Trivial File Transfer Protocol

John Cory-Wright

April 2024

```
TFTP        36 Acknowledgement, Block: 10
TFTP       548 Data Packet, Block: 11
TFTP        36 Acknowledgement, Block: 11
TFTP       548 Data Packet, Block: 12
TFTP        36 Acknowledgement, Block: 12
TFTP       548 Data Packet, Block: 13
TFTP        36 Acknowledgement, Block: 13
TFTP       548 Data Packet, Block: 14
TFTP        36 Acknowledgement, Block: 14
TFTP       548 Data Packet, Block: 15
TFTP        36 Acknowledgement, Block: 15
TFTP       548 Data Packet, Block: 16
TFTP        36 Acknowledgement, Block: 16
TFTP       548 Data Packet, Block: 17
TFTP        36 Acknowledgement, Block: 17
TFTP       548 Data Packet, Block: 18
TFTP        36 Acknowledgement, Block: 18
TFTP       548 Data Packet, Block: 19
TFTP        36 Acknowledgement, Block: 19
TFTP       548 Data Packet, Block: 20
TFTP        36 Acknowledgement, Block: 20
TFTP       548 Data Packet, Block: 21
TFTP        36 Acknowledgement, Block: 21
TFTP       548 Data Packet, Block: 22
TFTP        36 Acknowledgement, Block: 22
TFTP       548 Data Packet, Block: 23
```

Contents

1 Introduction

Trivial File Transfer Protocol (TFTP) is a simple protocol allowing for the transferring of files over a network. Achieved by implementing an acknowledgement based transfer, where each non-terminal packet is acknowledged before the next is sent.

2 Overview

A single TFTP communication begins with a request from a client to either read or write a file from the server. This serves as the handshake of the protocol. Upon accepting the request, the server begins to either send or receive a file in blocks of 512 bytes. These data packets contain the block its self and a block number, each data packet must be then acknowledged by an acknowledgment packet containing that block number. Upon reaching the end of the file, the server or client will send a block of data less than 512 bytes, thus ending the communication. This last packet however, does not need to be acknowledged.

In this implementation, all errors cause termination of the connection. Particularly, only one error is suitably handled wherein the server does not have access to the file the client requested. Secondly, this implementation only involves octet encoded data and does not support ASCII or mail requests.

2.1 Traffic Identifiers (TIDs)

Once a transfer is established each end of the connection chooses their own TID to be used for the entirety of the remaining connection. This should be done randomly to make the possibility of two consecutive TIDs being equal very low. Every packet has both the source and destination TID associated with it and is passed, if used, to the supporting UDP protocol as source and destination ports.

Using Java, this can be implemented easily via making use of the random class and its supported ‘nextInt‘ method, generating a port within the ranges of private/dynamic ports: 49152-65535

```
int port = rand.nextInt((65535-49152)+65535);
socket = new DatagramSocket(rand.nextInt(port));
```

3 Packet Structure

All packets begin with an operation code, or opcode, denoting what kind of packet they are.

opcode	operation
1	Read request (RRQ)
2	Write request (WRQ)
3	Data (DATA)
4	Acknowledgment (ACK)
5	Error (ERROR)

3.1 Read/Write Requests (RRQ/WRQ)

Read and write request packets both follow the same format only differing in the beginning opcode as such:

2 bytes	string	1 byte	string	1 byte
Opcode	Filename	0	Mode	0

Read and write requests (opcode 1 and 2 respectively) follow this. The filename is a sequence of bytes terminated by a 0 byte, in this implementation the mode is always octet and is simply the bytes pertaining to octet followed by a terminating 0 byte.

In Java, these are generated in both client classes by a method called 'createRequest':

```
private byte[] createRequest(byte opcode) {
        byte[] filenameBytes = filename.getBytes();
        int requestSize = 2 + filename.getBytes().length + 1 + octetBytes.length
        byte[] request = new byte[requestSize];
        int index = 0;
        request[index] = 0;
        index++;
        request[index] = opcode;
        index++;
        for(int i = 0; i < filenameBytes.length; i++){
            request[index] = filenameBytes[i];
            index++;
        }
        request[index] = 0;
        index++;
        for(int i = 0; i < octetBytes.length; i++){
            request[index] = octetBytes[i];
            index++;
        }
        request[index] = 0;
        return(request);
}
```

which translates the requested filename given by a user into a sequence of bytes and constructs the request packet according to the RFC spec.

3.2 Data (DATA)

2 bytes	2 bytes	n bytes
Opcode	Block #	Data

All file data is transferred via these DATA packets (opcode 3) containing the data in blocks of 512 bytes maximum, with a data file containing less than 512 bytes denoting the final packet in a single communication. Block numbers begin with 1 and increase by one per each new block of data. Data packets can be generated by the following Java method 'getDataPacket':

```
private byte[] getDataPacket(int blockNumber) {

    byte[] fileData;
    byte[] blockBuf = getBlockNumberBytes(blockNumber);

    try {
        fileData = Files.readAllBytes(Paths.get(fileDirectory+filename));
        if (blockNumber == ((fileData.length / 512) + 2)){
            System.out.println("Last packet in file");
            return null;
        }
        int to;
        if (blockNumber > fileData.length / 512) {
            to = (512 * (blockNumber - 1)) + (fileData.length % 512);
        } else {
            to = 512 * blockNumber;
        }
        int from = (blockNumber - 1) * 512
        byte[] segment = Arrays.copyOfRange(fileData, from , to);
        byte[] dataPacket = new byte[4 + segment.length];
        dataPacket[0] = 0;
        dataPacket[1] = 3;
        dataPacket[2] = blockBuf[0];
        dataPacket[3] = blockBuf[1];
        System.arraycopy(segment, 0, dataPacket, 4, segment.length);

        return dataPacket;
    } catch (IOException e) {
        System.out.println("File does not exist.");
        e.printStackTrace();
        return null;
}
```

This makes use of modular arithmetic to find the particular segment of a file's bytes to be sent according to the block number requested. In this case returning null if a file does not exist. The block number, as an integer, needs to be translated into 2 bytes before being added to the packet.

3.3 Acknowledgment (ACK)

2 bytes	2 bytes
Opcode	Block #

All data packets must be acknowledged, this is done so via these ACK packets (opcode 4) with block numbers equalling the most recent data packet that was successfully received by either client or server. Acknowledgement packets have been generated in Java by the following method 'getAckPacket()'

```
private byte[] getAckPacket(int blockNumber) {

    byte[] ackBuf = new byte[4];
    byte[] blockBuf = getBlockNumberBytes(blockNumber);
    ackBuf[0] = 0;
    ackBuf[1] = (byte) 4;
    ackBuf[2] = blockBuf[0];
    ackBuf[3] = blockBuf[1];

    return ackBuf;
}
```

3.4 Error (ERROR)

2 bytes	2 bytes	string	1 bytes
Opcode	ErrorCode	ErrMsg	0

Error packets (opcode 5) are always sent to handle an error occurring and cause the side sending them to terminate communication immediately. In this particular implementation only file not found errors are required to be implemented, denoted by the error code 1 and the error message "File not found". These are generated in Java by the following 'GetErrorPacket()' method

```
private byte[] getErrorPacket(){

    byte[] errorMsg = ("File not found");
    byte[] errorBuf = new byte[5 + errorMsg.length];
    errorBuf[0] = 0;
    errorBuf[1] = 5;
    errorBuf[2] = 0;
    errorBuf[3] = 1;
    System.arraycopy(errorMsg, 0, errorBuf, 4, errorMsg.length);
    errorBuf[errorBuf.length-1] = 0;
    return errorBuf;
}
```

Which translates the previous error message into bytes and constructs the packet as per the RFC specification.

4 Implementation

both server and client essentially have the same role, needing to receive data packets, handle them, and then respond with corresponding acknowledgements. However, the client is the only one sending request packets and the server is the only one receiving request packets. Due to this, apart from request packets, the above implementations for packet creation is the same either side. However, the order in which either side handles packets differs as a client must send in order to begin receiving and a server must receive before opening an independent thread, choosing a TID, and responding to initial communications.

4.1 Client and Server Response

Both sides have a central switch statement deciding on which response to send back in the method 'getResponseHeader()':

```
private byte[] getResponseHeader(DatagramPacket packet) {

    byte opcode = packet.getData()[1];
    switch (opcode) {
        case OP_WRQ:
        case OP_RRQ:
            return handleRequest(packet);
        case OP_DAT:
            return handleData(packet);
        case OP_ACK:
            return handleAck(packet);
        case OP_ERR:
        default:
            return null;
    }

}
```

Using an opcode gleaned from the received packet, a corresponding response header is generated according to TFTP's RFC spec. In the case of the client the need of a WRQ/RRQ case is omitted as it is superfluous.

'handleRequest()' generates and returns the corresponding ACK or DATA packets in response to a WRQ or RRQ respsectively, generated as per the methods defined in sections 3.2 and 3.3. Only used in the Server implementation of this protocol:

```
private byte[] handleRequest(DatagramPacket packet) {

    byte opcode = packet.getData()[1];
    if (opcode == 1){
        return(getDataPacket(1));
    } else {
```

```
        return getAck(0);
    }
}
```

'handleData()' appends data packet to a file data buffer, generates and returns an acknowledgement packet as stated in section 3.3. If final data packet is received, it writes the full file buffer to the filename provided in initial transmission, overwriting if required.:

```
private byte[] handleData(DatagramPacket packet) {

    byte[] data = Arrays.copyOfRange(packet.getData(), 4, packet.getLength());
    byte[] oldBuf = fileBuf;
    byte[] newBuf = Arrays.copyOf(oldBuf, oldBuf.length + data.length);
    System.arraycopy(data, 0, newBuf, oldBuf.length, data.length);
    if (packet.getLength() < 516) {
        readToFile();
    }

    return getAck(getBlockNumber(packet.getData()));
}
```

'handleAck()' generates and returns next data packet as stated above in section 3.2:

```
private byte[] handleAck(DatagramPacket packet) {
    int blockNumber = getBlockNumber(packet.getData());
    return getDataPacket(blockNumber + 1);
}
```

4.2 Handling Multiple Clients

In order to acheive both the valid TIDs and the ability to handle multiple concurrent users, a thread pool has been used in the server. Each client connected is assigned an idle thread from the thread pool generated in the server's constructor. With the servers main running loop as such:

```
public void run() {
    byte[] buf;
    DatagramPacket packet;
    try {
        while (serverRunning) {
            buf = new byte[516];
            packet = new DatagramPacket(buf, buf.length);
            serverSocket.receive(packet);
            if (packet.getData()[1] == 1 || packet.getData()[1] == 2) {
                TFTPServerThread serverThread = new TFTPServerThread(packet);
                pool.execute(serverThread);
```

```
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
}
```

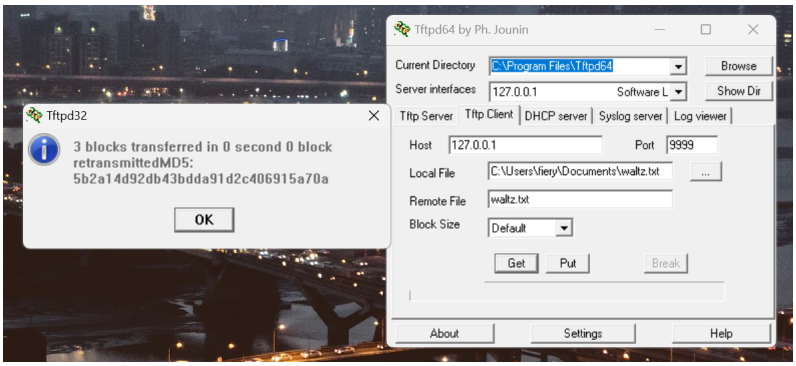
each new thread binds its datagram socket to a random port number from 49152-65535 and proceeds to handle the entire TFTP connection as stated in 4.1.

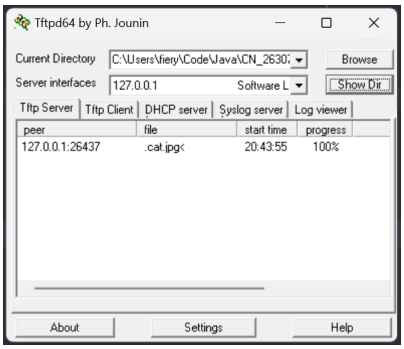
4.3 Cross-Compatibility

Throughout the project, a third party software called 'Tftpd64' has been used to debug and test inter-operability of the implemented TFTP protocol. Alongside this Wireshark can display the validity in its successful identification of the packets sent as TFTP:

```
229 14.283359    127.0.0.1       127.0.0.1       TFTP      50 Write Request, File: waltz.txt, Transfer type: octet
230 14.285625    127.0.0.1       127.0.0.1       TFTP      36 Acknowledgement, Block: 0
231 14.308759    127.0.0.1       127.0.0.1       TFTP     548 Data Packet, Block: 1
232 14.308990    127.0.0.1       127.0.0.1       TFTP      36 Acknowledgement, Block: 1
233 14.309348    127.0.0.1       127.0.0.1       TFTP     548 Data Packet, Block: 2
234 14.309446    127.0.0.1       127.0.0.1       TFTP      36 Acknowledgement, Block: 2
235 14.309790    127.0.0.1       127.0.0.1       TFTP     224 Data Packet, Block: 3 (last)
236 14.311085    127.0.0.1       127.0.0.1       TFTP      36 Acknowledgement, Block: 3
```

Furthermore Tftpd64 can interface with the implementation of both client and server over UDP succesfully as such:




Tftpd64 by Ph. Jounin
Current Directory
C:\Users\fiery\Code\Java\CN_2630
Browse
Server interfaces
127.0.0.1
Software L
Show Dir
Tftp Server
Tftp Client
DHCP server
Syslog server
Log viewer
peer
file
start time
progress
127.0.0.1:26437
.cat.jpg<
20:43:55
100%
About
Settings
Help