Computer Networks:

Candidate number: 279187

Introduction

The Trivial File Transfer Protocol (TFTP) is a simple communication protocol utilized to transmit or receive files in a client-server application. We will test the TFTP protocol in two set cases. The first case involves UDP (Task 1), and the second case involves TCP. For these instances, we utilize a simplified version of TFTP, as outlined in RFC 1350, which is limited to support: 1. Octet mode (files transferred only as raw sequences of bytes, not as characters); 2. Error handling only when the server fails to deliver because the file cannot be found 3. Timeouts and retransmissions. This report will describe the specific protocols and how the implementation of each case was handled, including source code, structure, and critical decisions for each task.

1. TFTP OVER UDP:

Protocol Description:

Based on the outline of the RFC 1350, the TFTP protocol handles request acknowledgments through simple opcodes and datagrams. In further detail, TFTP handles: 1. Read Requests (*op_read_request*) 2. Write Requests (*op_write_request*) 3. Data (*op_data*) 4. Acknowledgements (*op_acknowledgement*) 5. Errors (*op_error*). Additionally, the protocol handles network transmission issues such as acknowledgments not arriving on time and having to re-send packets through socket timeouts (*socket.setSoTimeout*)

UDP Server Implementation: (TFTP-UDP-Server)

In the TFTPUDPServer.java file, I bind the server on port 9000 and ensure that for each incoming request, a *DatagramPacket* is received worker thread (*RequestHandler*) is formed to interpret said request, thus achieving concurrency as the main thread is always available for possible new requests, while each of the worker threads handles single file transfers (read/write one file at a time). In terms of the functioning of the protocol handling read, and writing requests in this TFTP implementation by employing The handleReadRequest(...) method for a read request (RRQ) uses a FileInputStream to open the requested local file, reads up to 512 bytes at a time, creates a packet with the data payload, a 2-byte block number, and a 2-byte opcode of 3, and then delivers the packet to the client. After that, it watches for an ACK packet with opcode 4. The identical data packet is sent again after a 3-second timeout if no ACK is received. Until a block less than 512 bytes is transmitted, signifying the end of the file, this process is repeated. The server replies with an error packet (opcode 5) that reads, "File not found," if the file is not present. The handleWriteRequest(...) method generates a local file using a FileOutputStream in response to write requests (see protocol description) and signals readiness with an initial acknowledgement (ACK). After that, it continuously receives data (opcode 3) packets from the client, writes the content to the file, and, for each data block received, returns an ACK with the matching block number. The server resends the final ACK so the client can try again if a data packet is not received within the timeout. When the transmission ends and a data packet smaller than 512 bytes is received, the write procedure is complete.

Run Configuration: *Run Server (server is bound to set port)*

UDP Client Implementation: (TFTP-UDP-Client):

The TFTPUDPClient supported two primary operations: 1. File retrieval 2. File Upload. A read request (*op_read_request*) packet is created by the client and sent to the server in GET mode. After that, it goes into a loop where it gets DATA packets, writes them to a local file, and then returns acknowledgments (ACKs). When a data packet with fewer than 512 bytes—the final block—is received, the transfer is complete. The client attempts to send the read request or the most recent ACK multiple times after a timeout. The client sends a write request (WRQ) in PUT mode and waits for ACK (0). It retransmits the write request until it is successful if no acknowledgement is received. Following that, it pulls 512-byte chunks from the local file and transmits each one as a data packet in the format `[opcode=3][block Number][data]`. It then waits for an acknowledgment for each block and retransmits if required. When a data block is smaller than 512 bytes, the transmission is over. Minimal error handling is implemented, focusing on two scenarios: a server-generated "file not found" error that results in a printed error message, and repeated timeouts that cause the client to terminate the transfer.

Run Configurations:

Get: java TFTPUDPClient <serverIP> get <filename>

Put: java TFTPUDPClient <serverIP> put <filename>

2. TFTP Over TCP

Protocol Description

Block numbers and acknowledgments are not used in the TFTP protocol's block-based method because TCP already ensures dependability and proper ordering. Instead, a brief code (1 for read or 2 for write) is used to identify each request. Block numbering and separate ACK logic are not used in the continuous, stream-based interchange of the file contents, which is followed by a string fileName. (A missing file results in a human-readable notice along with an error code of -1.)

TCP Server Implementation: (TFTP-TCP-Server)

A ServerSocket in TFTPTCPServer.java listens on a specified port (9019). A ClientHandler thread is launched to manage the request each time a client connection is established. In this thread:

It reads the incoming DataInputStream's short opcode (1 for read, 2 for write).

The requested filename is then read from the same input stream.

The server tries to access the requested file from its local directory and then broadcasts its contents to the client if the opcode is 1 (read request). A brief integer indicates the size of each chunk. The server indicates that there is no more data by writing a short=0 after it is finished. The server writes -1 and a message explaining the error if the file cannot be located.

The client sends the data chunk sizes to the server, which reads the chunk bytes and writes them to a newly generated local file if the opcode is 2 (write request). When the client sends 0 to indicate the end of the file, it stops.

Run Configuration: *Run Server (The server is bound to a specific port, and then waits for incoming connections, handling each in a separate thread.)*

TCP Client Implementation: (TFTP-TCP-Client)

In the TCTPTCPClient implementation supported procedures take place as follows:

GET: transmits the filename along with short=1. The server streams chunks. The client reads the short "chunk size" (or -1 for mistakes) for each chunk. The client reports the error message if it is -1. The file is terminated if it is zero. Otherwise, until the transmission is complete, the chunk bytes are written to a local file.

PUT: Transmits the filename, along with short=2. The client sends [short=chunkSize][chunkData] to the server after reading from the local file in 512-byte chunks. The final piece is indicated with a 0 short. After that, the data is written to the server's local directory.

Since TCP is reliable by nature, no explicit timeout or acknowledgment logic is needed. Retransmissions are managed by the OS-level protocol, which also guarantees data arrival.

Run Configurations:

Get: java TFTPUDPClient <serverIP> get <filename>

Put: java TFTPUDPClient <serverIP> put <filename>

Experimental Procedure and Results:

4.1 Testing the UDP Version

- **Basic GET**:
    - **Server**: java TFTPUDPServer on port 9000.
    - **Client**: java TFTPUDPClient 127.0.0.1 get getTestFile.txt.
    - Verified the client obtains an identical local copy of getTestFile.txt.
- **Basic PUT**:
    - Created uploadTestFile.txt locally.
    - Ran java TFTPUDPClient 127.0.0.1 put uploadTestFile.txt.
    - Confirmed the server directory now contains uploadTestFile.txt.
- **Missing File**:
    - java TFTPUDPClient 127.0.0.1 get testNoFile.txt → server responds with an **ERROR** packet. The client prints: "Server error: File not found."
- **Simultaneous Clients**:
    - Multiple clients (both get and put) were launched. Each request spawns a worker thread on the server side. No collisions or interferences were observed.
- **Timeouts**:
    - Introduced short network delays/slowdowns. The client retried if no data/ACK arrived within a few seconds. This validated the setSoTimeout() approach and re-send logic.

4.2 Testing the TCP Version

- **Single GET**:
    - **Server**: java TFTPTCPServer on port 9019.
    - **Client**: java TFTPTCPClient 127.0.0.1 getTestFile
    - Observed correct streaming of chunks until a final block (<512 bytes) signaled completion.
- **Single PUT**:
    - java TFTPTCPClient 127.0.0.1 put uploadTestFile
    - A file named uploadTestFile is created on the server, matching the original size.
- **Multi-Client**:

- Launched two simultaneously, retrieving or uploading. Each connection spawns a dedicated thread on the server. All file transfers proceeded without blocking or data corruption.

- **Error Cases**:
  - If a requested file is missing, the server writes **-1** and an error string. The client logs “Server error: [error string].”
  - No advanced duplication checks, or extra error codes are used, fully adhering to the assignment’s simplified instructions.