Assignment 5: Implement Client Server Applications using Socket Programming

Deadline: ~~15~~th 19th November 2024, 11:59 PM

Goal of the Assignment: Get familiar with socket programming and learn to create your own client-server applications using socket programming.

Part 1: Implement a Collaborative Text Editor using (TCP) socket programming. [20 Points]

Create a collaborative text editor using client-server socket programming, allowing multiple clients to connect to a server, make edits in real-time, and see each other's updates. The server maintains a shared document and writes all changes to a file named temp.txt to ensure the document is saved.

Part 1 Requirements:

1. **Server Responsibilities:**
    - Listen for incoming client connections.
    - Accept multiple client connections (up to 10 clients).
    - Store and manage a shared document in memory.
    - Broadcast updates received from any client to all other connected clients.
    - Append each update to temp.txt to save the document's contents.
    - Maintain proper synchronization for concurrent edits using threading and locks.
    - Print messages to indicate server status and activity.
2. **Client Responsibilities:**
    - Connect to the server and receive the current state of the document.
    - Display incoming updates from the server.
    - Allow users to make edits, which are then sent to the server.
    - Print messages to show connection status and any received document updates.

Detailed Instructions:

1. **Setup and File Creation:**
    - The server should create temp.txt if it does not already exist. Each update received from a client should be appended to temp.txt.
    - When starting the server, print: "Server listening on port 8080"
    - If there is an error opening or creating temp.txt, print an error message: "Error opening file temp.txt"
2. **Server Functionalities and Message Flow:**
    - **Accepting Connections:**
        - The server should handle up to 10 clients concurrently, with each client served by a separate thread.
        - For each new client, print: "New client connected: <client_address>"
    - **Sending Initial Document State:**
        - When a client connects, the server should send the current state of the document to the client.
        - If the document has any pre-existing content, the client will receive it upon connection.
    - **Receiving and Broadcasting Updates:**
        - When a client sends an update, the server should:
            - Append the update to the document variable in memory.
            - Write the update to temp.txt for persistence.
            - Broadcast the update to all connected clients except the sender.
        - **Server Print Messages:**
            - When an update is received, print: "Received update from client <client_address>: <update_content>"
            - When broadcasting, print: "Broadcasting update to all clients"
    - **Client Disconnection Handling:**
        - If a client disconnects, the server should:
            - Remove the client from its active list.
            - Print: "Client <client_address> disconnected"
3. **Client Functionalities and Message Flow:**
    - **Connecting to Server:**

        - Upon successful connection to the server, the client should display: "Connected to the server. Start typing to edit the document:"
    - **Receiving Document State and Updates:**
        - When the client receives an update from the server, print: "Document updated: <received_content>"
    - **Sending User Edits:**
        - The client captures user input and sends it as an update to the server. After sending, no additional message is printed.
4. **Concurrency and Synchronization:**
    - Use proper logic [Hint: mutex] to lock and unlock access to the document data in memory and file operations to avoid race conditions.
    - Each client connection should be handled by a separate thread, allowing multiple clients to edit simultaneously without conflicts.
5. **File Saving and Persistence:**
    - Each client update is appended to temp.txt as soon as the server receives it.
    - The file temp.txt serves as a backup of the document's state so that even if the server stops and restarts, it can load and continue from the last saved state (this feature would need further enhancement to read from the file at server start-up).

Error Handling:

1. **Server Startup Errors:**
    - If there is an issue creating or binding the server socket, print: "Server error: Unable to start server on port 8080"
    - If there is an error opening temp.txt, print: "Error opening file temp.txt"
2. **Client Connection Errors:**
    - If the client cannot connect to the server, print: "Connection failed. Please check the server and try again."

Part 2: Implement Leader Election and torrent-like File sharing in peer-to-peer Networks using (TCP) socket programming. [25 Points]

Implement a simple peer-to-peer (P2P) torrent-like network with three nodes using socket programming, in which one node is designated as the leader. The leader will track available files across the network and facilitate file-sharing between nodes. There is no pre-defined leader in the network, and peers elect the leader.

You must write programs running on three nodes and achieve the following task.

Q1) Elect a node among the three as a leader. The leader node is a peer but keeps track of the following:

1. Files available on the p2p network, along with IP addresses and port numbers from where they can be obtained.
2. Logs must be stored in a file named "**peer_files.log**".

Leader Election: Leader election can be achieved by following the below-mentioned instructions:

a. Assign distinct identification numbers (ID) to each node. Depending on which you feel comfortable with, this can be static or random.

b. Set up TCP connections among the three nodes. To set up a TCP connection on a peer node, follow the instructions under the **"Instructions"** heading.

c. Exchange the identification numbers.

d. Choose the node with the lowest identity as the leader.

e. Once the leader is elected, the other two peers should now remove any connections they have among themselves.

After the leader election, to keep a log of files available among peers, the leader node must be able to request and receive file logs from other peers. For this, you must implement a "**SEND**" command, which the leader node sends to the peer, and in response to the "**SEND**" command, the peer shares its log of files (list of files along with its path) available in the server directory with the leader. You must print the leader's log. Log of files will be used further in order to solve the problem.

Hint: In setting up the TCP connections, you should order the server and client functionality across the three nodes.

Q2) Peer X requests the leader to get an image file named **'rollnumber.png'**, which is nothing but an image with your roll number (it might be a screenshot of your roll number or something else). Using the file tracking information, the leader must locate and retrieve the file from the appropriate peer (Peer Y) and send it to Peer X. If the leader itself has the file, it can directly send it to Peer X. Also, If the leader cannot locate the file, it must respond with "**404 Not Found".**

Instructions

1. Each peer creates 2 sockets per connection, one for listening (working as a server) and another to initiate the connection (working as a client).
2. You might need to use multi-threading to support multiple P2P connections.
3. Each node must be running two services (functions) i.e. server and client.
4. The client service on each node will first connect with the other two peers. As soon as the connection is established, the client broadcasts its ID.
5. The server will accept connections and receive the peer's ID. Once all IDs are obtained, each node will update its leader according to the policy mentioned earlier.
6. Once elected, the leader will send a "**SEND"** message to the peers and seek the log information of the files available. You can use the Unix command to retrieve the file name and path available in a server directory.
7. After receiving the information, each node stores it in its server directory with the name "**Peer_files.log**"
8. Properly document your solution and approach with justification.

Part 3: Making Echo Client/Server "Protocol Independent" [15 Points]

Revise echo client and server to be protocol independent **(support both IPv4 and IPv6).**

Hint 1: sockaddr is too small for sockaddr_in6. sockaddr_storage has enough size to support both sockaddr_in and sockaddr_in6. (You will see this in the server-side program.)
Hint 2: integrate getaddrinfo to avoid typing IPv6 address on your CLI
Hint 3: you may use hostname (IPv4: "localhost", IPv6: "ip6-localhost" address to develop/demonstrate the software on Ubuntu. They're written in "/etc/hosts".

Instructions for Implementation:

- You may choose any programming language (C, JAVA, Python, etc.)
- The software must be based on Socket Programming.
- Wrappers of API must not be used (messaging etc). Use send/recv or read/write using a TCP/UDP socket.
- Keep a record of Reference.

Deliverables in a tarball on GC:

1. A report detailing your implementation and the results. The core idea of your answer to each question. Better visibility, like screenshots of the application, will be appreciated. One single report file (<your roll no>_<Name>_<Assignment5>.pdf for all three parts.
2. Screenshots showing the working of your code.
3. Create a separate folder for each part and include the source code, executables (for C/C++), and README as a separate file for each part so that TAs and instructor can compile source code and execute the binary anytime.
4. Submit all files in a single zip file named as <your roll no>_<Name>_<Assignment5>.zip
5. Report Quality **[2 Points].**

Note: A plagiarism check will be done on your submitted code/report. We may call you for a presentation. In the presentation, you are expected to explain and demonstrate the applications you built.