //BUFFER BYTE[] CONFIGURATION
        byte[] buffer = new byte[blocksize]; //DATA SEGMENT OF PACKET

        //IF FILE SIZE IS INITIALLY SMALLER THAN BUFFER SIZE THEN SET BUFFER TO JUST FILE'S SIZE
        if(SIZE < blocksize)
            buffer = new byte[SIZE];

        int ctr = 1, ACKval = 0; //COUNTERS FOR BLOCK#
        boolean error = false, validACK = false;
        while((bytesRead = inputStream.read(buffer)) > 0) { //While file not done streaming.
            validACK = false;
            do {
                //Sending byte of file
                byte[] packetByte = tftp.getDataPacket(ctr, buffer); //BUILD A DATA TFTP PACKET
                packet = new DatagramPacket(packetByte,packetByte.length);
                socket.send(packet);

                //Await for response
                packet = new DatagramPacket(new byte[blocksize], blocksize);
                socket.receive(packet);

                boolean isACK = tftp.isACK(packet.getData()), isError = tftp.isError(packet.getData());
                if(isACK) {
                    ACKval = tftp.extractBlockNumber(packetByte);
                    //Change to comparing received block number in ACK to ctr TODO
                    if(true/**ACKval == ctr*/) {
                        validACK = true;
                        ctr++;
                    }
                }else if(isError){
                    //Structure at {Error Code, Error Message}
                    String[] err = tftp.extractError(packet.getData());
                    //HANDLE ERRORS HERE
                }else {
                  //
                }
            }while(!validACK && !error);
            //Repeat if sending if the ACK received is not a 'new' block of data.

            //RECOMPUTES BLKSIZE IF AVAILABLE DATA IS LESS THAN THE BLKSIZE;
            //DO NOT MOVE THIS. LET IT BE PLACED LAST.
            if(inputStream.available() < blocksize) {
                blocksize = inputStream.available();
                buffer = new byte[blocksize];
            }
        }
        inputStream.close();
        return true;
    } catch (Exception e) {
        gui.popDialog("Exception occured:\n" + e.getLocalizedMessage(), "Error",
                      JOptionPane.ERROR_MESSAGE);
    }
    return false; //A fatal error (non-TFTP) occurs.
}
```

```java
private int askReadPermission(String filename, String[] opts, String[] vals) {
    String methodName = "askReadPermission(filename,opts,vals)";
    this.TSIZE = -1;
    this.BUFFER_SIZE = 512;
    if(!isConnected() || filename == null || filename.length() == 0)
        return this.TSIZE;
    u.printMessage(this.className, methodName, "Building write request packet ... ");
    String mode = "octet";
    byte[] rrq = tftp.getRRQPacket(filename, mode, opts, vals);
    packet = new DatagramPacket(rrq, rrq.length);
    try {
        //Send packet to serve
        socket.connect(target, this.PORT); //USE 'CONTROL' SOCKET OF TFTP
        socket.send(packet);

        //Receive ACK or ERROR packet
        packet = new DatagramPacket(new byte[512], 512);
        //LIMITED TO 512 AS THE RFC DOCUMENT REVEALS THAT ANY REQUEST IS LIMITED TO 512 OCTETS,
        //IMPLYING THAT ANY OACK MAY BE THE SAME.

        //SWITCH OVER PACKET TO SPECIFIED DATAPORT AND NOT TO PORT 69
        socket.connect(this.target, this.DATAPORT);
        socket.receive(packet);

        //Trim excess bytes from packets
        byte[] trimmedPacket = u.trimPacket(packet, this.className, methodName); //TRIMMED RCV

        //Confirm that the packet is an OACK and not an Error
        boolean isOACK = tftp.isOACK(trimmedPacket), isError = !tftp.isError(trimmedPacket);
        if(isOACK && isError){
            String[][] checking = tftp.extractOACK(trimmedPacket);

            int match = 0;

            //Revamped design which now considers the insisted blksize and tsize by the TFTP server as
noted on a Wireshark test.
            for(int i = 0; i < vals.length; i++) {
                for(int j = 0;  j < checking[0].length; j++) {
                    if(opts[i].equalsIgnoreCase(checking[0][j])) { //Same Item
                        if(checking[0][j].equalsIgnoreCase("blksize")){ //Check if TFTP asserts a blksize
                            this.BUFFER_SIZE = Integer.parseInt(checking[1][j]);
                            match++;
                        }else if(checking[0][j].equalsIgnoreCase("tsize")){ //Check tsize TFTP returns
                            this.TSIZE = Integer.parseInt(checking[1][j]);
                            match++;
                        }else if(checking[1][j].equalsIgnoreCase(vals[i]) &&
                                    (!checking[0][j].equalsIgnoreCase("blksize") &&
                                    !checking[0][j].equalsIgnoreCase("tsize"))
                                ){
                            match++;
                        }
                    }
                }
            }
            if(match != vals.length)
                this.TSIZE = -1;
        }else{
            //Confirm that the packet is an Error
            if(isError){
                String[] error = tftp.extractError(trimmedPacket);
                displayError(error, methodName);
                this.TSIZE = -1;
            }
        }
    }catch (Exception e) {
        gui.popDialog("Exception occured:\n" + e.getLocalizedMessage(),"Error",
                      JOptionPane.ERROR_MESSAGE);
    }
    return this.TSIZE; //Modify freely when needed.
}
```

```java
private boolean askWritePermission(File f, String[] opts, String[] vals) {
    String methodName = "askWritePermission(f,opts,vals)";

    boolean permission = false;
    if(!isConnected() || f == null)
        return permission;
    String mode = "octet";
    byte[] wrq = tftp.getWRQPacket(f, mode, opts, vals);
    packet = new DatagramPacket(wrq, wrq.length);
    try {
        //Send packet to serve
        socket.connect(target, this.PORT); //USE 'CONTROL' SOCKET OF TFTP
        socket.send(packet);

        //Receive ACK or ERROR packet
        //LIMITED TO 512 AS THE RFC DOCUMENT REVEALS THAT ANY REQUEST IS LIMITED TO 512 OCTETS,
        //IMPLYING THAT ANY OACK MAY BE THE SAME.
        packet = new DatagramPacket(new byte[512], 512);

        //SWITCH OVER PACKET TO SPECIFIED DATAPORT AND NOT TO PORT 69
        socket.connect(this.target, this.DATAPORT);
        socket.receive(packet);

        //Trim excess bytes from packets
        byte[] trimmedPacket = u.trimPacket(packet, this.className, methodName); //TRIMMED RCV

        //Confirm that the packet is an OACK and not an Error
        if(tftp.isOACK(trimmedPacket) && !tftp.isError(trimmedPacket)){
            String[][] checking = tftp.extractOACK(trimmedPacket);

            int match = 0;

            //Revamped design which now considers the insisted blksize of the
            //TFTP server as noted on a Wireshark test.
            for(int i = 0; i < vals.length; i++) {
                for(int j = 0;  j < checking[0].length; j++) {
                    if(opts[i].equalsIgnoreCase(checking[0][j])) { //Same Item
                        if(checking[0][j].equalsIgnoreCase("blksize")){ //Check if TFTP asserts a blksize
                            this.BUFFER_SIZE = Integer.parseInt(checking[1][j]);
                            match++;
                        }
                        if(checking[1][j].equalsIgnoreCase(vals[i]) &&
                           !checking[0][j].equalsIgnoreCase("blksize")
                          ){
                            match++;
                        }
                    }
                }
            }
            if(match == vals.length)
                permission = true;
        }else{
            //Confirm that the packet is an Error
            if(tftp.isError(trimmedPacket)){
                String[] error = tftp.extractError(trimmedPacket);
                displayError(error, methodName);
            }
        }
    }catch (Exception e) {
        gui.popDialog("Exception occured:\n" + e.getLocalizedMessage(),"Error",
JOptionPane.ERROR_MESSAGE);
    }
    return permission;
}
```

Appendix

1. buildOpcode()

```
private byte[] buildOpcode(byte opcode) {
    byte[] opcodeByte = {getPaddingByte(), opcode};
    return opcodeByte;
}
```

2. buildOptVals()

```
private byte[] buildOptsVals(byte[][] opts, byte[][] vals) {
    if(opts == null || vals == null) {
        return null;
    }else {
        if(opts.length != vals.length) {
            return null;
        }else {
            ArrayList<Byte> optsvals = new ArrayList<Byte>();
            for(int i = 0; i < opts.length; i++) {
                //append opts[i] then add padding
                for(int j = 0;  j < opts[i].length; j++)
                    optsvals.add(opts[i][j]);
                optsvals.add(getPaddingByte());
                //append vals[i] then add padding
                for(int j = 0;  j < vals[i].length; j++)
                    optsvals.add(vals[i][j]);
                optsvals.add(getPaddingByte());
            }
            //convert ArrayList<Byte> to byte[]
            return  u.ByteListToByteArr(optsvals);
        }
    }
}
```

3. getPaddingByte()

```
1 private byte getPaddingByte() {
2   Short padding = 0;
3   return padding.byteValue();
4 }
```

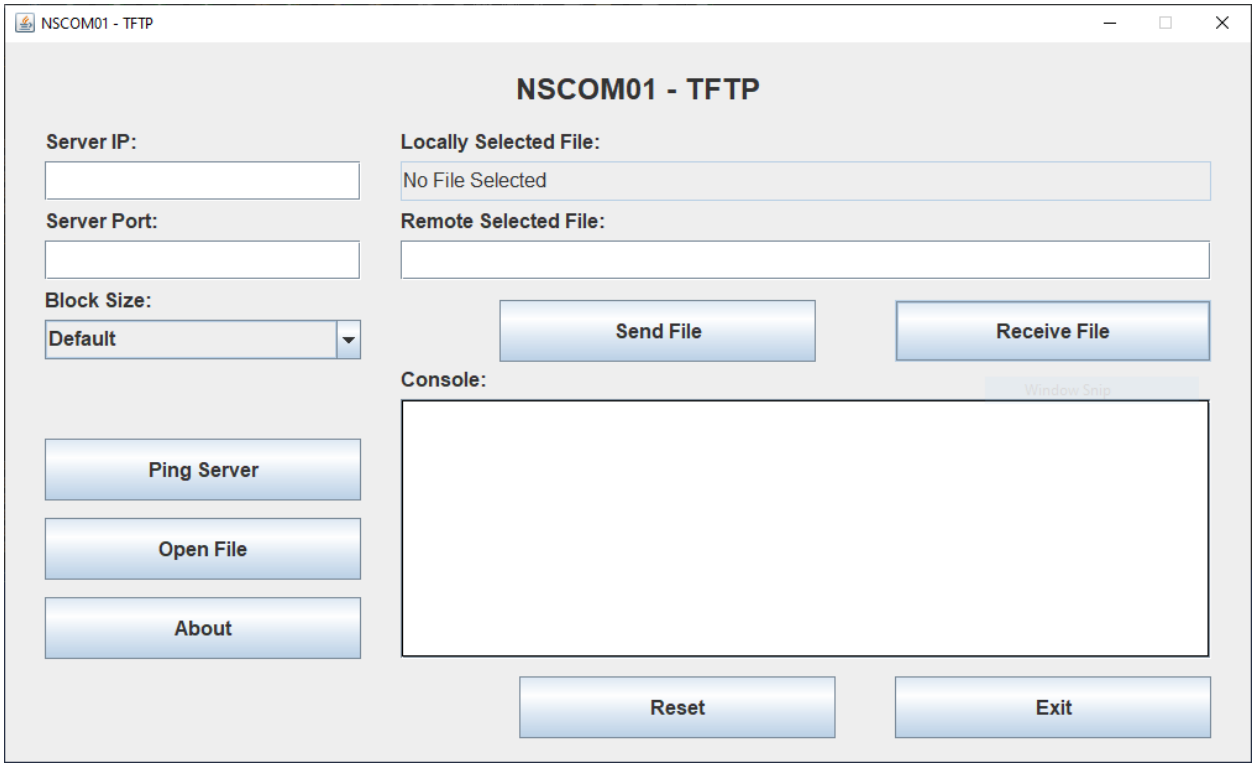
4. getPaddingByteArr()

```
1 private byte[] getPaddingByteArr() {
2   byte[] arr = {getPaddingByte()};
3   return arr;
4 }
```

5. combineBytes()

```
1 private byte[] combineBytes(byte[][] bytes){
2     int size = 0, ctr = 0;
3     for(int i = 0; i < bytes.length; i++)
4         size += bytes[i].length;
5     byte[] combinedBytes = new byte[size];
6     for(byte[] byteArr: bytes) {
7         for(byte b: byteArr) {
8             combinedBytes[ctr] = b;
9             ctr++;
10        }
11    }
12    return combinedBytes;
13 }
```

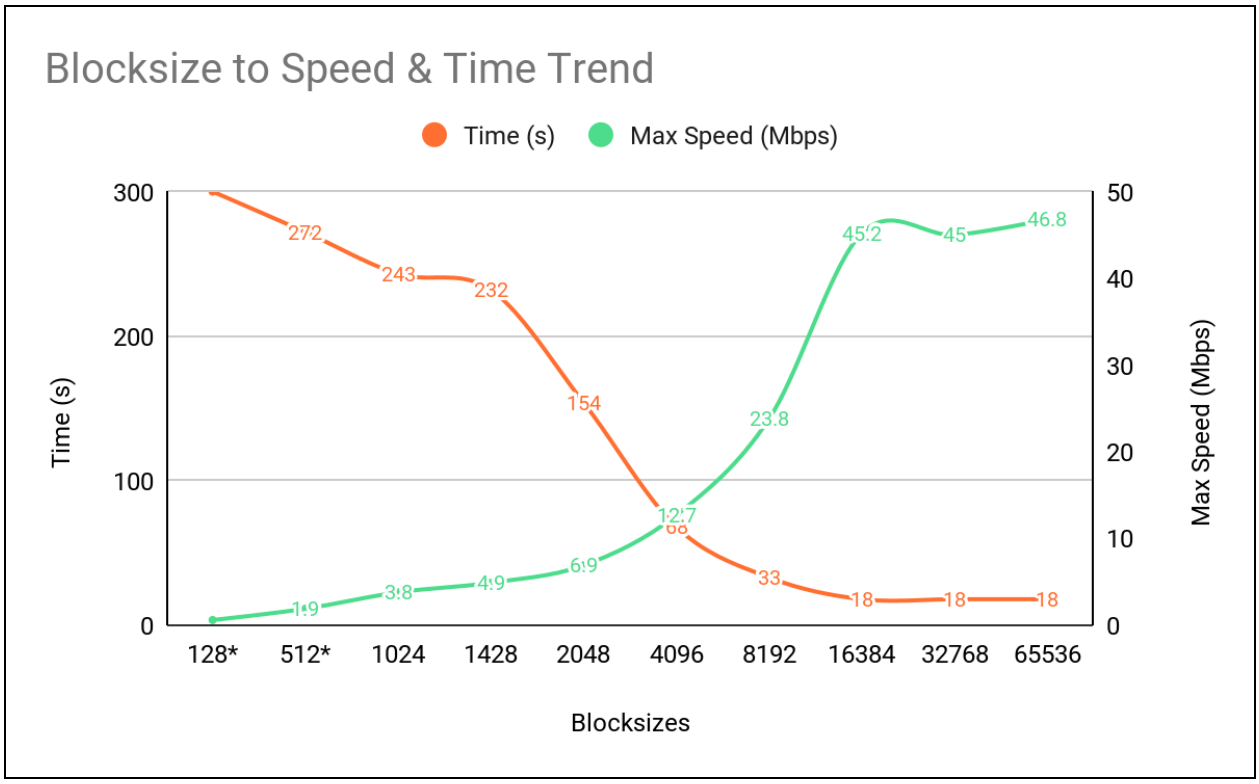
6. GUI Layout
   Some of the GUI's components were referenced from the TFTPd TFTP Client.



Benchmarks

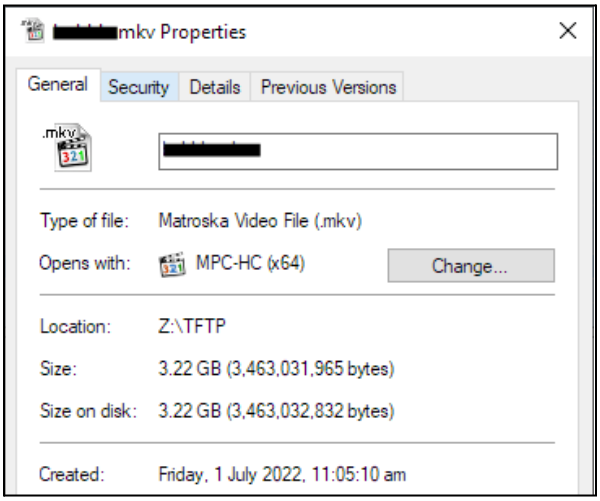
The benchmark was conducted between a client and a server, with the client being connected to WiFi at a max throughput of 250Mbps and the server being connected via wire with a max throughput of 1Gbps. In all, no real bottlenecks are expected in the benchmark process.

Blocksize (bytes)	Time (s)	Max Speed (Mbps)
128*	300	0.6
512*	272	1.9
1024	243	3.8
1428	232	4.9
2048	154	6.9
4096	68	12.7
8192	33	23.8
16384	18	45.2
32768	18	45
65536	18	46.8



Further testing was done on a 3.22GB file, primarily for edge case testing, which resulted in around 854 seconds of transmission (via Scratch test).

```
(2022/07/01 12:22:21) Client.writeToServer(f,opts,vals): ACK Block#:
(2022/07/01 12:22:21) Client.writeToServer(f,opts,vals): blksize adj
(2022/07/01 12:22:21) Client.writeToServer(f,opts,vals): Closing str
(2022/07/01 12:22:21) Client.closeConnection(): Closing connection..
(2022/07/01 12:22:21) Client.closeConnection(): true
(2022/07/01 12:22:21) Client.reset()
Benchmarking successful: 2022/07/01 12:08:07 - 2022/07/01 12:22:21

Testing time elapsed: 854.0seconds
```



Summary of Feature State

Requirements	Status	Note
GUI/CLI	OK	
User Specified Server	OK	
Support for Upload and Download of binary files	OK	.bin files do sometimes fail;
Program can send any file to the server as long as the file is accessible according to OS privileges.	OK	
Program allows user to provide filename to use when saving the downloaded file.	OK	
Timeout for unresponsive server.	Limited	Network-based timeout, not TFTP-based; Defaulted to 3 seconds of no network activity.
Handling duplicate ACK	OK	Implemented for both upload and download. Working but not fully tested.
User prompt for file not found, access violation, and disk full errors.	OK	Implemented and working but not fully tested, especially for full disk error (Error Code 3).
Option to specify transfer blocksize	OK	Server sometimes forces the client to use a specified value which could be expected.
Communicate transfer size to the server when uploading	OK	