COP701: Assignment - 3

Implementation of TFTP Client-Server with Enhanced Features

INDIAN INSTITUTE OF TECHNOLOGY, DELHI
Hauz Khas, New Delhi -110016, India

Under the Guidance of
Dr. SMRUTI RANJAN SARANGI CSE Department

Submitted by :
Rahul Kumar (2023MCS2491)
Atisha Wankhede (2023MCS2498)

Abstract:

This project presents the development of a TFTP (Trivial File Transfer Protocol) client-server system in C++ on a Linux operating system. The implementation adheres to RFC 1350 specifications while incorporating additional features inspired by FTP. These enhancements include the ability for the client to retrieve the server's directory structure, create new folders on the server, and support for handling multiple clients concurrently. Furthermore, data transmission is optimized through compression and decompression using the DEFLATE algorithm, with an added O(n) Huffman algorithm for efficient data encoding.

Acknowledgments:
I express my gratitude to **Dr. SMRUTI RANJAN SARANGI** for their guidance and support throughout the development of this project. Special thanks to the Teaching assistants for their valuable insights and assistance.

Introduction:

The project introduces a TFTP client-server system with expanded functionalities beyond the RFC 1350 standards. The motivation for these enhancements stems from the desire to emulate features found in FTP while maintaining TFTP's simplicity.

The primary motivation behind these enhancements is rooted in the aspiration to amalgamate the simplicity of TFTP with the versatility of FTP. While TFTP's minimalism is advantageous in terms of resource efficiency and ease of implementation, it often falls short when users require more advanced capabilities such as directory manipulation, folder creation, and concurrent client support. By integrating these features into our TFTP implementation, we aim to create a solution that combines the lightweight nature of TFTP with the extended functionality reminiscent of FTP.

This project is positioned at the intersection of practicality and complexity, seeking to strike a balance between simplicity and utility in the realm of file transfer protocols.

Literature Review:

TFTP, or Trivial File Transfer Protocol, is a simple and lightweight protocol designed for transferring files between computers on a network. Developed in the late 1980s, TFTP aims to provide a minimalistic and efficient means of file transfer. Here's a detailed review of its key characteristics and functionalities:

Simplicity and Minimalism:

TFTP is known for its simplicity, emphasizing a minimal set of features. It operates with a small command set, consisting mainly of read and write requests, making it easy to implement and use.

Connectionless Protocol:

TFTP operates over UDP (User Datagram Protocol), which is connectionless. Unlike TCP (Transmission Control Protocol), TFTP doesn't establish a persistent connection before transferring data. While this reduces overhead, it also means that TFTP lacks built-in error recovery mechanisms.

Read and Write Operations:
The fundamental operations in TFTP are reading and writing files. A TFTP client can request a file from a TFTP server (read operation) or send a file to the server (write operation).

Error Handling:
TFTP employs a simple error detection mechanism. If an error occurs during data transmission, TFTP sends an error packet back to the sender, specifying the type of error. However, it lacks automatic mechanisms for retransmitting lost or corrupted packets.

Port 69:
TFTP typically uses UDP port 69 for communication. The client and server exchange messages through this port during the file transfer process.

No Authentication:
TFTP does not include built-in authentication mechanisms. Access control is often managed at the operating system level or through network configuration.

Limited Features:
TFTP's feature set is limited compared to more robust file transfer protocols like FTP. It lacks directory manipulation commands and advanced functionalities, making it suitable for specific use cases where simplicity is prioritized. But have added some directory manipulation features to this project

Commonly Used in Bootstrapping:
Due to its simplicity and lightweight nature, TFTP is often used in scenarios where a small, quick file transfer is needed, such as bootstrapping processes for network devices.

DEFLATE Algorithm:
The DEFLATE algorithm is a widely used compression technique that significantly reduces the size of data for efficient storage and transmission. It combines LZ77, a sliding window compression method that replaces repeated sequences, with Huffman coding, which assigns shorter codes to more frequent symbols. DEFLATE processes data in blocks, dynamically adjusting its compression strategy based on the content. Its block structure, header, and trailer information contribute to its adaptability and effectiveness. DEFLATE is extensively employed in compression utilities and formats like gzip and zlib, making it a versatile choice for enhancing data transfer efficiency in various applications.

Methodology:

In this project, we designed and built the TFTP client-server system by writing code in the C++ programming language. We specifically focused on making it work smoothly on Linux. The design of our system followed the rules outlined in RFC 1350, which is like a guidebook that defines how TFTP should work.

Now, here's the cool part: we didn't stop at the basics. We added some extra features to make our TFTP system more awesome. First off, users can not only transfer files but also do things like checking out what files are on the server, creating new folders, and even having multiple computers (clients) talk to the server at the same time.

To make file transfers faster, we also used a compression technique called the DEFLATE algorithm. This fancy algorithm makes files smaller before sending them, so they take up less space and move quickly through the network. And, inside DEFLATE, we brought in an even smarter buddy called the O(n) Huffman algorithm to make the compression process super efficient.

Local Medium	Internet	Datagram	TFTP

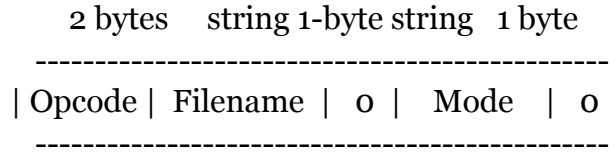
Order of Headers

TFTP Packets

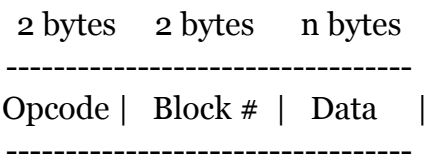
TFTP supports five types of packets, all of which have been mentioned above:

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

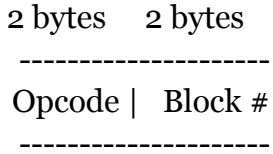
Packet Structure



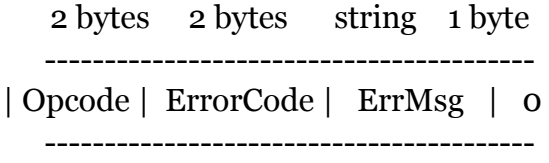
RRQ/WRQ packet



DATA packet

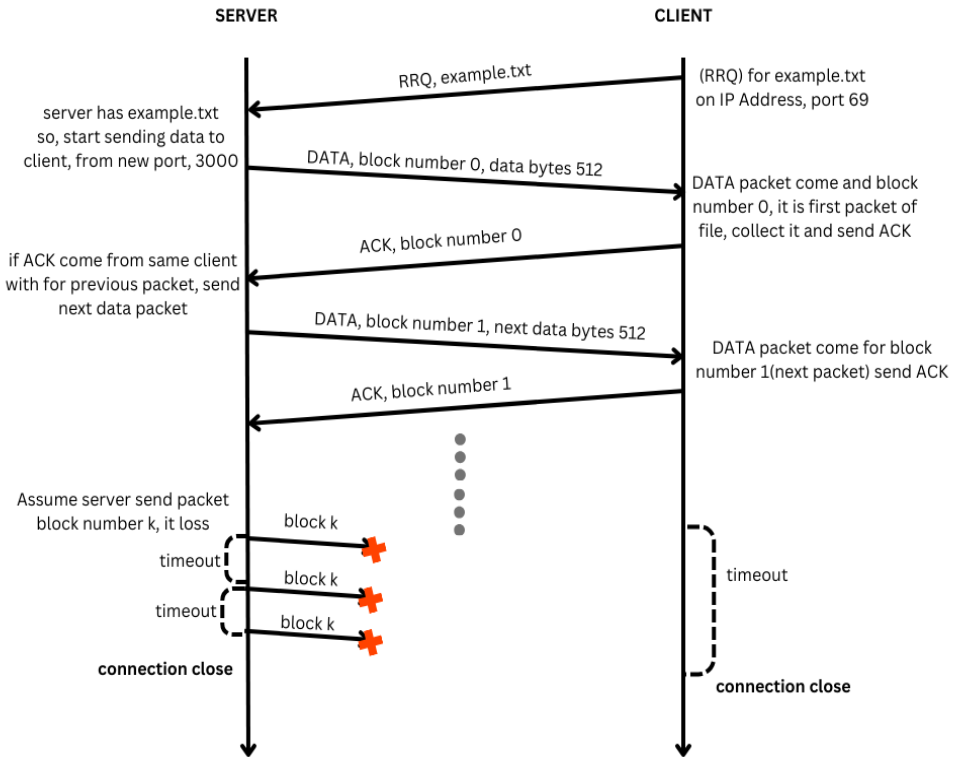


ACK packet

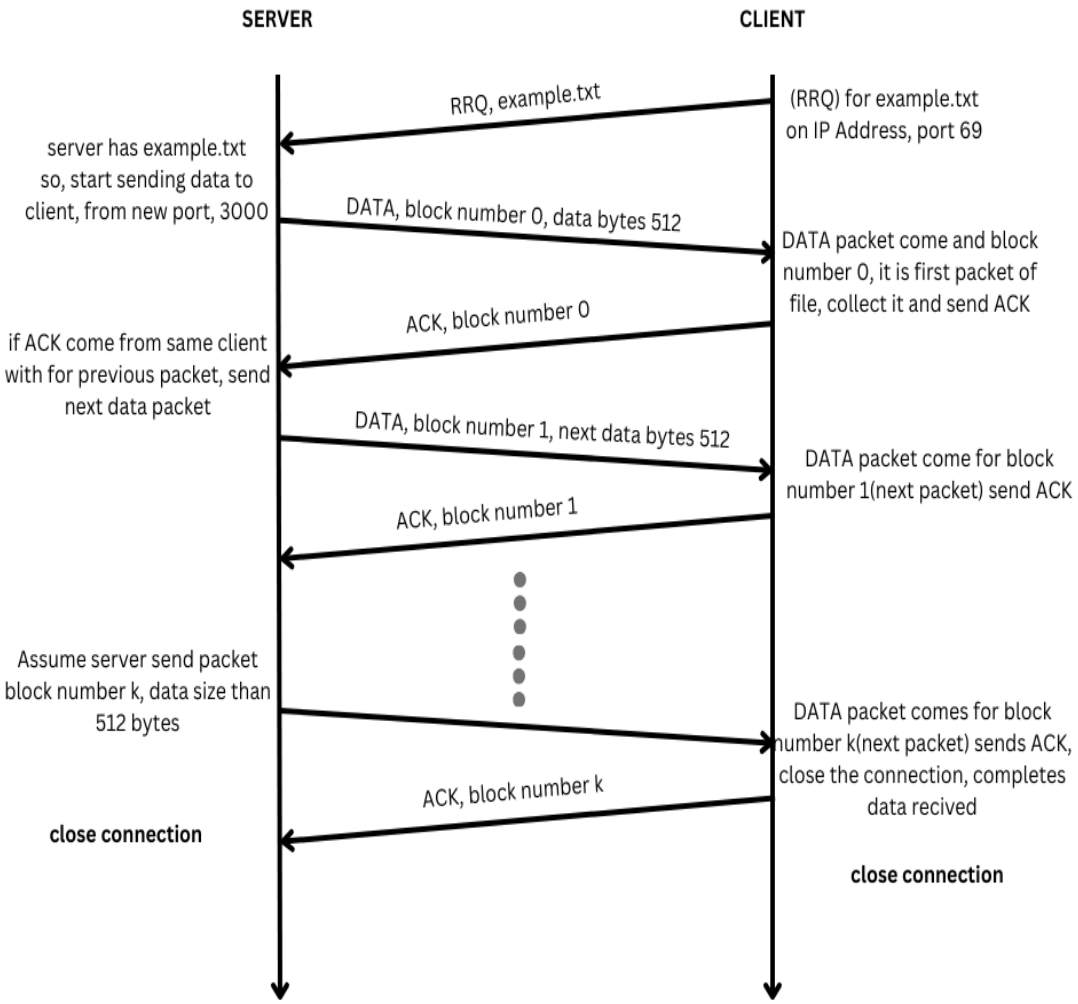


ERROR packet

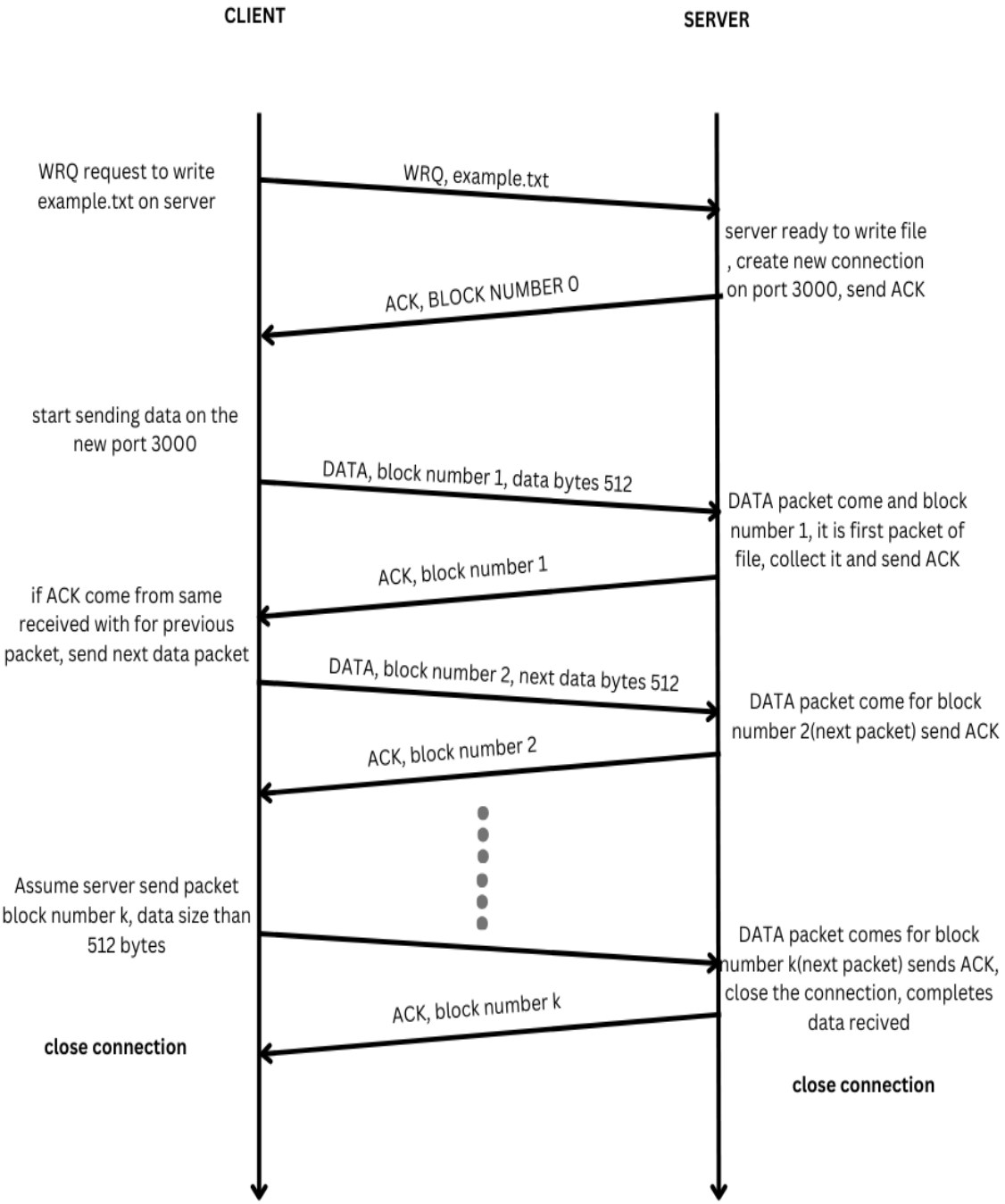
- RRQ request by a client, connection disconnected due to timeout



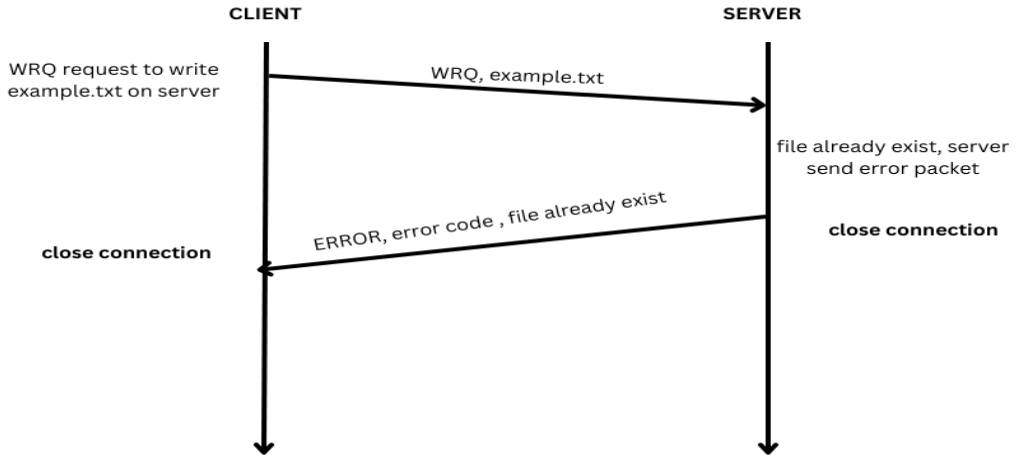
- RRQ request by a client, connection closed after a successful data transfer



- The client sends a WRQ request to write data on the server, data is successfully transferred



- The client sends WRQ for a file, but the file already exists on the server, the server sends an error and the connection closes



Same way other errors will be sent if any other error occurs during wrq request
Other, errors can occur

- Timeout occurs
- Access violation
- Illegal TFTP operation
- Unknown transfer ID
- File not found

For more detail refer to the code

Project Implementation:

Detailed technical aspects of the TFTP client-server implementation, including algorithms used for compression, folder creation, and concurrent client handling. Code snippets and architectural decisions are discussed to provide insight into the project structure.

As the above diagrams explain how the client-server communicates to transfer the data, let's first talk about the project requirements

This project is created on a Linux environment, I am providing the details to download and run the project if you are a Linux user

Download the project

- You can download the project from the zip or you can download the complete project from the GitHub repo if it gets public

How to Run

- Create a folder TFTP and copy the code into this folder
- Move to this folder cd ./path_to_folder/TFTP
- If you want to run the server. Move to server folder cd ./server
- If you want to run the client. Move to client folder cd ./client

Run server

- Move to the server folder
- Clean the server **make clean**
- Build the server **make**
- Run the server in admin privileges **sudo ./server**
- Enter your password
- You can run this server on an available IP address of your system by passing the IP address in the command as, **sudo ./server -ip IP_ADDRESS**

Run client

- Move to the client folder
- Clean the client **make clean**
- Build the client **make**
- Run the client commands
    - ❖ Read a file from server **./client READ filename**
    - ❖ Write a file on the server **./client WRITE filename**
    - ❖ Check the folder structure of the server a the path **./client DIR path (**default ./ for server root**)**

- ❖ Create the folder on the server **./client MDIR <path from the root>**

For all client commands at the end if you want to specify the IP address you can do it -ip ip_address. E.g ./client READ filename -ip server_ip_address

Run tests

- First, you have to download the Google test environment (gtest link)
- Download in the same folder where you have a project
- **git clone https://github.com/google/googletest.git**
- **cd googletest**
- Create the make file, the project already contains the makefile
- **cmake -Bbuild**
- **cmake --build build**
- **sudo cmake --build build --target install**
- **cmake -Bbuild**
- **cmake --build build**
- Now test.cpp is build, you can check your test cases
- **./build/a.out**

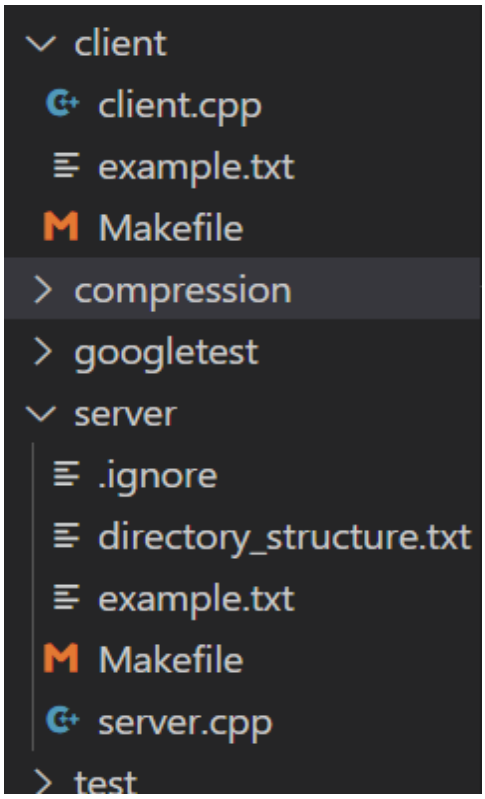
How server code works

The server by default listens on the localhost 127.0.0.1 and on port 69, you can change its IP address by giving the IP address running the run command, now server starts listing on the IP and port 69, if any client wants to connect, and the client will send a request on server address if the request is valid then the server will serve the request until the operation not complete or terminated due to an error, the client first connect on the default port number 69 then server create a new port number for serving this client for subsequence request, server can handle multiple clients at the same time using multithreading, when every a client completely server then the new thread closed and port assigned release for new connection

How client code works

When the client starts it is the IP address on which the server listing, it specifies the READ, WRITE, DIR, and MDIR request, if any other request specify it will throw an error, if the server accepts the request successfully then the client can continue the connection to complete the request otherwise the server will throw the error packet then after listing to error packet client terminate the connection and display the error occur, it also shows the progress of data being send or received

Project structure:



- The client folder contains the client.cpp which is used to connect with the server, by using the make file, make file autorun all commands needed to start the client, for more commands you can refer to the Command section above
- The server folder contains the server.cpp which is used for server actions, by using the make file you can build a server, for all the server-related commands please, the above section for how to run the server
- .ignore file in the server is used to hide the files from the client, when every client asks for server directory structure then by looking .ignore file you can hide the file from the client
- The directory_structure.txt file contains the updated file structure of the server which the client can view
- The compression folder contains all the implementation of compression algorithms
- The test folder contains the test cases for the project

External resources we used:

https://datatracker.ietf.org/doc/html/rfc1350
https://doi.org/10.1145/3342555

Conclusion:

In wrapping up our project, we've done some pretty cool things with TFTP! We made it better by adding features like checking out server files, creating folders, and handling lots of friends connecting at the same time. Plus, we made file transfers faster with a clever compression trick called DEFLATE.

So, what's the takeaway? Our project isn't just about following the rules; it's about making TFTP more awesome. We've given it some new superpowers, and who knows, maybe it can inspire others to do even cooler stuff with file transfers in the future!