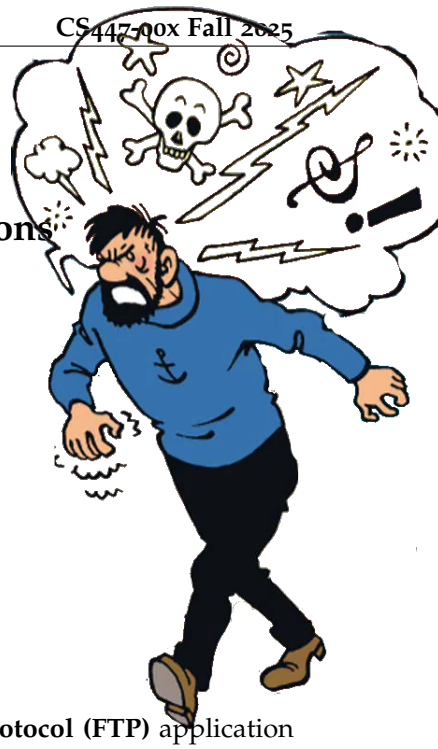


CS447-00x : Networks and Data Communications

Programming Assignment #1 (P2)

Total Points: 200

Assigned Date : Thursday, October 23, 2025
Due Date : Thursday, November 13, 2025 @ 01:59:59 p.m.

Overview

Your second programming assignment is to implement a **Distributed File Transfer Protocol (FTP)** application using the socket interface. Based on the classic FTP (RFC 959), this assignment requires you to build a federated file network. This will allow you to explore client-server architecture, complex peer-to-peer communication, and dynamic data management.
Given the 3-week timeframe, we'll focus on a subset of the RFC 959 protocol, but with a significant "distributed" enhancement. This will provide valuable experience in:

a. Applying and expanding upon networking concepts from P1 within a complex, real-world protocol.
b. Proficiently reading and interpreting RFC 959 specifications.
c. Implementing a "data channel redirect" model, where one server orchestrates a file transfer directly from a peer to a client.
d. Developing robust peer-discovery and state management mechanisms.

Back Story

"Billions of blue blistering barnacles!" Captain Haddock bellowed, sending a flurry of crumpled manifests scattering across the deck of the Karaboudjan. *"This file system is a mess! By the time I get wind of which ship has which cargo manifest, it's too late! We're losing valuable trading opportunities!"* He paced furiously, the ancient file-copying system failing him once again.
Mik, ever the resourceful companion, emerged from the radio room, a glint in his eye. *"Captain, my dear friend,"* he announced, *"I believe I have stumbled upon a solution! I've been studying this remarkable technology called FTP, the File Transfer Protocol!"*.
Mik unrolled a schematic. *"Imagine, Captain! You connect to **any** ship in the fleet, just once. You ask for a file list, and my system gives you a **unified manifest** of all files on **all** ships! Then, you ask this ship for any file, and it will be smart enough to **orchestrate the transfer for you**, even if the file is on another ship!"*.
Haddock, intrigued, stroked his beard. *"Thunder and lightning, Mik! A single connection to see everything, and it handles all the fetching? If this 'Distributed FTP' contraption can save me from this administrative nightmare, I'm all ears! Make it so!"*.

Technical Requirements: `client`-Server

1. Your FTP application will utilize TCP for `client` ↔ `server` and `server` ↔ `server` communication.
2. Refer to **RFC 959** (https://datatracker.ietf.org/doc/html/rfc959) for command descriptions, response codes, and sample interactions.

3. Implement the following minimum capabilities and command syntax:
   - **Control Connection Commands**:
     - `USER anonymous`: Authenticate user.
       - The `server` should acknowledge the user authentication with the reply `230 User logged in, proceed.` and reject all other usernames with the reply `530 Authentication failed.` **and** close the connection.
     - `PWD`: Print Working Directory.
     - `CWD <path>`: Change Working Directory. Your server should support navigating into subfolders.
     - `CDUP`: Change to Parent Directory.
     - `QUIT`: Terminates connection.
   - **Data Connection Commands (Passive Mode)**: (only mode implemented in this assignment)
     - `PASV`: Enter passive mode. The `server` must respond with `227 Entering Passive Mode (h1,h2,h3,h4,p1,p2)`.
       - `h1,h2,h3,h4` are the 4 bytes of the server's IP address.
       - `p1,p2` are the 2 bytes of the new port number. The port is calculated as **`(p1 * 256) + p2`**.
       - See RFC 959, Section 4.1.2 for the full specification.
     - `LIST [path]`: Lists contents of the current or specified directory over the data connection.
     - `RETR <path>`: Retrieves a file over the data connection.

   Data Connection: The data connection is ephemeral and used for one transfer only.
   - For **every** `LIST` or `RETR` command, the client must first send `PASV`, connect to the new data port, and **then** send the `LIST/RETR` command over the control connection.
   - If a `client` sends `LIST` or `RETR` without an established data connection, your server must respond with `425 Can't open data connection.`
   - **NOTE**: If the `server` notices that the `client` is asking to `RETR` a remote file, then negotiate and grab the corresponding ephemeral socket address from the corresponding peer (based on the map) and relay to the `client` using a `227` response as if responding to a `PASV` command.
   - The `server` is responsible for closing the data connection immediately after the transfer is complete.
5. **Security**: Your server must be "**jailed**" to the `db/` root directory. No command (`CWD`, `RETR`, etc.) should ever be able to access or modify files or directories outside of this `db/` folder.
6. Ensure `server` responses include correct RFC 959 reply codes for all commands and errors.
7. Support multiple concurrent `clients`.

Technical Requirements: P2P (The Distributed System)

This is the core of the assignment. Your `server` must not only serve its own files but also act as part of a distributed network, providing a unified view of all files.

1. `servers` must listen for peer connections on a designated port, used exclusively for peer discovery and `server` ↔ `server` (P2P) communication. Assume all `servers` use the same P2P port but on different hosts.
2. `server` ↔ `server` communication should operate as background processes without user interaction.
3. **Peer Authentication**:
   - Peering servers authenticate by sending `USER peer@<IP_address>` (e.g., `USER peer@192.168.0.10`). The sending/requesting `server` must programmatically determine its own IP address.
   - The receiving server compares the IP address received with the `USER` command against the **actual** source IP from the incoming socket for authentication purposes.
   - If they match, reply `230 Peer authenticated, proceed`. Else reply `530 Authentication failed` and close the connection.

   Only after this handshake can other P2P commands be sent.
4. **Peer Discovery and Scanning**:
   - `servers` should periodically discover peers and their file lists (interval defined in `server.conf`).

- Your server must scan the `PEER_SUBNET` (e.g., `192.168.0.0/28`) on the designated P2P port to find other servers.
- **Solo Mode**: If the `PEER_SUBNET` value is left empty in `server.conf`, the server should run in "solo mode" and disable all P2P scanning.

5. **Unified `LIST` Command**: When a client sends `LIST`, the server must:
   i. Connect to all known peers and perform its own P2P `LIST` to build a master map of all files on the network. Ensure not to include duplicate and/or stale mappings due to successive scans and/or peers leaving.
   ii. Send a single, unified list to the client over the data connection. Refer the sample `LIST` output below for the expected format.
6. **Sample `LIST` Output**:
   - When the client is at the root (`db/`) and runs `LIST` (showing a mix of local and peer-owned folders):

     Answer

     ```
     drwxr-xr-x 1 local 0 Oct 22 16:30 comp.security
     drwxr-xr-x 1 local 0 Oct 22 16:30 news.world
     drwxr-xr-x 1 peer 0 Oct 22 16:35 reddit
     ```

   - When the client runs `LIST comp.security` (showing a mix of local and peer-owned files):

     Answer

     ```
     -rw-r-r- 1 local 1234 Oct 22 16:30 180432.txt
     -rw-r-r- 1 local 1234 Oct 22 16:30 69986.txt
     -rw-r-r- 1 peer 1234 Oct 22 16:36 70525.txt
     -rw-r-r- 1 peer 5678 Oct 22 16:35 70681.txt
     ```

7. **Distributed `RETR` (Data Channel Redirect)**: This is the most complex part. When a client requests a file:
   i. **If file is local**: The `server` handles it normally (`PASV` → `RETR`).
   ii. **If file is on a peer (`server`$_B$)**: The contact `server` (`server`$_A$) must orchestrate a redirect:
      - `client` sends `PASV` to `server`$_A$.
      - `server`$_A$, in the background, connects to `server`$_B$ (P2P) and sends `PASV`.
      - `server`$_B$ responds to `server`$_A$ with `227 ... (IP_B, Port_B)`.
      - `server`$_A$ responds to the `client` with `227 ... (IP_B, Port_B)` (forwarding `server`$_B$'s details).
      - The `client`, unaware, opens its data connection directly to `server`$_B$.
      - `client` sends `RETR <file>` to `server`$_A$.
      - `server`$_A$ forwards the `RETR <file>` command to `server`$_B$.
      - `server`$_B$ streams the file directly to the `client`.
      - `server`$_A$ forwards the final `226 Transfer complete` from `server`$_B$ to the `client`.
8. **P2P Command Subset**: After the initial `USER <ip_address>` authentication, servers will use the standard FTP commands (`CWD`, `PASV`, `LIST`, `RETR`) to communicate with each other.
9. **Error Handling**: Implement robust error handling for P2P operations.
10. Servers must implement a mechanism to detect connection termination by remote peers (e.g., `recv()` returning 0 or `ECONNRESET` handling). When a peer disconnects, the server must immediately update its internal file map to remove all entries associated with that peer. This ensures clients do not receive stale information and are not given redirects to offline servers.

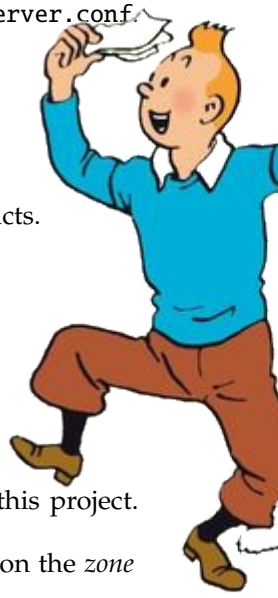
Functional Requirements

1. **Configuration Files**: Utilize the `server.conf` to define server's runtime parameters. Here's the assigned format:

server.conf

```
PORT=
PEER_INTERVAL=
PEER_SUBNET=
```

2. **P2P Port Allocation**: Calculate P2P port by adding a fixed offset of `200` to the `PORT` value from `server.conf`. e.g. If `PORT=30000`, the P2P port will be 30200.
3. **File Management**:
   - Files are stored in a `db` folder under server's working directory (this is the "root").
   - Subfolders in `db` represent different directories.
   - A sample file system under the `db` folder will be provided.
   - **ALERT!** Run each server from a unique folder (e.g., `server-1`, `server-2`) to avoid file conflicts.
4. **Case-Insensitive Commands**: Server should process commands case-insensitively.
5. **`client` Exit**: `clients` should exit gracefully using `QUIT`.



Logistics

1. **Server Code**: Feel free to leverage the `server.cpp` starter code from P1 as a foundation for this project. However, you'll need to adapt it to handle both `client-server` and `P2P` technical requirements.
2. **Client Code**: Use `TELNET` or `NETCAT (nc)` as your client. Both these tools are readily available on the *zone server*. No need to write (or submit) a project specific client code.
3. **Testing Environment**: Leverage the *zone server* during testing to mimic a real-world network topology.
4. **Testing and Documentation**:
   - Thoroughly test your application with various scenarios (local files, remote files, single server, multiple servers, peer disconnects), documenting your process and results with screenshots.
   - Ensure your code compiles and runs on a Linux machine. Provide a `README` with clear compilation instructions and any additional software dependencies.
5. At the end of your implementation, you should be able to:
   - Configure and run one FTP `server` and connect to it.
   - Perform `LIST`, `CWD`, `CDUP`, `PWD`, and `RETR` on local files.
   - Bring a additional peer FTP `server` online.
   - Observe that the first server's `LIST` command now shows a unified view of files from all online servers.
   - Successfully `RETR` a file from the first server that is physically located on another server.
   - Take one or more of the peer `servers` offline and observe that its files are (after a short delay) no longer listed by the first server.

Instructions

- **Start early!!** This assignment is a multi-faceted adventure involving substantial RFC reading and intricate implementation. Starting early is crucial to avoid a last-minute scramble.
- Begin by spending some time reading relevant sections from **RFC 959, FIle Transfer Protocol (FTP)** (https://datatracker.ietf.org/doc/html/rfc959) to gain a foundational understanding of how FTP works.
- Start with a minimum viable solution, gradually adding complexity and features. Maintain good version control habits throughout the development process. **Start simple, finish BIG!**.
- Implement your solution in C++. The sample code is written for C++20. Write clean, readable code, adhering to a recognized style guide like Google's C++ Style Guide https://google.github.io/styleguide/.
- Plan and design your server's architecture and data management strategy carefully. Document any design decisions you make, especially those based on your interpretation of the RFC, with clear explanations and citations.

- Always make design decisions within the boundaries of the RFC specifications. If unsure, consult the instructor for clarification.
- DEADLINE: **Thursday, November 13, 2025 @ 01:59:59 p.m.** through Moodle. Email submissions or requests to "review" your implementation before submission are not honored.

Deliverables

A complete solution comprises of two attachments:

1. Report (**pdf**): The report itself does not carry a lot of points. It's there as evidence/proof of work for partial credit considerations in situations were testing fails or is inconclusive. As such, use the report to convey information in a non-ambiguous way to maximize partial credit potential.
2. Compressed Tarball (**siueID-p2.tgz**):
   - Source Code Directory: Include all your C++ source code. As noted above, a client code is unnecessary.
   - `Makefile`: A Makefile is required. The instructor should be able to compile your code without errors or warnings by simply typing **make**.
   - `README`: Provide clear instructions on how to manually compile and run your code in case **make** fails.
   - Alert! Make a note to ONLY to include required files in your tarball. Unsolicited `config` files, `db` folders, executables, and/or hidden metafiles/folders will result in penalties.
   - Use the following command to create the compressed tarball:

     ```
     tar -zcvf siueID-p2.tgz p2/.
     ```

     Replace `siueID` with your actual email id *(not your 800 number)* and `p2/` with the name of your source code directory. Run this command from the immediate parent directory to avoid unnecessarily deep folder structures.

This assignment is a challenge designed to push your boundaries and foster growth. Embrace the learning process and don't hesitate to seek help when needed. Remember, the goal is to master these concepts, not just complete the assignment. Plagiarism, whether from classmates, online sources, or AI tools, stifles learning and compromises academic integrity. The instructor actively uses MOSS http://theory.stanford.edu/~aiken/moss/ to check for software similarity. Plagiarism has severe consequences as mentioned in the course Syllabus. No exceptions.

Your code should be a testament to your abilities, not a copy of someone else's work. Collaborate, don't copy! Learning from peers is great, but copying code (from classmates or online) is strictly prohibited. Cite any external resources you use for ideas, and then implement those ideas in your own way.

Some Useful Resources

- Linux Man pages – found in all Linux distributions
- Beej's Guide to Network Programming – A pretty thorough free online tutorial on basic network programming for C/C++ https://beej.us/guide/bgnet/
- Linux Socket Programming In C++ – https://tldp.org/LDP/LG/issue74/tougher.html
- The Linux HOWTO Page on Socket Programming – https://www.linuxhowtos.org/C_C++/socket.htm
- File Transfer Protcol (FTP) RFC #959 https://tools.ietf.org/html/rfc959