**Trivial File Transfer Protocol**

Specification

Mainly based on RFC 1350

*-- Tony Rafanomezantsoa --*

**Introduction**

TFTP is a simple application protocol used to transfer files. It is implemented on top of the Internet User Datagram Protocol. It is used to read files from a remote server *(Download)* or to write files to a remote server *(Upload)*.

**Protocol Details**

***+ Initial Connection Protocol***

Any file transfer is initiated by a client process that either reads files from a remote server (Download) or writes files to a remote server (Upload). On the other side, remote server listen and serves each incoming request. The IP address of the remote server is required by the client process to allow the file transfer (read or write).

To transfer files, client process initiates a request (either WRQ to write onto a foreign file system or RRQ to read from it), and receive a positive reply, from remote server, which is an acknowledgment packet for write or the first data packet for read. If the reply is an error packet, then the request has been denied.

Client and server create a connection and choose a TID for themselves to allow file tansfer. The transfer identifiers (TID) is a number between 0 and 65,535, which will serve as the port number for the client and server sockets. For both end of the connection (client and server), TID should be randomly chosen.

As an example, the following shows the intitial steps used to establish a connection in order to write a file:

**1.** *Host A sends a "WRQ" to host B with source = A's TID (randomly chosen) and destination equal to 69 by default (TFTP server default port number).*

**2.** *Host B sends a "ACK" (with block number = 0) to host A with source = B's TID (randomly chosen) and destination = A's TID.*

At this point the connection has been established between client and server processes, so the first data packet can be transferred. The two chosen TID's are then used for the remainder of the transfer.

During the file transfer, client process stores the server TID (from the initial connection procedure) in its memory to ensure the connection integrity. The reason is : It is possible that, the client process receive data from a server that it did not request (which may caused by network issues). So, when response arrives to the client process, It verifies if the remote server TID corresponds to the initial server TID stored in its memory, and if it corresponds, data transmission and connection continue, otherwise, the client process sends an error packet (ERROR) to the remote server, which lead the end of the connection.

**• Initial Connection Protocol –- Implementation & Algorithm**

As described before, any file transfer is initiated by a client process that either reads files from a remote server (Download) or writes files to a remote server (Upload).

*Client process description:*

A client process is represented by a Command Line Interface program. To transfer file, either read or write, the client process requires exactly 3 arguments : an *action*, a *filename*, an *IP address* of the remote server.

*client\_process\_program [ACTION] [FILENAME] [REMOTE SERVER IP ADDRESS]*

*\* ACTION:* This indicates the action of the client process and should be a string value, with either « read » or « write » value. (the string value is case insensitive)

\* *FILENAME:* This is a string value that indicates the name or path of the file to read from or to write to the remote server. If the *ACTION* value is equal to « read », *FILENAME* is relative to the folder where the server process (binary program) is located on the remote end-system. Otherwise, if the *ACTION* value is equal to « write », *FILENAME* is relative to the working directory where the client process is run.

\* *REMOTE SERVER IP ADDRESS:* This indicates the network interface address of the end-system where the server process is run. It can be an Ipv4 or Ipv6 address.

*NOTE : Mode in RRQ and WRQ*

Currently, this implementation of TFTP supports only the « octet » mode, for both the client and server. So,

In the client-side process :

* If the *ACTION* is equal to read, the client process will always initialize a RRQ packet to the server with the mode specified as « octet ».
* If the ACTION is equal to write, the client process will always initialize a WRQ packet to the server with the mode specified as « octet », and will handle all data in a sequence of bytes for writing / sending.

In the server-side process :

* The server process will verify the mode (for RRQ or WRQ), and if it is not set to  « octet », an error will be generated. And the server will handle all data in a sequence of bytes

*Client process – Initial Connection Protocol Implementation & Algorithm:*

First of all, to transfer file, either read or write, the client process needs to collect all arguments : the *ACTION*, the *FILENAME*, the *REMOTE SERVER IP ADDRESS*.

Next, those values (all arguments) should be parsed and verified correctly. Errors will be generated for invalid values.

The client process then creates a UDP socket (wich is the client socket). For the socket address, the IP address should be set to 0.0.0.0 (which represents the local machine), and the port number should be randomly chosen from the range of 0 to 65,535 (which represents the client TID as described before).

After that, the client process analyze the value of the *ACTION* :

*\** If *ACTION* is equal to *« read » (case insensitive) :*

1. Then, the client process initiates a « RRQ » packet (Read Request) to the remote server process.The server’s UDP socket address is the combination of the *REMOTE SERVER IP ADDRESS* and port number 69 which is the default port number for TFTP servers. The structure of the « RRQ » packet is the following :

2 bytes string 1 byte string 1 byte

------------------------------------------------

| Opcode | Filename | 0 | Mode | 0 |

------------------------------------------------

*RRQ packet structure*

For « RRQ » packet, the *opcode* is a 16-bit (2-byte) unsigned integer that has a value of 1. Filename will correspond to *FILENAME* in bytes and mode will correspond to « octet » (case insensitive) in bytes*.*

1. After the « RRQ » packet is sent, the client process waits and parses the response from the remote server, which should be the first data packet to read, with a block number equal to 1 (described in RFC 1350). If the packet is not a « DATA » packet or invalid « DATA »  packet, an « ERR » packet will be sent to the remote server. Bellow is the structure of the « DATA » packet :

2 bytes 2 bytes n bytes

----------------------------------

| Opcode | Block # | Data |

----------------------------------

*DATA packet structure*

For « DATA » packet, the *opcode* is a 16-bit (2-byte) unsigned integer that has a value of 3. Block is a 16-bit (2-byte) unsigned integer which represents the block number and data represents the data in sequence of bytes (from 0 to 512 bytes).

At this point, the connection is etablished. That connection is then used for the remainder of the transfer.

The remainder data transfer and storage will be discussed later.

*\** If *ACTION* is equal to *« write » (case insensitive) :*

1. The client process attempts to open the files for writing to the server. If an error occurs, the program will generate an error message and exit.
2. Then, the client process initiates a « WRQ » packet (Write Request) to the remote server process.The server’s UDP socket address is the combination of the *REMOTE SERVER IP ADDRESS* and port number 69 which is the default port number for TFTP servers. The structure of the « WRQ » packet is the following :

2 bytes string 1 byte string 1 byte

------------------------------------------------

| Opcode | Filename | 0 | Mode | 0 |

------------------------------------------------

*WRQ packet structure*

For « WRQ » packet, the *opcode* is a 16-bit (2-byte) unsigned integer that has a value of 2. Filename will correspond to *FILENAME* in bytes and mode will correspond to « octet » (case insensitive)in bytes*.*

1. After the « WRQ » packet is sent, the client process waits and parses the response from the remote server, which should be an acknowledgment packet for write, with a block number equal to 0 (described in RFC 1350). If the packet is not a « ACK » packet or invalid « ACK »  packet, an « ERR » packet will be sent to the remote server. Bellow is the structure of the « ACK » packet :

2 bytes 2 bytes

---------------------

| Opcode | Block # |

---------------------

ACK packet

For « ACK » packet, the *opcode* is a 16-bit (2-byte) unsigned integer that has a value of 4. Block is a 16-bit (2-byte) unsigned integer which represents the block number.

At this point, the connection is etablished. That connection is then used for the remainder of the transfer.

The remainder data transfer will be discussed later.

*Server process – Initial Connection Protocol Implementation & Algorithm:*

In order to etablish a connection with a TFTP client, the server listens on the socket address 0.0.0.0:69 (which is the default socket address for a TFTP server) and processes each incoming request to create a connection.

When a request arrives on that default address, which should be a request packet (either RRQ or WRQ), the server parses and analyzes the packet. If the packet is invalid, the server will generate an error « ERR » packet.

*Note : As previously mentioned, the current implementation of the server supports only the « octet » mode, especially for read action. If a different mode is requested, the server will generate an error « ERR » packet.*

Once the request has been parsed, the packet is analyzed:

*\** If the requestis a « *read request packet* » (RRQ)*:*

1. The server process creates a new UDP socket to create a connection with the client process. For the socket address, the IP address should be set to 0.0.0.0 (which represents the local machine), and the port number should be randomly chosen from the range of 0 to 65,535 (which represents the server TID as described before), and that connection is then used for the remainder of the transfer.
2. The server checks wether the mode is set to « octet » (case insensitive). If it is not, the server will generate an « ERR » packet.
3. The server process then opens and reads the file specified in RRQ. If any errors occur during file access, the server will generate an error « ERR » packet. (As previously mentionned, the file path or name is relative to the server binary path)
4. Then, the server process initalize a variable in its memory to store the current block number which is initially set to 1.
5. Then, the server replies to the client process with a 'DATA' packet, which contains a block of data with a size ranging from 1 to 512 bytes and with a block number equal to the block number in the memory, which is currently equal to 1.

*\** If the requestis a « *write request packet* » (WRQ)*:*

1. The server process creates a new UDP socket to create a connection with the client process. For the socket address, the IP address should be set to 0.0.0.0 (which represents the local machine), and the port number should be randomly chosen from the range of 0 to 65,535 (which represents the server TID as described before), and that connection is then used for the remainder of the transfer.
2. The server checks wether the mode is set to « octet » (case insensitive). If it is not, the server will generate an « ERR » packet
3. Next, the server creates a variable to store the last block number which is initially set to 0.
4. Then, the server replies to the client process with a 'ACK' packet, with block number equal to the last block number in the memory, which is currently equal to 0.

**• File Transmission –- Implementation & Algorithm**

*Client process – File Transmission Implementation & Algorithm:*

*\** For « read » *ACTION: [Client process receives data and sends acknowledgement]*

1. After the client process receives and verifies the first « DATA » packet for reading from the server, it creates a new file to store all incoming data. The filename is equivalent to *FILENAME*’s basename, and the file path to create the file is relative to the working directory where the client process is running.
2. Then, the client process reads all data from the « DATA » packet and appends it to the file. If any error occurs, an « ERR » packet will be generated and the program will exit.
3. Next, the client process sends a « ACK » packet to the server with block number equal to 1 (which corresponds to the current block number in the « DATA » packet).
4. After that, the client process stores the last block number in its memory. (which is currently equal to 1)
5. The client process then verifies if the appended data in the file is less than 512 bytes. If it is, the transmission concludes and the client process exits. If not, the file transmission continues:

* The client process waits and receives the next « DATA » packet.
* The client process verifies the server TID
* The client process parses and verifies the « DATA » packet.
* The client process reads and appends the data to the file (the file which is previously created)
* The client process sends a « ACK » packet to the server. (The block number corresponds to the current block number in the « DATA » packet)
* The client process update the block number in its memory with the current block number.
* The client process then verifies if the appended data in the file is less than 512 bytes. If it is, the transmission concludes and the client process exits. If not, the file transmission continues. That process is repeated for the remaining data transmission.

*\** For « write » *ACTION: [Client process sends data and receives acknowledgement]*

1. After the client process receives and verifies the first « ACK » packet for writing from the server (with block number equal to 0), the client process initialize a variable in its memory to store the current block number which is initally set to 1.
2. The client process then reads the data in the previously opened file in chuncks of 512 bytes. If any error occurs, an « ERR » packet will be generated and the program will exit.
3. Then the client process build a « DATA » packet with block number equal to the block number in the memory and with the chunks of data, and sends the « DATA » packet to the server.
4. The client process waits and receives a « ACK » packet from the server. The block number should be verified. (The block number should correspond to the block number stored in memory)
5. The client process then verifies if the data read from the file is less than 512 bytes. If it is, the transmission concludes and the client process exits. If not, the client process increments the block number in its memory and the process goes back to the step [2.].

*Server process – File Transmission Implementation & Algorithm:*

*\** For « read » *ACTION (RRQ): [Server process sends data and receives acknowledgement]*

1. After the server process replies the client with the first data packet, It waits an acknowledgement packet from the client process.
2. The « ACK » packet should be verified and the block number in the « ACK » packet should be equal to the current block number in the memory. If any error occurs, an « ERR » packet will be generated.
3. The server process then verify if the data used to reply the client process is less than 512 bytes. If It is, the transmission concludes and the server process waits and processes the next request. If not, the server process increments the current block number in the memory and sends next data packet, which is the same as sending the first data packet. That process is repeated until the file transmission is finished.

*\** For « write » *ACTION (WRQ): [Server process receives data and sends acknowledgement]*

1. After the server process replies the client with the first acknowledgement packet, then It creates a new file to store all incoming data.
2. The server process then waits, receives and parses the « DATA » packet from the client.

The « DATA » packet should be verified. If any error occurs, an « ERR » packet will be generated. (This process requires a buffer sized to 516 bytes)

1. Then, the server process retrieves the data in the « DATA » packet and appends it to the file, which is previously created.
2. Next, the server process sends an « ACK » packet to the client, with a block number equal to the block number in the « DATA » packet.
3. The server process then verify if the data used to reply the client process is less than 512 bytes. If It is, the transmission concludes and the server process waits and processes the next request. If not, the server process sets the last block number in the memory to the current block number in the « DATA » packet, and waits for the next « DATA » packet (go back to the step [2.]). That process is repeated until the file transmission is finished.