Documentation - Concurrent UDP File Transfer

The folder contains the implementation of the server and client programs for a simple TFTP interface that uses server mirrors to download a text file. The following document serves as an explanation of the design for the program. The language C++ was used for this project because of overall simplicity and flexibility to implement the program, as well as considerable experience with it.

Overview:

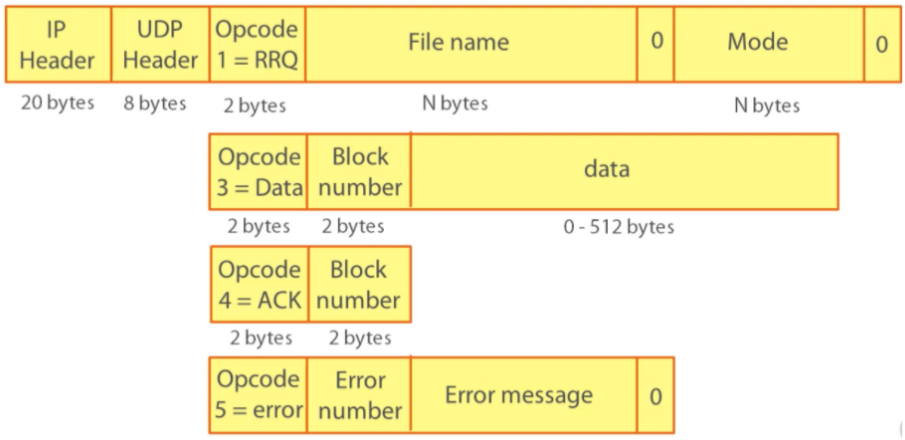
This project consists of a client and server that communicate with each other using a simplified version of TFTP which uses UDP. The goal of this project was to develop a client that reads IP addresses and ports from a file and sends a read request to the corresponding server for a specific chunk of a file contained in the same directory of the running server. By requesting multiple servers for chunks of the same file and then combining them into an output file we achieve more reliability against network problems and a design that improves download efficiency.

This assignment is broken down into two main parts: the client and the server. The client receives as arguments a file path, a number of chunks, and a server descriptor file. The file path represents the path of the file at the server's side in comparison to the current location of the program. The number of chunks represents the number of requests that the client will send to different servers which will then be reassembled into the complete file, and the server descriptor file contains the information of the aforementioned servers. It is the job of the client to handle TFTP requests with concurrency and then reassembled piecewise the file as the packets arrive independently. Also, the client requires error handling in the case where some servers are unavailable, requiring to adapt and resend the corresponding requests.

The server just needs to wait for incoming packets and handle each request in a different thread, opening a new port for the transfer of each chunk to the client. The server must be up and running regardless of the errors encountered.

Logic:

The server initializes a socket and listens until a new packet arrives. From there a new thread is created which opens a new port and sends a response to the client. The server responds with either file size or the requested chunk of a file, together with any error messages that might be produced. All packets sent from the server to the client use a timeout and ACKs to accomplish reliable data transfer, and each thread handles each chunk sequentially; each chunk sent from to the client is broken down into TFTP packets of 512 bytes and sent in order, waiting for the previous ACK to send the next packet. A traditional format of TFTP is used for each packet. The following image shows the corresponding format:



(Figure 1. Source: https://www.linkedin.com/pulse/tftp-client-implementation-c-sumit-jha/).

As soon as a packet arrives at the listening port, an ACK message is sent back to provoke the client to listen for incoming packets. Then the client socket enters a **read()** block awaiting the TFTP packets to arrive. A corresponding ACK message is sent as each packet arrives at the client, but if the client times out the program returns a failure in the file download, and progresses to request the next chunk. The sender of TFTP packets also awaits incoming ACK messages, and if it times out, it resends the message again and increases the count by three. If no ACK messages are received after 3 retransmissions the server quits the communication and sends an error message to the client.

Structure and Design:

The program is broken down into two main programs, and a utility source code with implementations of the TFTP functions that ensure reliable packet delivery. The client program iterates through each indicated file path and requests the servers in the server info file for the chunks of the specified chunk size. The handling of the breaking down into TFTP packets of 512 bytes is done solely by the server, so the client does not need to worry about keeping track of bytes. Any server that responds with an error message or that produces a system timeout is removed from the list of servers until the next iteration for the next file path. The UDP protocol was used for all client-server communication. Each chunk is downloaded into a temporary text file until all threads rejoin the main thread and a reassembling of the final output file is performed. This decision was made to ensure that the heap memory is never exhausted, allowing this file to download files of any size (in test runs it was able to successfully download 10+ GB). Since each thread has to rejoin before reassembling the file, a vector of threads is used to store all current threads. If necessary, a limit to the number of threads could be made using a trivial semaphore to block the program until a thread space becomes available. The server implementation was kept simple to allow for easier debugging, breaking down each important task into functions. A mutex is used in the server to ensure that no more than one thread can read from the input packet buffer, which would cause data to be misplaced and the downloaded file to be corrupted.

In the TFTP implementation, a version of **read(), send(),** and **sendto()** was created in a similar format to the standard. The main difference is the application of timers, retransmissions, and ACK messages to ensure proper communication and transfer between hosts.

Other style decisions:

While being very memory-consuming, **reinterpret cast** allowed for a clear and repeatable way of accessing specific bytes of input and output packets, either for reading or for writing. Also, **memcpy()** was a more consistent and reliable way to copy buffers or sections of buffers. Unfortunately, this hardcoding of the packets and buffers may cause systems running in incompatible 32bit or 64bit processors to have data corrupted. In the future, a proper conversion to network format could be used, but it was omitted to keep the code both simple and readable. Also, the function **err_sys()** was overridden by a custom version to ensure that our error messages accommodate the format and purpose of each task made by the program. Finally, a regex analyzer was used to break down the file path provided by the client to extract the actual file name and create the final output file.

Folder Contents:

UDP_TFTP

myserver - Server executable file

myclient - Client executable file

sample_output.txt - Sample text file for TFTP file transfer

sample_servers.txt - Sample server info file for client

DOC

UDP_TFTP.pdf - This file*

README - Markdown file with author info, folder contents, and short descriptions.

SRC

tftp_server.cpp - Server source code in c++

tftp_server.h - Server header file

tftp_client.cpp - Client source code in c++

tftp_client.h - Client header file

util.cpp - TFTP functions and **err_sys()** in c++

util.h - Utility function header file

Makefile - Run **make** to compile, and **make client** or **make server** for test runs.

BIN

*.o - All object files created by the compiler