UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

Department of Computer Engineering

CEF 488:

SYSTEMS AND NETWORK PROGRAMMING

LAB 5

Distributed Systems (Multi-Computer Communication and Synchronization)

GROUP 10 MEMBERS	MATRICULE
EPANDA RICHARD JUNIOR	FE22A206
FOMECHE ABONGACHU SIDNEY	FE22A218
EBONG BAO SONE	FE22A194

1. Introduction

- **Course**: Systems and Network Programming (CEF488)
- **Lab**: Lab 5 – Distributed Systems: Multi-Computer Communication and Synchronization
- **Environment**: Ubuntu Linux on 3 machines or 3 terminal tabs with different ports; GCC; threading support (-pthread) or fork().

2. Objective

- **Primary Goal**: Implement a small distributed system in which three computers (or processes) communicate and synchronize tasks over a network, applying TCP/UDP sockets and synchronization techniques.
- **Outcome**: Build:
  - A multi-client TCP server that handles simultaneous connections from two clients.
  - Clients that communicate via the server (server relays messages).
  - Real-time data synchronization: when one client updates, server notifies other client immediately.
- **Key Concepts**: Multi-client handling (threads or fork()), message routing, synchronization signals, data consistency, coordination across nodes.

3. Tools & Software

- **Compiler**: GCC with pthreads support (gcc -pthread if threads used).
- **Editor**: Nano Text Editor
- **Terminal**: Linux
- **Network**: 3 machines on same LAN (wifi), or on one machine using different ports (for testing).
- **Utilities**: netstat, ip addr, optionally telnet for testing; logs printed to console for verification.

4. Lab 5 – Task 1: Multi-Client TCP Server

Design Choices

- **Approach**: Use threads (pthread) to handle multiple clients concurrently.
  - Alternative: fork(), but threads lighter and easier to share in-memory data structures if needed.
- **Server responsibilities**:
  - Listen on a designated port (e.g., 10000).
  - Accept incoming connections from two clients (or more).
  - For each new connection, spawn a thread to handle that client.
  - Maintain a list of active client sockets (e.g., in a shared array protected by mutex if needed).
  - When a client sends a message, server processes it (e.g., logs, or prepares to forward to other client).
- **Client responsibilities**:
  - Connect to server IP:port.
  - Send initial "Hello from Client X" message.
  - Then either wait for messages from server (relay from other client) or send updates.

Steps Taken

1. **Write multi-client server code** in multi_server.c:
   - Include <pthread.h>, <arpa/inet.h>, <netinet/in.h>, <unistd.h>, <stdio.h>, <stdlib.h>, <string.h>.
   - Define MAX_CLIENTS = 2 (or configurable).
   - Data structures: array of client sockets, a mutex to protect access.
   - In main():
     - Create socket, bind to port (e.g., 10000), listen().
     - Loop: accept() new connection; when accepted, store socket in array, create thread: pthread_create(&tid, NULL, client_handler, (void*)&client_socket).
   - **client_handler**:
     - Receive initial greeting.
     - Enter loop: recv() messages from this client. On receiving, print "Server received from client X: ...".
     - Relay to other clients: iterate over client list, for each other client, send() the message.
     - If client disconnects (recv() returns ≤0), remove from list, close socket, exit thread.

        - Use mutex when accessing shared client list.
2. **Compile**:
3. gcc multi_server.c -o multi_server -pthread
4. **Write client code** in client.c (same for both clients, with minor differences in initial ID):
    - Include <arpa/inet.h>, <netinet/in.h>, <unistd.h>, <stdio.h>, <stdlib.h>, <string.h>, <pthread.h> if client also needs to listen concurrently.
    - **Design**: For full duplex, each client runs two threads:
        - **Send thread**: reads from stdin or sends periodic messages/updates to server.
        - **Receive thread**: loops recv() to get messages from server (relayed from other client), prints them.
    - If simpler (sequential), client can: send a message, then block on recv(), but real-time sync benefits from threads.
5. **Compile**:
6. gcc client.c -o client -pthread
7. **Test on same machine**:
    - Use different ports? For same machine, server port is fixed; run two client instances in separate terminal tabs, each connecting to 127.0.0.1:10000.
    - Terminal 1: ./multi_server
    - Terminal 2: ./client (Client 1)
    - Terminal 3: ./client (Client 2)
    - Client 1 sends message; server relays to Client 2; vice versa.
8. **Test across machines**:
    - On Server PC: ./multi_server
    - On Client PC 1: set server IP in client code, compile & run.
    - On Client PC 2: likewise.
    - Verify that each client can connect, and messages from one appear at the other.
9. **Answering Questions**:
    - *How do you handle multiple clients connecting at once? Why is multithreading necessary?*
        - Without threads, server would block on one client's recv(), cannot serve others concurrently. Threads allow each client to be served in parallel.
    - *How does the server ensure each client's data is handled separately?*
        - Each thread has its own socket descriptor. Shared data structures (client list) accessed under mutex; messages routed appropriately.
    - *Race conditions and data consistency?*
        - Protect shared client list with mutex when adding/removing; when iterating to send, lock around iteration or use thread-safe copies to avoid sending on closed socket.

4.3 Observations & Logs

- **Server startup**: “Server listening on port 10000...”
- **Client connections**: “Client connected: socket fd X”
- **Message relay**: “Received from Client 1: Hello” → “Relaying to Client 2” → Client 2 prints “Client 1 says: Hello”
- **Client disconnect**: When a client exits, server thread detects recv() ≤ 0, prints “Client disconnected,” removes from list.
- **Edge cases**: If more than MAX_CLIENTS attempt to connect, server can reject or block until a slot frees; in our lab we accept exactly two.
- **Error handling**: Check return values of socket(), bind(), listen(), accept(), pthread_create(), recv(), send(). On fatal error, log and exit or clean up gracefully.

5. Lab 5 – Task 2: Synchronization Between Multiple Clients via Server

5.1 Design & Preparation

- **Goal**: Clients communicate with each other through server; server acts as intermediary.
- Already largely covered in Task 1: message relay between clients implements synchronization of communications.
- For explicit synchronization (e.g., “START” signal before clients begin), implement:
    - After both clients connect, server sends a “START” message to both, and only after receiving “ACK” from both does server allow normal message relay.
    - Use a barrier-like mechanism: server tracks connected clients; when second connects, send “START” to both; clients wait for “START” before sending regular messages.
    - Clients: upon connection, block on recv() until “START” arrives, then enable user input thread.
- **Acknowledgment**: Server may wait for client acknowledgments: after sending “START,” server recv() from each client an “ACK” message before proceeding.

5.2 Steps Taken

1. **Extend server code** (multi_server_sync.c):
    - Maintain connected_count; after each accept(), increment count.
    - If connected_count == MAX_CLIENTS, in main thread (or dedicated coordinator thread) send "START" via each client socket.
    - Then optionally wait for each client's "ACK": in each client-handling thread, after initial greeting, recv() expected "ACK" before allowing further messages.
    - Only after both ACKs received do threads enter the normal receive-and-relay loop.
2. **Modify client code** (client_sync.c):
    - After connecting, immediately call recv() to read the "START" message from server.
    - Print "Received START from server. Sending ACK..." and send back "ACK".
    - Then enter sending/receiving threads for normal chat/data exchange.
3. **Compile**:
4. gcc multi_server_sync.c -o server_sync -pthread
5. gcc client_sync.c -o client_sync -pthread
6. **Test**:
    - Run ./server_sync.
    - Run two client instances: each ./client_sync. They connect; server logs "Both clients connected; sending START".
    - Client terminals: block until "START" arrives; print log; send "ACK"; then prompt or auto-send test messages.
    - Server logs "Received ACK from Client 1", "Received ACK from Client 2"; then normal relay begins.
7. **Answering Questions**:
    - *How to ensure server acts as intermediary without race conditions?*
        - Use mutex around shared state (connected_count, client list). Ensure "START" broadcast happens only once when both connected.
    - *How to manage data flow so all clients receive appropriate messages?*
        - In handle thread: when message received from client i, iterate over list sending to all $j \neq i$. Guard list access with mutex.
    - *Potential issues: client reads before server writes or vice versa?*
        - Clients block on initial recv() for "START"—safe because server sends only when both connected. Further messages: server only relays after ACKs. If client tries to send before START, server can ignore or buffer; code should block sending thread until after START.

5.3 Observations & Logs

- Verified that until both clients connect, no “START” is sent. If only one connects, server sits waiting.
- After both connect, immediate “START” to both; clients send ACKs; server acknowledges and enters normal mode.
- Race conditions prevented by mutex; tested with delays in client startup to ensure proper ordering.
- Logging statements included timestamps or client IDs for clarity in screenshots.
- Handled client disconnects: if a client disconnects before or after START, server handles gracefully: logs disconnect, perhaps informs other client.

6. Lab 5 – Task 3: Real-Time Data Synchronization and Coordination

6.1 Design & Preparation

- **Goal**: If one client updates some shared state, server notifies other client immediately.
- For demonstration: implement a simple shared “counter” or “status”:
    - Client can send “UPDATE: X” indicating new value.
    - Server updates its in-memory state and broadcasts “STATE: X” to all clients.
- Alternatively, a simple chat-like model already achieves real-time message sync.
- For further synchronization (e.g., consistent clock): server may timestamp updates before broadcasting.
- Could also implement periodic “heartbeat” or keepalive to ensure clients are alive.

6.2 Steps Taken

1. **Extend server thread logic**:
    - In message handler: parse incoming messages. If message indicates an update to shared state, update server’s variable under mutex.
    - Broadcast new state to all clients (including sender or optionally everyone else).
    - E.g., if client1 sends “SET TEMP=25”, server sets current_temp = 25, then loops sending “TEMP=25” to client2.
2. **Client code**:

    - After initial synchronization, client thread reads user input or simulated periodic updates (for testing).
    - On receiving a broadcast from server (e.g., “TEMP=25”), client updates displayed state.
3. **Testing**:
    - Run server and two clients on same or different machines.
    - From Client 1: send “SET VAL=10”. Client 2 terminal shows “STATE VAL=10”.
    - Vice versa. Verified immediacy.
    - Checked under network delay by adding sleep() on server before broadcast; clients see delay accordingly.
4. **Answering Questions**:
    - *How ensure updates communicated in real time?*
        - Use blocking recv() loops in client threads and immediate broadcast in server upon receive.
        - For more robust: use non-blocking I/O or select/poll for multiplexing multiple sockets if single-threaded server.
    - *What synchronization techniques to prevent data inconsistency?*
        - Server serializes updates: processes one update at a time under mutex, broadcasts sequentially.
        - Use versioning or sequence numbers if needed. In lab scope, simple single shared variable with mutex is sufficient.
        - If updates from clients arrive simultaneously: server thread serializes by handling one recv() at a time per thread; potential race if two threads update same state concurrently—use a global mutex around state update and broadcast.

6.3 Observations & Logs

- Demonstrated that when one client issues update, other sees update immediately.
- Logged timestamps in server for when update received and when broadcast sent.
- Verified no stale or duplicated broadcasts: server code ensures broadcast once per update.
- Handled edge case: if a client disconnects and reconnects, server resets synchronization or informs remaining client.

7. Testing & Debugging (Lab 5)

- **Single-machine testing**: three terminal tabs: one server, two clients. Used 127.0.0.1 for all; unique client identifiers passed to server at connect (e.g., client passes an ID on connect).
- **Multi-machine testing**: three physical or virtual machines on LAN. Used server's LAN IP in client code. Ensured firewall opened port. Verified connectivity via ping.
- **Debugging tools**: printed detailed logs in server and clients. Used netstat -tln to check listening port. If client fails to connect, checked server log and network settings.
- **Concurrency issues**: tested rapid message sends from both clients; ensured server's mutex prevented simultaneous writes to shared data structures.
- **Error cases**: client disconnects mid-sync; server logs and continues with remaining client; optionally waits for reconnection.

8. Challenges Encountered & Resolutions

- **Thread synchronization**: ensuring safe access to shared client list and state. Resolved by using pthread_mutex_t around critical sections.
- **Client startup ordering**: if client starts before server, connect() fails; resolved by adding retry logic or instructing team to start server first.
- **Network configuration**: firewall or NAT issues on multi-machine; opened ports or used same subnet.
- **Testing under time pressure**: used simple console I/O rather than GUI; test scripts automated a few messages.
- **Resource cleanup**: ensuring sockets closed and threads joined on shutdown; implemented signal handler (e.g., SIGINT) to gracefully terminate server and client threads.
- **Error handling**: always check return values; print descriptive errors.

9. Answers to Lab 5 "Questions to Consider"

- **How handle multiple clients connecting at once? Why is multi-threading necessary?**
    - Use threads so each connection is processed concurrently; without threads or non-blocking I/O, server cannot handle multiple sockets simultaneously.
- **How ensure server intermediates without race conditions?**
    - Protect shared data (client sockets list, shared state) with mutex. Use atomic operations or synchronized blocks.
- **How manage data flow so all clients receive intended messages?**

    - On receive from one client, server loops through active client sockets (excluding sender if desired) and sends the message. Under mutex to avoid list modification during iteration.
- **Ensuring real-time updates communicated to all clients?**
    - Immediately after receiving update, server broadcasts. Clients blocked on recv() thread show updates promptly.
- **What synchronization techniques to prevent data inconsistency?**
    - Use mutex around state updates; sequence numbers if multiple updates; server serializes update handling; clients can confirm receipt (ACK) if needed.
- **Potential issue if client reads before server writes or vice versa?**
    - For initial handshake ("START"): clients block on recv() until server sends. For normal messages: if server sends while client not in recv(), data remains in TCP buffer until client calls recv(). If client tries to send before handshake, server can drop or buffer until handshake done.

10. Code Quality & Best Practices

- **Modularization**: Separate functions for socket setup, client handler, broadcast, synchronization barrier.
- **Error checking**: All system calls (socket(), bind(), listen(), accept(), connect(), recv(), send(), pthread_create(), etc.) checked; on failure, print perror() and handle gracefully.
- **Constants**: Defined constants for ports, buffer sizes, maximum clients.
- **Comments**: Inline comments explaining purpose of blocks, especially synchronization logic.
- **Resource cleanup**: On exit (e.g., SIGINT), server closes all client sockets, destroys mutex, exits threads.
- **Thread safety**: Use pthread_mutex_lock()/unlock() around shared structures.
- **Logging**: Print client IDs, timestamps, message contents; helpful for report screenshots.

11. Deliverables & Documentation

- **Source Code**: Provided multi_server_sync.c, client_sync.c (with threading and handshake), multi_server.c and client.c (basic relay), plus Lab 4 codes.
- **Screenshots**: Collected terminal screenshots showing:
    - Server listening and accepting multiple clients.
    - Clients receiving "START" and sending ACK.
    - Real-time update messages relayed.

- **Report**: This document serves as the comprehensive report.
- **Screenshots & Logs**: Saved under /lab4_lab5/screenshots/.
- **README**: Contains instructions to compile and run each component.

12. Conclusion

- **Lab 5**: Built a small distributed system with three nodes, used multithreading for concurrency, implemented synchronization handshake, real-time state updates, managed race conditions, tested across machines, and documented thoroughly.
- **Teamwork**: Roles divided (server developer, client developer, tester/documentation lead), frequent communication, quick debugging under time constraint.
- **Learning Outcomes**: Deepened knowledge of socket APIs, concurrency, synchronization in distributed settings, network debugging, and robust C programming practices.