ESP32-Based TCP Server for Wireless Communication using Netcat

Abhinav K L
AM.EN.U4ECE22101

Adityakrishna Vinod
AM.EN.U4ECE22103

Ardra R Nair
AM.EN.U4ECE22112

Snigdha Sivakumar
AM.EN.U4ECE22140

Unnikrishnan Jayan
AM.EN.U4ECE22145

***Abstract*—In this project, the ESP32 microcontroller is used to demonstrate a basic TCP server configuration.In station mode, the ESP32 uses a specified SSID and password to establish a connection with a Wi-Fi network.Once connected, it launches a TCP server on port 3333 and waits for new connections.The ESP32 can be connected to and receive data from a client device, such as a PC using Netcat.To ensure two-way communication, the server logs the data it receives and returns it.To ensure seamless server operation and Wi-Fi handling, FreeRTOS is utilized for task management.Features like error checks and auto-reconnect aid in maintaining a steady connection.For learning or testing embedded networking, this setup is lightweight and excellent.It's helpful for things like remotely controlling devices or transferring sensor data.All in all, it's a useful tool for anyone working on network-based or Internet of Things projects.**

***Index Terms*—TCP/IP, IP Routing , Access Control , Client-Server Model , FreeRTOS , ESP32 , Netcat.**

I. Introduction

One of the transport-layer protocols that ensures reliable, well-structured, and error-checked transfer of data from device to device over a network is the Transmission Control Protocol (TCP). In order to demonstrate real-time communication between embedded devices and remote clients, we implemented a TCP server on the ESP32 microcontroller for this project. If the ESP32 is put into Wi-Fi Station mode, the system will join a specified wireless network and monitor port 3333 for arriving TCP connections. Data is transferred and received between a client machine e.g., a PC using Netcat—and the server. A good learning experience in computer networking, the project emphasizes building lightweight embedded networking applications, network task management with FreeRTOS, and the nuts and bolts of socket programming.

II. Objectives

The paper's objective is to design, implement and access a TCP server using ESP32 microcontroller, which basically functions in Wi-fi station mode. This is done in order to create bidirectional wireless communication with a client device, such as a PC running Netcat. The server can send data efficiently and accurately in real time, which shows ESP32's ability to manage network based communications in embedded systems.

This project uses FreeRTOS, which helps in scheduling of tasks in real time and improves the responsiveness of the system, to manage simultaneous operations. In order to ensure if it is dependable even in an unstable network environment, the implementation also incorporates strong error handling, client disconnection detection, and automatic Wi-Fi reconnection mechanisms.

This project not only enables actual real time communication, but it also provides a foundation for creating IoT applications such as remote monitoring, sensor data transmission and device control. It functions as a platform for embedded networking research and IoT system development.

III. Methodology

A. Hardware and Software Environment Setup

This project's main hardware component was the ESP32 microcontroller, which was picked because of its dual-core Xtensa LX6 microprocessor and built-in 2.4 GHz Wi-Fi (IEEE 802.11 b/g/n). The Espressif IoT Development Framework (ESP-IDF) which offers the required toolchain, APIs, and FreeRTOS integration, was used for development. Programming languages like C/C++ was mainly used. Flexible TCP client emulation was made possible by using a standard personal computer running the Netcat utility (such as the nc command) for the client side communication testing and their validation.

B. Wi-Fi Station Mode Configuration

The ESP32 was configured to operate in **Wi-Fi Station (STA) mode**. This involved programming the device to connect to an existing wireless local area network (WLAN). The configuration process included setting the **Service Set Identifier (SSID)** and the **WPA2-PSK password** of the target Wi-Fi network. Connection management employed a state machine to handle various network states (e.g., `WIFI_EVENT_STA_START`, `WIFI_EVENT_STA_GOT_IP`, `WIFI_EVENT_STA_DISCONNECTED`). Robust retry logic was implemented, leveraging exponential backoff and maximum retry attempts, to ensure the ESP32 reliably acquired an **IP address** from the network's DHCP server and established a stable connection.

C. C. TCP Server Implementation

Upon successful Wi-Fi connection and IP address acquisition, a TCP server was initialized on the ESP32. The server was configured to listen for incoming client connections on port 3333. The core techniques involved were:

- **Socket Creation:** Using `socket()` to create a TCP socket.
- **Binding:** Binding the socket to the ESP32's assigned IP address and the specified port using `bind()`.
- **Listening:** Putting the socket into a listening state to accept incoming connections using `listen()`.
- **Accepting Connections:** Utilizing `accept()` in a blocking or non-blocking manner within a dedicated task to handle new client connection requests.
- **Data Exchange Protocol:** Once a client connection was established (represented by a new client socket descriptor), data reception was managed via `recv()`. The received data was then logged to the serial console and immediately echoed back to the client using `send()`, confirming successful bidirectional transmission over the TCP stream. This echoed data served as a simple application-layer acknowledgment.

D. Concurrency and Real-Time Management with FreeRTOS

- **Wi-Fi Task**:
  - Responsible for monitoring and maintaining the Wi-Fi connection.
  - Handles reconnections upon disconnection events.
  - Typically runs at a lower priority than the data handling task.
- **TCP Server Listener Task**:
  - Continuously calls `accept()` to handle new client connections.
  - Upon accepting a new client:
    - Either creates a new dedicated task for that client, or
    - Hands over the client socket to a pool of worker tasks.
- **Client Communication Tasks (or Worker Pool)**:
  - For each connected client, a dedicated FreeRTOS task or a shared worker task from a pool handles:
    - `recv()` operations.
    - `send()` operations.
  - Ensures that one slow client or network issue does not block other concurrent client operations.
- **Inter-task Communication and Synchronization**:
  - Use of FreeRTOS queues and semaphores was considered.
  - Primarily for inter-task communication and resource synchronization.
  - However, the abstract mainly focuses on task management.

E. Error Handling and Connection Resilience

- **Socket Error Handling**:
  - Monitors return values of socket API calls (`bind`, `listen`, `accept`, `recv`, `send`).
  - Enables appropriate error logging and recovery actions.
- **Client Disconnection Detection**:
  - Checks for `recv()` returning `0` or error codes indicating client-side disconnections.
  - Triggers cleanup of client-specific resources.
  - Ensures readiness for new incoming connections.
- **Wi-Fi Reconnection Logic**:
  - Detects Wi-Fi disconnections using events like `WIFI_EVENT_STA_DISCONNECTED`.
  - Automatically triggers a reconnection sequence to the access point.
  - Typically involves a series of retries with delays.
- **Graceful Shutdown**:
  - Implements mechanisms to gracefully close sockets and free resources.
  - Handles both server-initiated and client-initiated disconnections.
  - Prevents resource leaks and ensures system stability.

IV. Results

Using the Netcat tool, the system's ability to successfully communicate between an ESP32 microcontroller and a client PC was tested. From Figure 1, the ESP32 successfully obtained the IP address 192.168.103.229 and connected to a predefined access point in Wi-Fi Station mode.

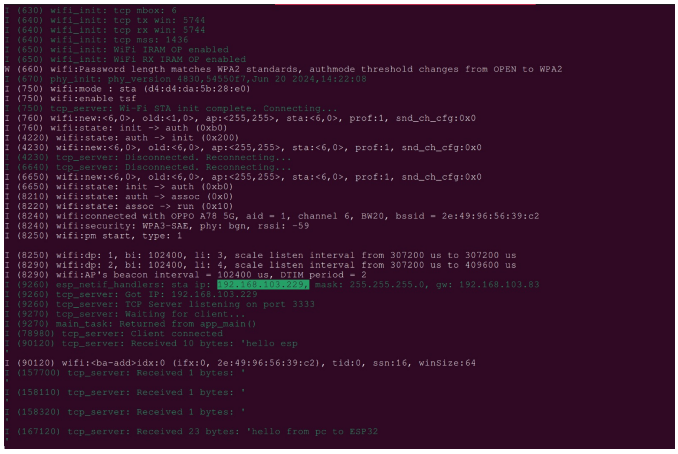

Fig. 1. ESP32 serial monitor output showing successful Wi-Fi connection, TCP server initialization, and client message reception.

After initializing, the ESP32 TCP server started listening for incoming connections on port 3333. A client terminal using Netcat established a TCP connection by executing the command '**nc 192.168.103.229 3333**'. Once connected, textual data such as "hello esp" and "hello from pc to ESP32" was sent from the client to the server. The ESP32 successfully received

and echoed the data, which was confirmed in the serial output, as seen in Figure 2.

This experiment verified the establishment of a reliable, bidirectional communication channel using TCP over Wi-Fi.

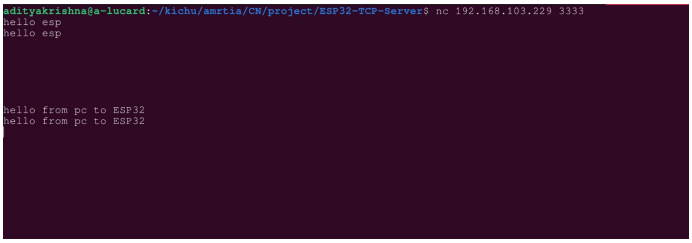

Fig. 2. Netcat terminal on PC client showing message transmission to the ESP32 server over TCP.

V. Conclusion

This project successfully implemented a simple TCP server on an ESP32 microcontroller, which enables very reliable bidirectional wireless communication with a Netcat client. By configuring the ESP32 in Wi-Fi STA mode, a stable connection was established. The server, initialized on port 3333, effectively managed real-time data exchange, with bidirectional transmission confirming data integrity. FreeRTOS integration gives asynchronous operations for Wi-Fi and server tasks, enhancing responsiveness. Comprehensive error checking and reconnection logic ensured network reliability and continuous operation. This validated setup provides an effective, lightweight solution for embedded networking applications, proving valuable for sensor data transmission, remote device control, and IoT prototyping.

References

[1] James Kurose and Keith Ross, “Computer Networking: A Top-Down Approach”, Seventh (Global) Edition, Pearson Education Ltd., 2017.

[2] Larry L. Peterson and Bruce S. Davie, “Computer Networks - A Systems Approach”, Morgan Kaufmann, Fifth Edition, 2011.