IoT Device Communication System Using C++ with Yocto for Raspberry Pi 4 and QEMU

Project Overview:

In this project, you will create an IoT device communication system that transmits sensor data between multiple QEMU client devices and a central server running on Raspberry Pi 4. The communication will occur through a TCP and UDP sockets, implemented in C++ using object-oriented principles. The goal is to design both unicast and multicast server-client applications, with all socket communication abstracted and encapsulated in well-structured, object-oriented classes.

This project demonstrates practical knowledge of Yocto for building the Raspberry Pi OS, Makefile for automating the build process, and socket programming for real-time data transmission.

Project Structure:

The project is divided into two main parts:

1. **Unicast TCP Server/Client Application**
    - Server: Raspberry Pi 4
    - Client: QEMU (Emulated Environment)
2. **Multicast UDP Server/Client Application**
    - Server: Raspberry Pi 4
    - Client 1: QEMU
    - Client 2: QEMU

Deliverables:

- Full source code implementing the project.
- Yocto recipes for building the project on Raspberry Pi 4.
- Project documentation:
    - Class structure and design decisions.
    - Step-by-step guide for setting up the development environment (Raspberry Pi 4 and QEMU) with Yocto.
    - Usage instructions (commands to run the server and clients).

Part 1: Unicast TCP Socket Application

Server Side (Raspberry Pi 4)

1. **Task:**
   Create a TCP socket server that listens on a specified port for client connections.
2. **Functionality:**
    - Accept client connections from QEMU.

- Receive messages from the client and respond with an acknowledgment + current temperature.

3. **Sample Output:**
   - Server: "Client connected from IP: [X.X.X.X]."
   - Server: "Received message: [Hello, Server!]"
   - Server: "Sending acknowledgment. Current Temperature is 39 ^{o}C "

Client Side (QEMU)

1. **Task:**
   Create a TCP socket client that connects to the server on Raspberry Pi 4.
2. **Functionality:**
   - Connect to the server.
   - Send a message and receive a response.
3. **Sample Output:**
   - Client: "Connected to Server."
   - Client: "Sending message: [Hello, Server!]"
   - Client: "Received acknowledgment: [Message received.]"

Part 2: Multicast UDP Socket Application

Server Side (Raspberry Pi 4)

1. **Task:**
   Create a UDP multicast server that sends data to multiple clients.
2. **Functionality:**
   - Broadcast data to clients using multicast UDP.
   - Server can send periodic updates to all connected clients.
3. **Sample Output:**
   - Server: "Sending multicast message: [System update at 10:00 AM]."

Client 1 and Client 2 (QEMU)

1. **Task:**
   Create two separate UDP clients on QEMU that listen to multicast messages.
2. **Functionality:**
   - Each client listens for multicast messages and displays them.
3. **Sample Output (Client 1 & Client 2):**
   - Client X: "Received multicast message: [System update at 10:00 AM]."

Development Environment:

- **Yocto Setup:**
  - The application should be cross-compiled for Raspberry Pi 4 using Yocto. Yocto recipes must be created for building and deploying both the server and client applications.
  - Instructions should be provided for setting up the Raspberry Pi 4 with the generated Yocto images.
- **QEMU Setup:**
  - QEMU will emulate the client devices for both the unicast TCP and multicast UDP applications.
- **Makefile:**
  - A well-structured Makefile to automate the build process for both the server and clients on their respective environments.

Tools and Technologies:

- **C++ (OOP)**: For building the socket application using inheritance, polymorphism, abstraction, and encapsulation.
- **STL Containers**: Use of containers like std::vector, std::queue, and std::string for managing data.
- **Yocto**: For cross-compilation and deployment on Raspberry Pi 4.
- **QEMU**: For emulating client-side communication.

OOP Design:

1. **Abstract Class: Socket**
   - **Responsibilities:**
     Define common socket behaviors (send, receive, connect, shutdown) with pure virtual functions.
   - **Functions:**
     - virtual void connect() = 0;
     - virtual void send(const std::string& message) = 0;
     - virtual void receive() = 0;
     - virtual void shutdown() = 0;
2. **Derived Classes: TCPSocket and UDPSocket**
   - Inherit from Socket and implement the specific behavior for TCP and UDP communication.
3. **Abstract Class: Channel**
   - **Data Member:**
     - Socket* channelSocket; – A pointer to a Socket object (either TCPSocket or UDPSocket).
   - **Responsibilities:**
     Handle socket interactions through its Socket member (which can be TCP or UDP).

- **Functions:**
  - virtual void start() = 0;
  - virtual void stop() = 0;
  - virtual void send(const std::string& message) = 0;
  - virtual void receive() = 0;

4. **Derived Classes: ServerChannel and ClientChannel**
   - Implement specific server and client behaviors by utilizing the channelSocket to manage communication.

Thank You

Edges For Training Team