Trivial File Transfer Protocol

Yogesh Jagadeesan
yj6026@rit.edu

Dler Ahmad
dha3142@rit.edu

1. INTRODUCTION

There are a wide variety of file transfer protocols whose focus is, as the name suggests, transfer of files from one host to another. With the increasing demands in the need to transfer files among nearby and faraway hosts, irrespective of their operating environments, a variety of file transfer protocols have surfaced over the years. The FTP for instance, works on a TCP based network, transfering files from one host to another. It employs a client-server architecture and also separate data and control channels in order to securely transfer files among hosts through means of an authentication. The FTPS, standing for FTP over SSL adds an extra layer of security to the existing FTP, employing security at the data and control channels as well in order to securely transfer files. These protocols support different transfer modes such as ASCII, EBCDIC and the like which may be mutually agreed upon by two or more of the hosts in operation and hence might receive blocks of data in that format.

One such very simple protocol is the TFTP, standing for Trivial File Transfer Protocol. The communication always happens between any two hosts in this case who mutually agree upon a particular format in which they transfer and receive files. Actual transfers happen in chunks of 512 bytes until all the chunks are eventually sent. No failure handling mechanisms are employed, in the sense if any chunk is lost, then the resulting file will be incomplete and the transfer would have to happen from the beginning. Security was not taken into account either except, possibly, at the initial connection establishment step.

2. PROBLEM

The problem here we are trying to solve is based on the working of a simple TFTP. The specifications of this protocol are mentioned in RFC 1350[1]. The operations supported by are put(to transfer files to remote server), get(acquire files from the remote server), mode(determines the mode of transfer which can vary between binary and ascii) and ?(which just displays usage information). According to the afore mentioned RFC, 'put' and 'get' commands work by transfering chunks of fixed sized data at a time. The type of the data transferred is specified by the mode. Binary mode(octet) can be used to transfer text files, multimedia files and the like while ASCII mode is restricted to transfer of text files. Attempting to use ASCII mode on multimedia files would result in a corrupt file. Also, the port with which the transfer takes place is decided by the TFTP server. The initial connection by a TFTP client is made to port 69 and the first response from the TFTP server holds the port number assigned to that specific client. From then on, the client is restricted to use that returned port. Attempt to use any other port would result in an "Illegal TFTP operation" message from the server.

Glados supports TFTP file transfers as well. Our goal is to mimic the working of a TFTP client following precise specifications from the RFC using Perl. This would mean that specific, accurately sized packets containing appropriate opcodes should be used for appropriate operations. An initial connection establishment to the TFTP server is made by either sending an RRQ(Read Request) packet or the WRQ(Write Request) packet. Assume we are mimicking the 'get' TFTP command. The initial packet contents are as follows. The first two bytes contain the opcode '01' converted to bytes which corresponds to RRQ. This should be followed by the full path of the file including the name of the file converted to bytes. A one byte separator containing '0' follows the file name followed by the mode of operation(either the "octet" or "netascii" string converted to bytes). The end of the packet is another '0' occupying one byte. This packet initiates data transfer from server to client and is sent to port 69 of 'glados.cs.rit.edu'. If this packet is constructed properly, the first response holds the first 512 bytes of the requested file containing the opcode '03' (indicating data occupying 2 bytes), the block number of this chunk(occupying 2 bytes) followed by the data itself(varying from 0 to 512 bytes). The client should have to acknowledge this data packet by sending a packet that contains the opcode '04' and the block number we are acknowledging, each occupying two bytes. This acknowledgement would result in the server sending the next 512 bytes of data which is again acknowledged and the sequence continues until a data packet is received which is 0 to 511 bytes long marking the end of the file. This last data packet should be acknowledged as well by the client in order for the connection to terminate normally.

The file should be constructed on the client side using each of the data packet received from the server. Also, packets with opcode '05' indicate an error along with the error message which should be extracted and displayed as well should such a packet arrive.

Similar procedure is followed for the 'put' operation as well. A WRQ packet from the client initiates the operation. In this case, the client sends packets of 512 bytes of the data to be placed on the server and for each of these packets, the server sends an ACK packet to indicate successful receipt.

3. METHODOLOGY

As previously mentioned, Perl is used for implementation of this TFTP. The modes of transfer are stored as strings according to what the user needs. It should be remembered that ASCII mode is only used for transfer of text files while binary mode can be used for any type of files.

The 'get' operation, as mentioned, requires an initial RRQ packet. The file name to be received is got from the user from STDIN. The communication to the glados server is done using UDP. Appropriate UDP socket is created with 69 as the initial port number and the address of 'glados.cs.rit.edu' as the ip address. The construction of the packet is done using the 'pack'[2] method in Perl. The opcodes are packed using the 'n' as the parameter which indicates an unsigned short in big-endian order. The filename, which is in the form of a string is packed using the 'a' parameter which indicates arbitrary binary data. The final packed content is transferred using the UDP socket using the 'send' method. The initial response is received using the 'recv' method. The packet is expected to have opcodes and binary data as well so 'n' and 'a' are used respectively to decode(using 'unpack') the actual opcode and the data received. If this initial response doesn't indicate any error(can be found using the opcode received) and does indeed contain data, a file stream is opened in the current directory with the same name as the user's input and the initial data from the packet is written to this stream. Also, port number '69' is overwritten to the port number from the response and all successive 'send' and 'recv' happen via this new port. Each data packet contains a block number as well which is incorporated in the acknowledgement packet to be sent from the client. The rest of the sequence involves alternating between receiving successive data packet(and thereby writing it to the file stream) and sending acknowledgement packet for the received data packet. Finally, when a data packet is received that is less than 516 bytes in length (which means less than 512 bytes of data excluding the opcode and block number), it marks the last chunk of the file. The file stream is closed after writing this last chunk and is acknowledged as well thus properly terminating the connection.

The 'put' operates in a similar fashion. After receiving the the local and remote filename from the user as string, the initial WRQ packet is sent containing the remote filename. The initial port number to which this packet is sent is '69'. If this initial response is an acknowledgement packet(indicated by the opcode) containing block number 0, a file stream is opened and the port number is overwritten to the new port received from this packet. Thereby the sequence involves reading the file in chunks of 512 bytes of data(using the 'read' method), constructing data packets using the 'pack' method and sending it over to the server through the UDP socket using the 'send' method. The response is received using the 'recv' method which would be acknowledgement to the last sent chunk of data which is 'unpack'ed and deciphered. The sequence follows until all of the file is read in which case the connection is closed and the file creation is complete at the server.

4. CONCLUSIONS AND FUTURE WORK

This implementation has resulted in seamless transfer and receipt of files from the glados server. However, due to restrictions, the transfers and receipt were restricted to only the "/local/sandbox" directory in the remote server. Binary mode is good enough to transfer any type of file, from text to multimedia while ASCII is restricted to proper transfer of just text files. Also, the server assigns a dedicated port for each transfer until its completion. The data is not encrypted in any way during these transfers but is only split into chunks of 512 bytes.

If given more time, a couple of features can be added to the current implementation. The users can be allowed to request for multiple files at the same time and each transfer can possibly happen in a separate thread. Also once a request for a transfer is made, the users may not have to wait until its complete before he can initiate another if needed. This can be done by delegating the transfer operation to a separate thread and user interaction to another. This can be especially useful if transfer of a large file is taking place so that the user is not blocked until it is complete.

Failure handling can also be employed. If a transfer fails or if an error has occurred, the protocol doesn't allow resume of transfer from the point where it was broken. However, it can still be reinitiated. A mechanism can be introduced in which a failure is not immediately notified to the user but 2 or 3 reinitiation attempts are made before doing the same.

A load handling mechanism can also be introduced. If a large number of simultaneous requests are made, a queue can be introduced and its made sure that requests are handled in batches from the queue instead of attempting to handle all the transfer requests at the same time.

For security reasons, a handshake mechanism can be introduced in order to ensure that the client is indeed talking to the actual server and not to any intruder(man in the middle). This may be done by purposely sending a few invalid requests and checking to see if the server is responding with the appropriate opcode. For instance, a non-existent file can be requested on purpose in order to check if the opcode returned is indeed 01 which corresponds to "File not found". Assuming the remote server is indeed secure, there's no way for the intruder to know if the file exists or not on the actual server.

5. REFERENCES

[1] K. Sollins, *https://www.ietf.org/rfc/rfc1350.txt.* Network Working Group, 1992.

[2] *http://perldoc.perl.org/functions/pack.html* Pack and Unpack in Perl.