CS-371: Introduction to Computer Networking – University of Kentucky
Prepared for Javid Fathi for Dr. Simone Silvestri – April 24, 2020

Project Report
Design of a Network-Private Remote Folder

Abstract

As the final project for this class, a network-private remote folder was developed to demonstrate practical networking techniques at the Application Layer. This project was an individual effort developed by Javid Fathi, with opportunities for long-term support and future development.

Introduction & Motivation

Network-private remote folders are an effective way of accessing, modifying and storing private data across pre-existing computer networks. In short, a folder for private is built on a computer server, along with boilerplate code to handle connections and client requests for the private folder. Separately – and generally on a separate machine – a client software is developed to pass user requests to server for processing. Once successful connected, this client allows a user to access and alter the files present on the folder – without directly accessing the folder or its machine.

In order to practice socket programming and develop networking solutions at the Application Layer, it was requested that students develop a network-private remote folder that could be run using wired connections – expectedly with a single machine. Although this project was originally intended to support wireless connections as well, the COVID-19 outbreak forced students to rescale the project to an individual effort. So, the following implementation assumes that the network runs on a wired – or internal – datagram service.

Project Design & Implementation

The software is extremely modularized in order to ensure long-term support, and developer customizability. Extensive documentation for each function is available in the program files.

Functions

Functions can be identified as one of three types:

1. Functions shared between the server and the client;
2. Functions specific to the server-code; and
3. Function specific to the client-code.

Below, each function's purpose and usage in the program is briefly summarized.

1. Shared Functions
   - Receive_Packet

     Uses the connection socket's file descriptor to receive an expected packet of variable length; returns a packet of bytes to the caller function; used extensively.
   - Download_File

     Supports the download of a file from the server-side to a designated temporary file on the client-side; used by the DIR and DOWNLOAD processes.
   - Process

     On the client side, passes meaningful user input to the process appropriately requested. On the server side, partially decodes and parses a packet of bytes from the client to direct the packet to the appropriate process.

   The following are shared "commands" or process functions that serve specific purposes.

   - LOGIN

     Log in to a specific account on the server – only command permitted after connection until client is authenticated.
   - UPLOAD

     Upload a file from the client's local directory to the server's private folder.
   - DOWNLOAD filename

     Download a file from the server to the local directory
   - DELETE filename

     Delete a file on the server's private folder.
   - DIR

     Transmit and display a directory of the server's private folder to the client terminal.

2. Server-Specific Functions
    - Folder_Documentation

      Document the contents of the server's private folder to the server's internal memory. If previous documentation exists, use it to determine if – and how many times – files have been previously downloaded. Create and update relevant documentation after the session is complete.
    - Main/Initial Driver Function

      Bind the server's reusable socket to a specific IP and port – then listen for and accept connection requests from the client. Ensure that the client logs in and then enter a service loop for the duration of the communication.
3. Client-Specific Functions
    - CONNECT

      Connect to the server with a requested IP address and port number; connection request denied if not equal to the hard-coded IP address and port number. Only process permitted until connection established.
    - Quit

      Hard-coded system exit; called if/when server-side errors occur.
    - Input

      Processes user input to ensure that passed commands are valid. Calls HELP function and soft-quits if requested or deemed necessary.
    - Main/Initial Driver Function

      Simple driver. Displays Header, calls Input function and displays Footer upon a user-requested quit.
    - Terminal-Display Functions:

      These front-end functions were included to ensure the user can easily navigate the client program.
        - Help

          Give the user a helpful guide on command formats.
        - Header

          Displays a program banner and initial command to the user.
        - Footer

          Displays a program footer.

Data Structures

1. Headers
   One of the most important details of the project's implementation was the use of uniform headers in the server and client programs. Combined with the software's emphasis on modularity, these headers allow future developers to efficiently and effectively update the code without seriously impacting the rest of the software.

2. Documentation Dictionaries
   The server's internal code builds and periodically updates Python dictionaries which hold documentation for the private folder. These dictionaries include the number of lifetime downloads for each file on the server, the login information of permitted users and the second-by-second transmission rates of every upload or download on the server (transferring a kilobyte or more of data). These dictionaries are used to update external files which hold the data for future use and analysis – by both the developers and the sever-code itself.

3. Sockets and File Descriptors
   By maintaining sockets and server/client file-descriptors at a global level, the software is capable of handling the connection process across a set of different functions, with the disconnection process being handled by a system exit on either end of the server. This is possible because – whenever the server is sending data, it immediately waits to receive a response; and whenever the client is sending data, it immediately waits to receive a response. If the response received is unexpected – or non-existent – then the software will hard-quit on both sides of the connection.

Program Pseudocode

Given the size of the project, both the server and the client have extensive documentation in their respective code-files. That being said, the following pseudocode will be more of a summary than an explanation of the code developed.

Relevant Server Pseudocode, in Expected Chronological Order:

```
# Import relevant Python classes and modules (os, sys, time, socket, functools.partial)

# Given global definitions for command keywords, documentation dictionaries, relevant sockets
and file descriptors, etc.

main():
        # Modify global variables for server_socket, client_fd
# Run initial documentation on contents of the server folder.
Folder_Documentation()
# Set server_socket to be reusable, bind to permitted (IP, PORT)
server_socket.listen()
while True:
        (client_fd, address) = server_socket.accept()
        client_fd.send(First_Packet = SERVER_NAME)
        # Do nothing until the client correctly logs in.
while not AUTHENTICATED:
                Login()
        # Service loop for remainder of the communication
while True:
                packet = Receive_Packet()
                Process(packet)

Folder_Documentation(updateDoc = False, unless indicated otherwise):
        # Modify global variable for num_Dwnlds
        # Use the Download Doc to populate the numDownloads dictionary.
        if not updateDoc:
                fd = open(DWNLD_DOC, “READ”)
                for filename in os.listdir(SERVER_FOLDER):
                        file_documented = False
                        num_Dwnlds[filename] = 0
                        for documentation in fd:
                                if documentation[name] == filename:
                                        documented = True
                                        num_Dwnlds[file] = documentation[file_dwnds]
                fd.close()
                # Append undocumented files to Download Doc for later.
        else if updateDoc:
                fd1 = open(DWNLD_DOC, “write”)
                for file_data in num_Dwlds:
                        fd1.write(file_data)
                fd1.close()
                if (download_experiments or upload_experiments):
                        fd2 = open(SESSION_RPT, “append”)
                        fd2.write(“This Session:”)
```

```
            for experiment in download_experiments and upload_experiments:
                fd2.write(Experiment Type and Number)
                for datapoint in experiment:
                    fd2.write(Datapoint Number, datapoint)
            fd2.close()

Login():
    # Modify global variable for AUTHENTICATED
    AUTHENTICATED = True
    Failure_Reason = ‘none’
    packet = Receive_Packet()
    if (packet.prefix != CMD_LOGIN):
        AUTHENTICATED = False
        Failure_Reason = NO_LOGIN
    else:
        if (packet.requested_user not in PERMITTED_USERS):
            AUTHENTICATED = False
            Failure_Reason = INVALID_LOGIN
        elif (packet.requested_pwd not PERMITTED_USER[packet.requested_pwd]):
            AUTHENTICATED = False
            Failure_Reason = INVALID_LOGIN
    if not AUTHENTICATED:
        client_fd.send(SERVER_FAILURE + Failure_Reason)
    # Otherwise, user is authenticated!

# This tiny function saves hundreds of lines of code. Same in server and client!
Receive_Packet():
        packet_header = client_fd.recieve(HEADER_LENGTH)
    if no packet_header:
        Folder_Documentation(updateDoc=True)
        sys.exit()
    else:
        packet_length = int(packet_header)
        packet = client_fd.recieve(packet_length)
    return packet

Process(packet):
    packet_cmd = packet.prefix
    if (packet_cmd == CMD_UPLOAD): Upload(packet.payload)
    elif (packet_cmd == CMD_DOWNLOAD): Download(packet.payload)
    elif (packet_cmd == CMD_DELETE): Delete(packet.payload)
    elif (packet_cmd == CMD_DIR): Dir()
    else: No action taken.
```

```
Download_File(filename, fileloc):
    # Modify global variable for download_experiments
    filesize = get_file_size(fileloc)
    client_fd.send(filesize)
    fd = open(fileloc, “read binary data”)
    download_start = time.time()
    bytes_per_second = 0
    prev_time = download_start
    curr_time = prev_time
    transmission_data = []
    for data_chunk in fd:
        client_fd.send(data_chunk)
        bytes_per_second += len(data_chunk)
        curr_time = time.time()
        if (curr_time – prev_time >= 1):
            prev_time = curr_time
            transmission_data.append(bytes_per_second)
            bytes_per_second = 0
    download_time = curr_time – download_start
    if (len(transmission_data) >= 2):
        download_experiments.append(transmission_data)
    if (Receive_Packet() != SERVER_SUCCESS): Client-Side Error
        Folder_Documentation(updateDoc=True)
        sys.exit()
    return (download_time, filesize)
```

All other functions have been deemed beyond the scope of this pseudocode. Please review the server code-file for further documentation (Server\server.py).

Relevant Client Pseudocode, in Expected Chronological Order:

```
# Import relevant Python classes and modules (os, sys, time, socket, functools.partial)

# Given global definitions for command keywords, documentation dictionaries, relevant sockets and file descriptors, etc.

main():
    Header()
    Input()
    Footer()
```

```
Input():
        user_in = ‘’
        num_invalid_in = 0
        while True:
                try: user_in = input(SERVER_NAME >)
                except Ctl+C: break # exits Input

                if no user_in: continue
                elif user_in.prefix == CMD_QUIT: break # exits Input
                elif user_in.prefix == CMD_HELP: Help(), num_invalid_in = 0
                elif user_in.prefix not in VALID_CMDS: num_invalid_in += 1
                elif not CONNECTED and user_in.prefix != CMD_CONNECT: num_invalid_in += 1
                else:
                        if Process(user_in): num_invalid_in = 0
                        else: num_invalid_in += 1
                if num_invalid_in == 3: Help(), num_invalid_in = 0

Process(user_in):
        Command = user_in.prefix
        if Command == CMD_CONNECT: Connect(Tokenize(user_in))
        elif Command == CMD_LOGIN: Login(Tokenize(user_in))
        elif not AUTHENTICATED: return False
        elif Command == CMD_UPLOAD: Upload(Tokenize(user_in))
        elif Command == CMD_DOWNLOAD: Download(Tokenize(user_in))
        elif Command == CMD_DELETE: Delete(Tokenize(user_in))
        else: Serious Error, Quit() # as in hard-quit
        return True

Connect(Tokens):
        # Modify global variables for CONNECTED, SERVER_NAME, server_fd
        if (CONNECTED or Tokens == Invalid_Format): return
        else:
                server_fd.connect(IP, PORT)
                SERVER_NAME = Receive_Packet()
                CONNECTED = True

Receive_Packet():
        packet_header = client_fd.recieve(HEADER_LENGTH)
        if no packet_header:
                Folder_Documentation(updateDoc=True)
                sys.exit()
        else:
                packet_length = int(packet_header)
                packet = client_fd.recieve(packet_length)
        return packet
```

```
Login(Tokens):
    # Modify global variables for AUTHENTICATED, LOGGED_IN_USER
    if (AUTHENTICATED or Tokens == Invalid_Format): return
    else:
        server_fd.send(Tokens.requested_login)
        server_response = Receive_Packet()
        if (server_response.prefix == SERVER_FAILURE):
            print(failure_reason = server_response.payload)
            # Either username/pwd combo not found or server-side error (Hard Quit)
        else if (server_response == SERVER_SUCCESS):
            AUTHENTICATED = True
            LOGGED_IN_USER = Tokens.requested.login.requested_username

Download_File(filesize, fileloc_tmp=TEMP_FILE):
    num_bytes_read = 0
    open(fileloc_tmp, “clear data, then append binary data”)
    while (num_bytes_read != filesize):
        packet = Receive_Packet()
        fd_tmp.write(packet)
        num_bytes_read += len(packet)
    fd_tmp.close()
    server_fd.send(SUCCESS)
```

All other functions have been deemed beyond the scope of this pseudocode. Please review the client code-file for further documentation (Client\remote_folder.py).

Methodology & Rationale

This server-client software started with a series of rudimentary socket programming tutorials in Python, which were developed into a comprehensive software through an Object-Oriented approach to the project.

Although this approach made the final result much longer than necessary, it does make long-term support of the software much easier to manage.

The Receive_Packet() and Download_File() functions are the most important functions of the server and client code, as both are extensively used to streamline the communication of the program processes.

It was assumed that the server’s private folder would be untouched – expect by the *documented* commands of the client – for the entire time that the server was awake, and so the processes did not include an error case where a file that was present at the start of the session was removed by an non-client process. This was assumed because the project specified that the software services a *private* folder – one which would, in practice, remain unaltered by unauthenticated processes.

The LOGIN process was implemented in addition to the baseline expectations of the project, with a username and password expected. Usable combination include:

jfathi, password
silvestri, password

It should be noted that this login data was hard-coded in to the server's global headers (PERMITTED_USERS). Given additional time, this would have been extended to a private text file, with opportunities to LOGOUT, document a user's sessions and REGISTER new users.

While the phrase ACK is nowhere to be seen the code-files, packet acknowledgements are continuously used in the software implementation – with each generally referred to as a server_response or a client_response – generally piggybacking off the next packet required (i.e.: filesize, login_data, prefix vs. payload). Although it was assumed that the underlying datagram service was both wired and reliable – error cases were assumed to be possible but were *not* to be corrected. If a packet was received with unexpected data, the software was to be shut down on both ends of the communication.

Furthermore, it should be noted that software was built to support no more than one client-connection at a time. As soon as the client disconnects, the server shuts down. This was done for the ease of experimentation on the part of the developer. Although this can be redesigned, it would take a few lines of error-case code – causing a break in main() or a return to main – each time the Receive_Packet() is invoked. Although feasible, this would drastically extend the length of the program – for reasons outside of the project specifications. Needless to say, it was not pursued.

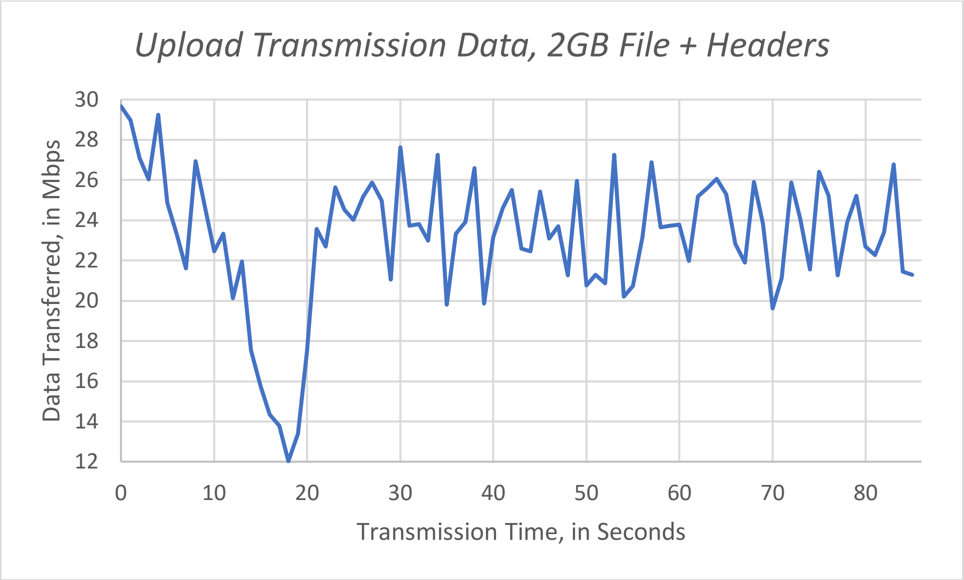
Transmission Experiments

The following experiments were conducted using a 2 Gigabyte file of "dummy data" passed both ways across the channel.

It should be noted that the following experiments do *not* count bytes transmitted that belonged to packet headers. Header lengths (with a constant length of 10-bytes) were ignored in order to assess the effective rate of incoming and outgoing data, as it would be relevant to the UPLOAD and DOWNLOAD processes.

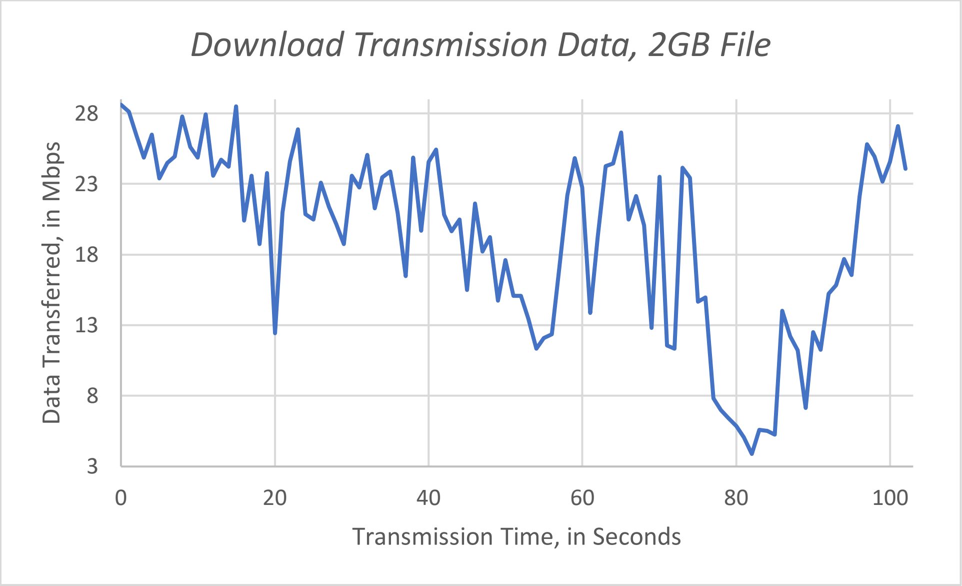
So, while a 2GB file transfer would pass 2001953126 bytes across the channel, only 2000000000 could be registered as data. Furthermore, neither implementation is expected to register exactly the number of bytes in a file. However, the number of unregistered bytes has been deemed negligible for our purposes, specifically when not considering the first and last datapoints in the stream.

Upload Experiment



The graph above shows an initial variation in the TCP congestion window, with an eventual stabilization of the amount of data permitted on the channel around the 30-second mark.

Download Experiment



The graph above shows an serious variation in the TCP congestion window for the duration of the experiment, with the most significant changes in the bandwidth available taking place over the final 20 seconds of transmission.

Conclusions

In conclusion, a software private-remote folder was developed using socket programming techniques and object-oriented programming. This software has been developed for long-term support and customization by future developers. It is likely that the most difficult extension to this project would be the conversion of the software to support wireless communication channels. However, it would be the most useful and practical extension, by far.