**TRUSTED TRANSPORT PROTOCOL (TTP)**

**&**

**740 FILE TRANSFER PROTOCOL (740FTP)**

**DESIGN DOCUMENT**

**By Ramya Balaraman(rbalaram)**

**& Venkatesh Sriram (vsriram)**

**CONTENTS**

INSTRUCTIONS TO RUN THE PROGRAM 2

EXPECTED OUTPUT (without tests) 4

Example 1: Without fragmentation/reassembly (File size in this example: 411 bytes) 4

Example 2: With fragmentation & reassembly (File size in this example: 40,280 bytes) 6

PROTOCOL SPECIFICATIONS 13

ARCHITECTURE 14

CLASS DIAGRAM 15

PROGRAM FLOW 16

FEATURES IMPLEMENTED CHECKLIST 18

# INSTRUCTIONS TO RUN THE PROGRAM

1. Configure Run Configuration arguments of FTPClient & FTPServer programs to accept window size and retransmission timer interval (in milliseconds) as first and second arguments respectively. Currently these parameters have been set to 20 and 7000 respectively.
2. FTPServer has been hardcoded to listen on Port 2221 for testing purposes as port 21 was already bound and could not be used. Modify this port number in FTPServer.java if required.
3. FTPClient has been hardcoded with source port as 2000. Modify this port number if required in FTPClient.java. Also modify the destination IP if it is not localhost.
4. Run FTPServer.java
5. Run **multiple** FTPClient.java on different machines. (Note: Modify the Destination IP if running the client from a different machine) You will be prompted to enter the file name on the console window. Enter any file name from the following:
   1. Sample1.txt
   2. Sample2.rtf
   3. DesignDoc.docx
   4. DesignDoc.pdf
   5. Smallest.gif
   6. Small.gif
   7. Medium.jpg
   8. ClassDiagram.png

Note: Large files like DesignDoc.docx, DesignDoc.pdf take a considerable time to transfer especially with our version of DatagramService (which includes Test Cases). Please be patient ☺

1. The minimum and maximum delay introduced in the tests is 7.5 and 15 seconds respectively. We have considered the retransmission timer interval to be less than 15seconds during our testing. Please change delay according to timer interval if needed.
2. Note: At times, due to delay and dropped packets introduced by the test cases, the client program repeatedly sends the same acknowledgement (of the last packet correctly received) and it appears like the program is stuck. But please be patient, it will proceed to completion.

Test with smaller files like Sample1.txt or Small.gif in this case to test the logic.

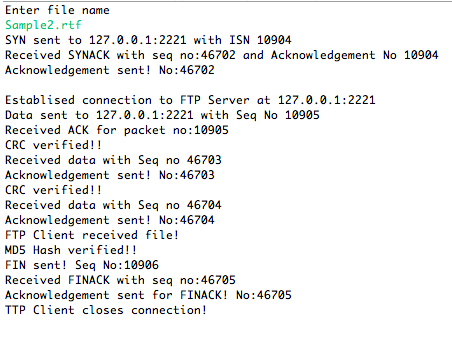
# EXPECTED OUTPUT (without tests)

## Example 1: Without fragmentation/reassembly (File size in this example: 411 bytes)

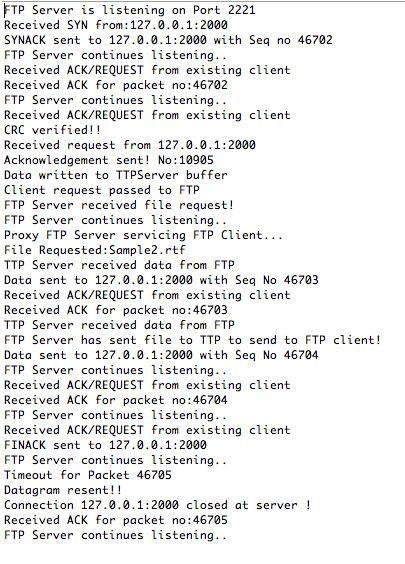
**[Note: Max data bytes in one packet is 1281 + 9 bytes header]**

The file requested will be received in the **ClientFiles** folder.

**FTP Client:**

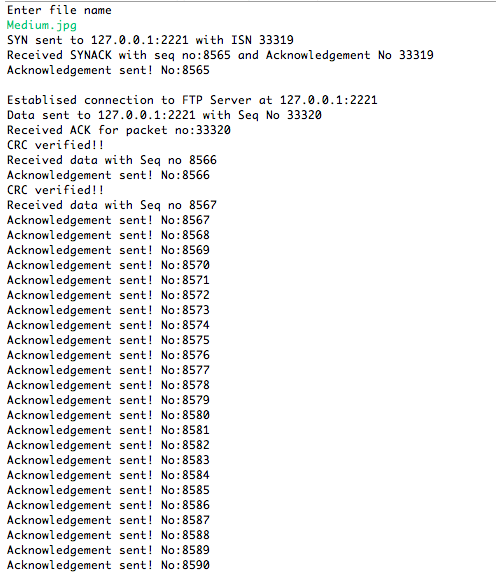
****

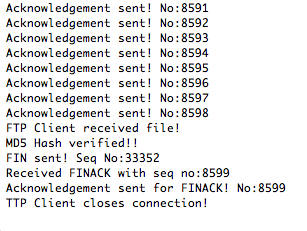
**FTP Server:**

****

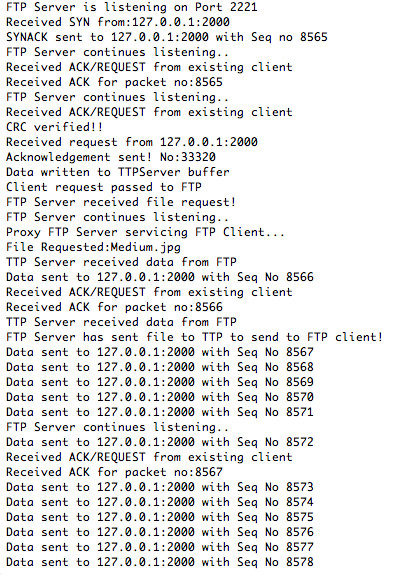
## Example 2: With fragmentation & reassembly (File size in this example: 40,280 bytes)

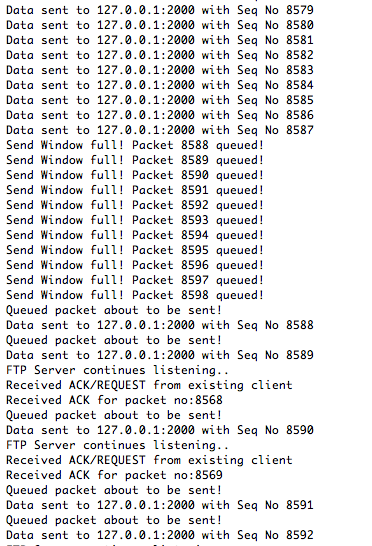
**FTP Client:**

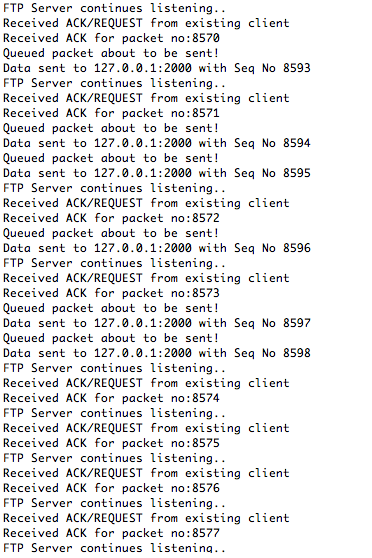
****

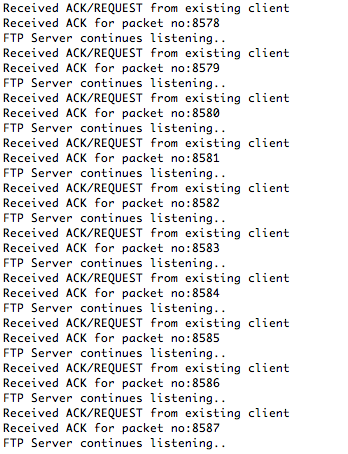
****

**FTP Server:**

****

****

****

****

# 

# PROTOCOL SPECIFICATIONS

1. Initial Sequence Number (ISN):

Sequence number is of data type int and can be a maximum of 4 bytes. The initial sequence number is a randomly generated value from 0 to 65535.

1. Connection Setup: 3-way handshake.

TTP Client sends SYN, TTP Server responds with SYN ACK, TTP Client sends ACK

1. Payload contains 9 bytes of header

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Header Flags

Acknowledgement No

Sequence Number

1. Sequence Number is assigned packet-wise, and not byte-wise as in TCP.
2. The Acknowledgement number denotes the sequence number of the last received packet, and **not** the byte value of the next byte expected, as in TCP.
3. The header structure of the payload is the following-

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N/A | N/A | N/A | FINACKACK | EOF | SYN | ACK | FIN |

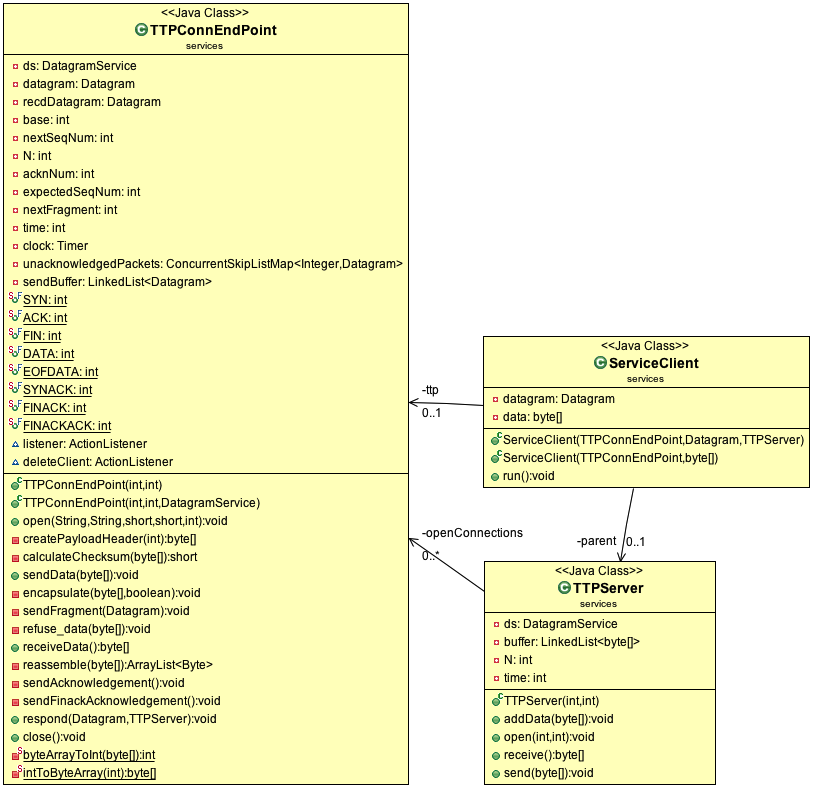
1. Maximum number of bytes in one packet is 1281 + 9 bytes for the header.
2. Fragmentation and reassembly is done using the EOF flag, which signifies the End of File marker.
3. The checksum is calculated over the payload using the same method as the UDP checksum. It is verified at the receiver by recalculating the checksum and comparing it with the value in the checksum field.
4. Connection teardown- A three step process is followed which is analogous to the three-way handshake for connection establishment.

FIN from client, FINACK from server, FINACKACK from client

# ARCHITECTURE

# 

# CLASS DIAGRAM



**NOTE:**

TTPConnEndPoint class, representing an end point of a reliable connection, can be considered analogous to a Socket. It contains the entire logic of Go-Back-N mechanism.

The TTPServer listens for incoming SYN connection requests from clients who are running TTPConnEndPoint and creates a new instance of TTPConnEndPoint on the server side to handle each connection. The TTPServer thereafter continues listening for new connection requests and the actual work of interfacing with the client endpoint is handled by the server TTPConnEndPoint.

# PROGRAM FLOW

1. The FTPServer is configured using Run Configuration to accept 2 command line arguments i.e. TTPServer send window size and TTPServer retransmission timer interval. FTPServer instantiates TTPServer using these parameters.
2. It calls open(int port, int verbose) on TTPServer instance which in turn initializes a DatagramService instance on the specified port.

Note: FTP Server is hardcoded to listen on port 2221 in this case for testing purposes since FTP port 21 is already bound.

1. FTPServer listens continuously for file requests on Port 2221 by calling receive() on TTPServer instance in a while(true) loop. TTPServer receive() in turn calls receiveDatagram() of the underlying DatagramService instance.
2. FTPClient is configured using Run Configuration in Eclipse to accept 2 command line arguments i.e. send window size and retransmission timer interval.
3. It creates an instance of TTPConnEndPoint, which contains the entire logic for Go-Back-N mechanism. TTPConnEndPoint class, representing an end point of a reliable connection, can be considered analogous to a Socket.
4. FTPClient calls open(int port, int verbose) on this instance which in turn creates a DatagramService instance on the specified port. It also sends a SYN request to the TTPServer.
5. The TTPServer which is continuously listening on the specified port, creates a TTPConnEndPoint instance for every new incoming connection i.e. SYN. The client is then serviced in a separate thread by this instance. TTPServer maintains a ConcurrentSkipListMap of all active connections identified by client IP & port.
6. Note that this TTPConnEndPoint shares the same DatagramService instance as the TTPServer. The difference is that while the TTPServer continuously listens using receiveDatagram() of the underlying DatagramService, the TTPConnEndPoint instance implements responds to the received requests and ACK’s using sendDatagram() of the same DatagramService instance albeit in a separate thread. Thus both listening for new connections and interacting with existing clients can happen simultaneously.
7. Thereafter, the TTPServer passes every message from an existing client to the respective TTPConnEndpoint instance for servicing in a new thread.
8. We perform a 3-way handshake to setup connection i.e. SYN from the TTP client, SYNACK from the TTP Server endpoint and ACK from the TTP client.
9. After establishing connection, FTP client sends a request for the file. This request is accepted as an input from the console on running the FTPClient program.
10. The listening TTPServer passes the request to the server endpoint servicing this particular client. If it falls within the receive window of the server endpoint, it acknowledges the packet and writes this request datagram to a buffer (implemented using a Queue) at TTPServer.
11. The FTPServer which is continuously listening through the receive() of TTPServer reads requests from this queue.
12. On receiving a request, the FTPServer launches a ProxyFTPServer instance in a new thread to read the file and transfer the bytes to the FTP client.
13. The ProxyFTPServer instance calls send(byte[]) of the TTPServer to send the file to the FTPClient. It also sends the MD5 hash of the file to the client.
14. The TTPServer in turn passes this file data to the TTPConnEndPoint instance interfacing with the particular client.
15. The FTP Client on receiving this data through its TTPConnEndPoint first verifies the MD5 hash of the file. On verification, the file is created within the ClientFiles folder in the project path.
16. The FTPClient closes the connection by calling close() on the TTPConnEndPoint instance after receiving a file correctly or receiving an error i.e. MD5 hash mismatch.
17. Connection teardown is a 3 step process – FIN from the TTP client endpoint to TTP server endpoint, FINACK from the TTP server endpoint and then ACK from the TTP client endpoint. The TTPServer deletes the entry in its ConcurrentSkipListMap corresponding to the particular connection on teardown. TTPClient endpoint on the other hand, sets its DatagramService instance to null, hence it can no longer send or receive data. The instance will soon be garbage collected.

# FEATURES IMPLEMENTED CHECKLIST

1. Go Back N
2. TTP exposes the following functions to the application layer protocols-
3. open()
4. receive()
5. send()
6. close()
7. Robust against delayed packets, packet reordering, dropped packets and duplicate packets.
8. Fragmentation and Reassembly
9. UDP checksum
10. MD5 hash verification
11. Multi-threaded TTP (handles multiple connections simultaneously).
12. Test cases (Extra-credit)