UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

Department of Computer Engineering

CEF 488:

SYSTEMS AND NETWORK PROGRAMMING

LAB 4 :

Socket Programming (Intercomputer Communication)

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

Lab 4 – Socket Programming (Intercomputer Communication)

1. Title & Identification

- **Course**: Systems and Network Programming (CEF488)
- **Lab**: Lab 4 – Socket Programming: Intercomputer Communication
- **Environment**: Ubuntu Linux (or any Linux), GCC, VS Code/Terminal, same-machine testing (localhost) and LAN testing (two machines).

2. Objective

- **Primary Goal**: Learn how to implement basic client-server communication using TCP and UDP sockets in C, understand connection-oriented vs connectionless paradigms, and test communication between processes/machines.
- **Outcome**: Develop and test:
    - A simple TCP server-client pair.
    - A simple UDP server-client pair.
    - Verify correct operation on the same host (localhost) and across two machines over LAN.

3. Tools & Software

- **Compiler**: GCC (`gcc`)
- **Editor**: VS Code / Nano / Vim
- **Terminal**: Linux terminal (Ubuntu)
- **Networking commands**: `ip addr`, `netstat` (to verify listening ports)
- **Test setup**:
    - **Single-machine**: run server and client in two terminal tabs, use `127.0.0.1`.
    - **Multi-machine**: two PCs on same LAN; server binds to INADDR_ANY; client uses server's LAN IP from `hostname -I`.

4. Lab 4 – Task 1: Simple TCP Server-Client Communication

4.1 Design & Preparation

- **Server**:
  - Create a TCP socket: `socket(AF_INET, SOCK_STREAM, 0)`.
  - Bind to port (e.g., 8080) on INADDR_ANY.
  - Listen with backlog (e.g., 3).
  - Accept one connection: `accept()`.
  - `read()` from client, process/display message.
  - `send()` a response back.
  - Close connection and socket.
- **Client**:
  - Create TCP socket.
  - Prepare `struct sockaddr_in` with server IP (`127.0.0.1` for localhost test; LAN IP for multi-machine).
  - `connect()` to server:port.
  - `send()` a message.
  - `read()` the server's response.
  - Close socket.

4.2 Steps Taken

1. **Write server code** in `tcp_server.c`.
   - Verified inclusion of `<netinet/in.h>`, `<arpa/inet.h>`, `<unistd.h>`, `<string.h>`, `<stdio.h>`, `<stdlib.h>`.
   - Error checks: after `socket()`, `bind()`, `listen()`, `accept()`.
   - Printed logs: "Server listening...", "Received: ...", "Sent response."
2. **Compile**:
3. `gcc tcp_server.c -o server`
4. **Write client code** in `tcp_client.c`.
   - Error checks: `socket()`, `connect()`.
   - Use `inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)` for localhost test.
   - Printed logs: "Client sent: ...", "Client received: ...".
5. **Compile**:
6. `gcc tcp_client.c -o client`
7. **Test on same machine**:
   - Terminal 1: `./server` → "Server listening on port 8080..."
   - Terminal 2: `./client` → Sends "Hello from client", client prints response; server prints received message.
   - Verified correct send/receive.
8. **Test across machines**:
   - On Server PC: `./server` binds to port 8080.
   - Find server IP: `hostname -I` e.g. `192.168.1.10`.

- On Client PC: in `tcp_client.c`, set `inet_pton(AF_INET, "192.168.1.10", ...)`.
- Compile & run client: `./client` → Verified connection succeeds; logs appear on both sides.
- If "Connection refused": checked firewall (disabled or allowed port 8080), correct IP.

9. **Answering Questions**:
   - *What happens if server tries to accept before listen()?*
     - `accept()` fails; must call `listen()` first to mark socket passive.
   - *How does `accept()` work, and why is it needed?*
     - After `listen()`, `accept()` blocks until a client connects; returns new socket for data exchange, allowing original socket to continue listening.
   - *Error handling*: Verified errors print via `perror()`, and server/client exit gracefully.

4.3 Observations & Logs

- On same host: near-zero latency; immediate message exchange.
- Across LAN: small delay (<1ms) if same network; tested ping and latency.
- Verified reliability: TCP ensures all bytes delivered in order.
- Edge case: If client disconnects unexpectedly, server should handle `read()` returning 0; in our simple code, server exits after one client; in extended version, loop and re-accept.

5. Lab 4 – Task 2: Simple UDP Server-Client Communication

5.1 Design & Preparation

- **UDP Server**:
  - Create UDP socket: `socket(AF_INET, SOCK_DGRAM, 0)`.
  - Bind to port (e.g., 9090) on INADDR_ANY.
  - `recvfrom()` to receive datagram from any client.
  - Print received data and client address.
  - `sendto()` response back to the source address.
  - Close socket.
- **UDP Client**:
  - Create UDP socket.
  - Prepare `struct sockaddr_in` for server IP (localhost `127.0.0.1` or LAN IP).
  - `sendto()` message to server.
  - `recvfrom()` response (bind client address implicitly by sending).
  - Close socket.

5.2 Steps Taken

1. **Write UDP server code** in `udp_server.c`.
    - Included `<arpa/inet.h>`, `<netinet/in.h>`, `<unistd.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`.
    - Error checks: `socket()`, `bind()`.
    - Print "UDP Server listening on port 9090..."
    - Call `recvfrom()` once (or loop if extended).
    - Print "Server received: ...".
    - Send back "Hello from UDP server".
    - Close socket.
2. **Compile**:
3. `gcc udp_server.c -o udp_server`
4. **Write UDP client code** in `udp_client.c`.
    - Error checks: `socket()`.
    - Set server address: `inet_addr("127.0.0.1")` for localhost.
    - `sendto()` "Hello from UDP client".
    - `recvfrom()` buffer and print.
    - Close socket.
5. **Compile**:
6. `gcc udp_client.c -o udp_client`
7. **Test on same machine**:
    - Terminal 1: `./udp_server` → "UDP Server is listening on port 9090..."
    - Terminal 2: `./udp_client` → "Client sent message...", "Client received: ...".
    - Server terminal shows received message and sent response.
8. **Test across machines**:
    - On server PC: run `./udp_server`.
    - Client PC: set server IP in code to server's LAN IP; compile & run `./udp_client`.
    - Verified messages arrive; occasional packet loss possible but minimal on LAN.
9. **Answering Questions**:
    - *Advantages/disadvantages of UDP vs TCP?*
        - UDP: low overhead, no connection setup, faster; but unreliable, unordered, no congestion control. Good for simple status messages or real-time data where occasional loss is acceptable.
        - TCP: reliable, ordered, connection-oriented, but higher overhead and latency from handshake and congestion control.
    - *Difference between `sendto()`/`recvfrom()` and `send()`/`recv()`?*
        - `sendto()`/`recvfrom()` specify target/source address per call (no connection). `send()`/`recv()` operate on an established connection (TCP).
    - *Network delays or packet loss effect on UDP?*
        - Messages may be dropped or arrive out-of-order; must implement reliability at application layer if needed.

5.3 Observations & Logs

- On same host: immediate exchange, no packet loss.
- Across LAN: also immediate; tested with ping; few milliseconds.
- Confirmed that server does not need to "listen" or "accept" before receiving.
- If multiple clients send concurrently, server can loop on `recvfrom()`.
- Tested error handling: if `recvfrom()` returns -1, print `perror()`.

6. Lab 4 – Task 3: Testing the Communication

6.1 Testing Plan

- **Same-machine testing**: as above, use `127.0.0.1`.
- **Cross-machine testing**: ensure both machines on same subnet; firewall disabled or port allowed.
- **Verification**:
  - TCP: successful connection, correct responses.
  - UDP: successful send/receive; check occasional loss by sending in a loop.
- **Tools**:
  - `netstat -tuln` to confirm server is listening on correct port.
  - `ping` to verify network connectivity.
  - `telnet server_ip 8080` (for TCP) to check if port is open.

6.2 Results & Screenshots

- **TCP**:
  - Server terminal: "Server listening on port 8080... Received: Hello from client; Sent response."
  - Client terminal: "Client sent message: Hello from client; Client received: Hello from server."
- **UDP**:
  - Server terminal: "UDP Server listening on port 9090... Received: Hello from UDP client; Sent response."
  - Client terminal: "Client sent: Hello from UDP client; Client received: Hello from UDP server."
- Verified logs captured as screenshots for report.

6.3 Challenges & Resolutions

- **Bind errors**: "Address already in use" if previous server running; resolved by killing old process or choosing different port.
- **Firewall blocks**: On multi-machine tests, disabled ufw or allowed port via `sudo ufw allow 8080/udp` etc.
- **Incorrect IP**: Mistyping server IP in client; resolved by double-checking `hostname -I`.
- **Buffer sizes**: Ensured buffers large enough (1024 bytes) for simple messages.
- **Error handling**: Added `perror()` and exit on errors in code.

7. Answers to Lab 4 "Questions to Consider"

1. **What happens if the server tries to accept connections before calling listen()?**
   - `accept()` will fail; socket must be in listening state after `listen()`.
2. **How does accept() work, and why does the server need to accept incoming connections?**
   - After `listen()`, accept blocks until a client connects; returns a new socket descriptor for that connection while original socket remains listening for further clients.
3. **Advantages/disadvantages of UDP vs TCP?**
   - See above: UDP is faster, no handshake, but unreliable; TCP is reliable and ordered but higher overhead.
4. **Difference between sendto()/recvfrom() vs send()/recv()?**
   - `sendto()/recvfrom()` used for connectionless UDP, specifying peer address on each call; `send()/recv()` used on a connected TCP socket where peer is already established.
5. **Network delays or packet loss effect on UDP?**
   - Potential message loss; for critical data must add reliability at application layer; for lab's simple messages on LAN, loss unlikely.

8. Summary & Conclusion for Lab 4

- **Functionality**: Both TCP and UDP client-server pairs worked correctly on localhost and LAN.
- **Simplicity**: Code is straightforward, well-commented, easy to follow.
- **Testing**: Verified on same machine and across two machines; captured screenshots.
- **Documentation**: Report includes code structure, steps, answers, and challenges.
- **Learning**: Understood connection-oriented vs connectionless; socket APIs; basic network troubleshooting.