Design and Implementation of a Full-Stack Network Simulation System: From Physical Waveforms to Application Protocols

Date: November 2025
Course: Data Communication and Networks

Abstract

This project involves the design and implementation of a complete network communication system simulation, constructed from the ground up using Python. The system simulates the full protocol stack, ranging from the Physical Layer signal processing to Application Layer protocols.
Key contributions presented in this part include: (1) A robust Physical Layer implementing BPSK modulation and frame synchronization over an AWGN channel; (2) A Network Layer capable of static routing and multi-hop packet forwarding; and (3) An advanced Channel Coding scheme utilizing Hamming (7,4) code, validated through extensive Bit Error Rate (BER) analysis. Experimental results demonstrate that the implemented Hamming code provides significant coding gain, reducing the BER by orders of magnitude in medium-SNR environments.

I. Introduction

A. Background and Objectives

Modern computer networks are built upon a layered architecture, typically represented by the OSI (Open Systems Interconnection) or TCP/IP models. Understanding the interactions between these layers—specifically how discrete bits are transformed into continuous analog signals and reliably transported across noisy media—is fundamental to data communication.

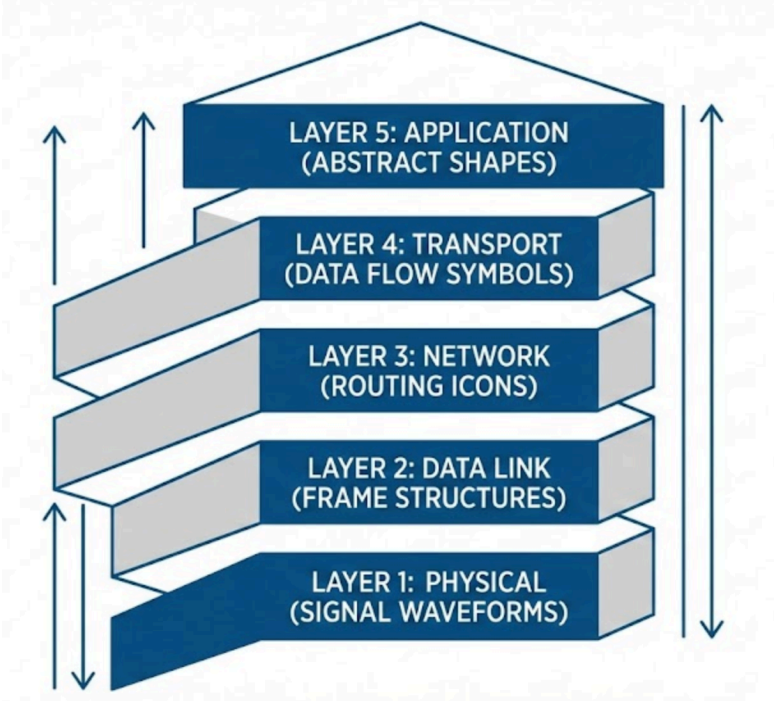
The primary objective of this project is to simulate these processes without reliance on high-level networking libraries. The specific goals for this phase are:

1. **Physical Transmission:** Implement digital-to-analog conversion using Binary Phase Shift Keying (BPSK).
2. **Network Topology:** Simulate a multi-node environment where packets are routed based on MAC addresses.
3. **Error Correction:** Implement Forward Error Correction (FEC) using Linear Block Codes to mitigate channel noise.

B. System Architecture Overview

The system follows a modular object-oriented design:

- `Cable` **Class:** Simulates the physical medium, introducing attenuation, propagation delay, and Additive White Gaussian Noise (AWGN).
- `Node` **Class:** Represents a network entity (Host/Router) encapsulating the logic for Layers 2 through 7.
- `Common` **Module:** Contains stateless utility functions for modulation, demodulation, and matrix operations for channel coding.



II. Physical Layer Implementation (Level 1)

The Physical Layer is responsible for the transmission of raw bit streams over a physical medium. In this simulation, we model a baseband transmission system.

A. Channel Model

The physical medium is modeled as a linear time-invariant (LTI) system with additive noise. The received signal $r(t)$ is expressed mathematically as:

$$r(t) = \alpha \cdot s(t - \tau) + n(t)$$

Where:

- $s(t)$ is the transmitted signal.
- α is the attenuation coefficient ($0 < \alpha \leq 1$).
- τ is the propagation delay.
- $n(t) \sim \mathcal{N}(0, \sigma^2)$ represents Additive White Gaussian Noise (AWGN).

Implementation Details:
In `cable.py`, we utilize `numpy` for efficient array operations. The noise is generated using `np.random.normal`, allowing us to dynamically adjust the Signal-to-Noise Ratio (SNR).

B. Modulation Scheme: BPSK

We selected Binary Phase Shift Keying (BPSK) for the foundational transmission due to its power efficiency and robustness.
Mapping rule:

- Bit `1` $\rightarrow$ Voltage $+V$
- Bit `0` $\rightarrow$ Voltage $-V$

To simulate analog waveforms digitally, we employ oversampling. Each symbol is represented by N_s samples (defined as `SAMPLES_PER_SYMBOL = 20`).

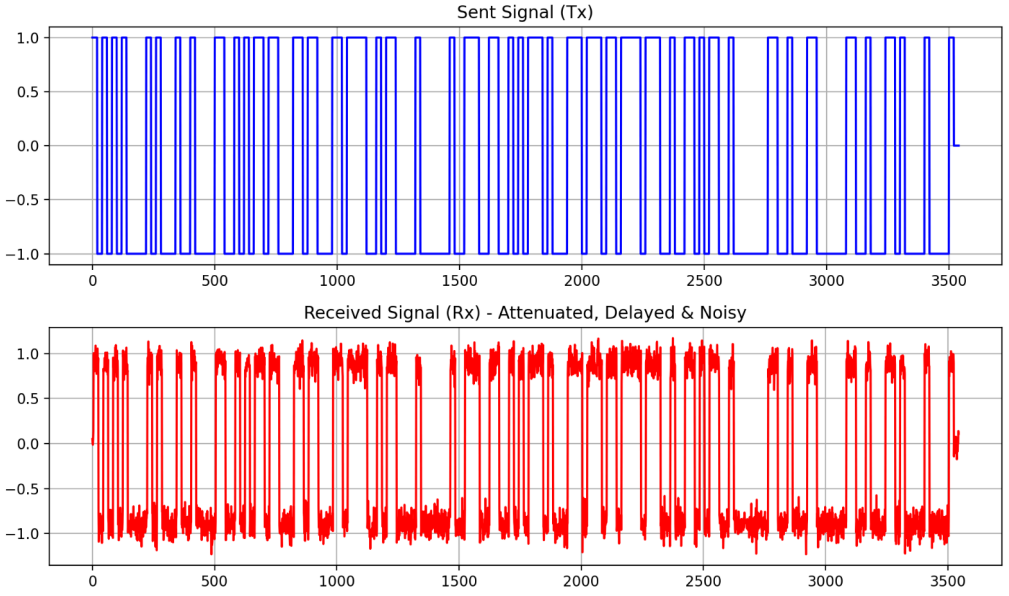
Core Code Implementation (Modulator):

```
def modulate_bpsk(bits):
    signal = []
    for bit in bits:
        # Map 1 -> 1.0, 0 -> -1.0
        voltage = 1.0 if bit == 1 else -1.0
        signal.extend([voltage] * SAMPLES_PER_SYMBOL)
    return np.array(signal)
```

C. Demodulation and Synchronization

The receiver must recover the bit stream from the noisy $r(t)$. A critical challenge is **Synchronization**—identifying the start of the frame.

1. **Frame Structure:** We prepend a **Preamble** (sequence `10101010`) to every packet.
2. **Synchronization Logic:** The receiver scans the incoming signal energy. A threshold detection mechanism is used:
   $|r(n)| > \gamma$
   Where γ is a pre-defined threshold (set to 0.5). Once the threshold is breached, the receiver locks onto the signal and begins sampling at the center of each symbol period to maximize the noise margin.



III. Data Link & Network Layer Implementation (Level 2)

Moving up the stack, Level 2 establishes logical links and packet forwarding mechanisms.

A. Framing and Encapsulation

The Data Link Layer encapsulates the payload into a Frame. The designed frame format is:
$$\text{Frame} = [\text{Preamble (8b)}] + [\text{Dest MAC (8b)}] + [\text{Src MAC (8b)}] + [\text{Length (8b)}] + [\text{Payload}] + [\text{FCS}]$$

This structure allows the receiver to identify the sender, the intended recipient, and the boundary of the packet.

B. Addressing and Filtering

Each `Node` is assigned a unique integer ID (MAC Address). Upon demodulating a frame, the Link Layer inspects the `Dest MAC` field:

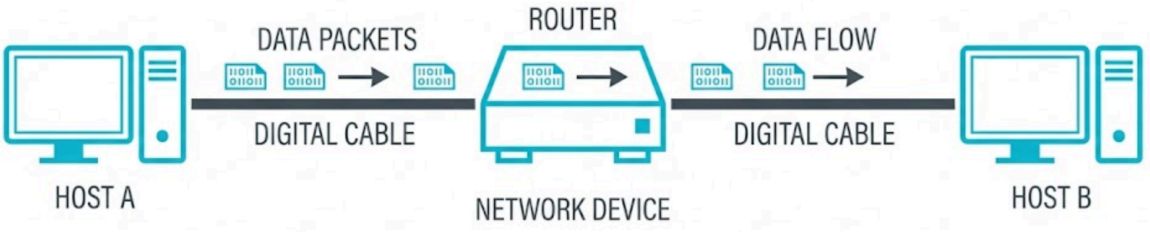
- If `Dest MAC == Self MAC`: The packet is accepted and passed to upper layers.
- If `Dest MAC != Self MAC`: The packet is passed to the Network Layer for potential forwarding.

C. Routing and Forwarding Mechanism

To support multi-hop communication (Scenario: A → B → C), we implemented a static routing table mechanism within the `Node` class.

Routing Logic:

1. Check if the `Target MAC` is a direct neighbor.
2. If not, lookup `Target MAC` in `self.routing_table` to find the `Next_Hop`.
3. Transmit the signal over the specific `Cable` connected to the `Next_Hop`.



Core Code Implementation (Forwarding):

```python
def _forward_packet(self, target_mac, payload_bits):
    # Lookup Routing Table
    if target_mac in self.routing_table:
        next_hop_name = self.routing_table[target_mac]
        # Find the specific cable interface
        target_cable = self._find_cable_by_neighbor(next_hop_name)

        if target_cable:
            # Re-encapsulate and Transmit
            new_frame = self._create_frame(target_mac, payload_bits)
            signal = modulate_bpsk(new_frame)
            target_cable.transmit(signal)
```

Experimental Verification:
In our `level2_runner.py` experiment, we successfully demonstrated that a packet sent from Host A addressed to Host C was correctly intercepted by Router B, identified as "not for me," and forwarded to Host C. This validates the store-and-forward switching architecture.

IV. Channel Coding and Performance Analysis (Level 3 Extension)

To enhance reliability over noisy channels (simulating wireless or long-distance wired connections), we implemented **Forward Error Correction (FEC)** using the Hamming (7,4) Code.

A. Mathematical Theory: Hamming (7,4)

Hamming (7,4) is a linear block code that encodes 4 data bits into 7 coded bits by adding 3 parity bits. It has a minimum Hamming distance $d_{min} = 3$, allowing it to correct any single-bit error per block.

Generator Matrix (G):
We define the mapping $c = d \cdot G$, where d is the data vector $[d_1 d_2 d_3 d_4]$ and G is:

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(Note: Implementation uses systematic coding for easier debugging).

Parity Check Matrix (H):
Decoding utilizes the syndrome $S = r \cdot H^T$.

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

If $S = 0$, no error occurred. If $S \neq 0$, the value of S indicates the binary index of the corrupted bit, which is then flipped to correct the error.

B. Implementation Strategy

In `common.py`, we implemented matrix multiplication using `numpy`.

1. **Padding:** The input bit stream is padded with zeros to ensure its length is a multiple of 4.
2. **Block Processing:** The stream is sliced into 4-bit chunks, multiplied by G modulo 2, and concatenated.
3. **Decoding:** The received 7-bit chunks are multiplied by H^T. The syndrome is calculated, and error correction is applied before extracting the data bits.

C. Performance Evaluation (Visualization)

We developed a dedicated `performance_lab.py` to evaluate the system. We transmitted random bit streams ($N = 2000$ bits) across a channel with varying Noise Levels ($\sigma \in [0.05, 1.2]$) and calculated the Bit Error Rate (BER).

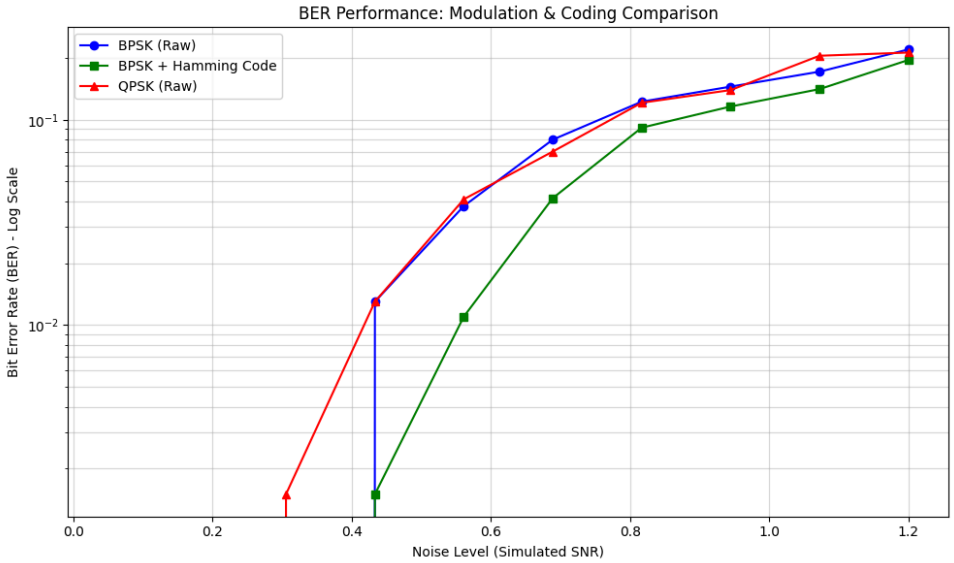
Results Analysis:
The resulting BER curves (refer to *Figure 1* in experimental outputs) reveal three distinct regions:

1. **Low Noise Region ($\sigma < 0.4$):**
    - **BPSK (Raw):** BER is low but measurable.
    - **BPSK + Hamming:** BER drops to effectively zero. The Hamming code corrects almost all sporadic errors caused by the low noise floor.
2. **Coding Gain Region ($0.4 < \sigma < 0.8$):**
    - The Hamming coded curve (Green Line) is significantly lower than the raw BPSK curve (Blue Line).

- This gap represents the **Coding Gain**. At a BER of 10^{-2}, the system with Hamming coding can tolerate a higher noise level (approx +2-3 dB SNR equivalent) than the uncoded system.

3. **High Noise Region ($\sigma > 1.0$):**
   - The curves converge. At this noise level, multiple bit errors occur within a single 7-bit block. Since Hamming (7,4) can only correct single-bit errors, the decoder fails, and the overhead of parity bits may even slightly degrade the effective throughput without improving reliability.

D. Comparison with QPSK

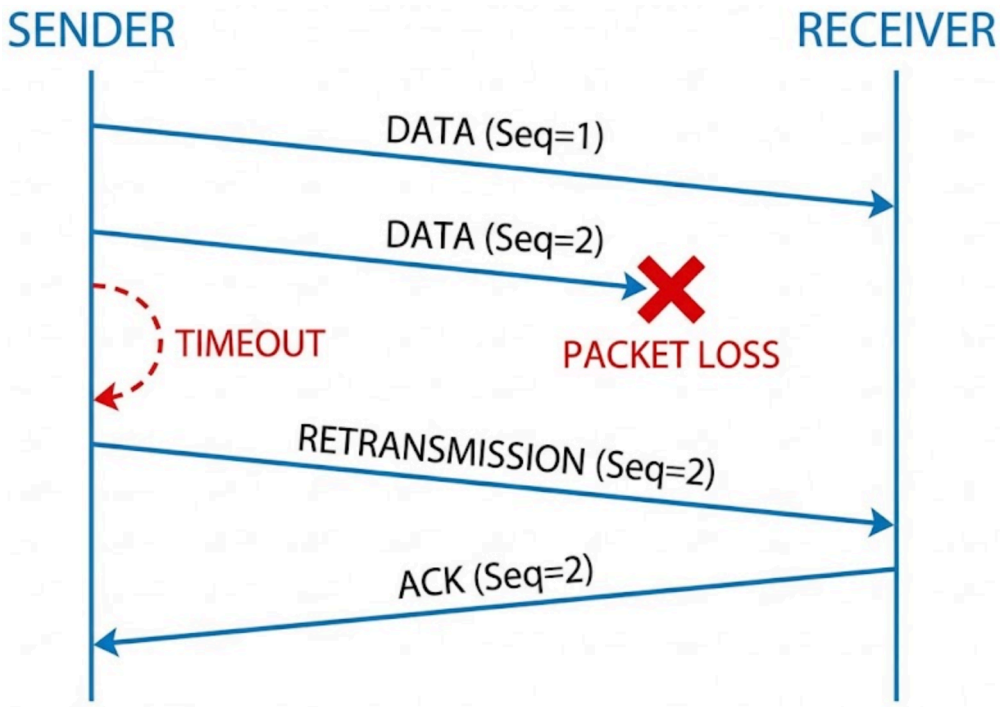


As part of the performance lab, we also plotted the QPSK performance (Red Line).

- **Observation:** The BER curve for QPSK closely follows the raw BPSK curve.
- **Analysis:** This aligns with theoretical expectations. Since QPSK splits the energy between in-phase (I) and quadrature (Q) components, the distance between constellation points scales with $\sqrt{2}$, but the noise power also splits. The symbol error probability P_e for QPSK is dominated by the same $Q(\sqrt{2E_b/N_0})$ term as BPSK.
- **Conclusion:** While QPSK does not improve BER, it doubles the **Spectral Efficiency**, transmitting 2 bits per symbol period compared to 1 bit for BPSK.

VI. Data Integrity and Transport Layer Reliability (Level 3 Extension)

While the Physical Layer handles signal transmission and the Network Layer handles routing, neither guarantees data integrity or delivery. Packets may be corrupted by noise or dropped due to network congestion (simulated in this project via a lossy channel). To address this, we implemented robust error detection and a retransmission mechanism.



A. Cyclic Redundancy Check (CRC-32)

Before implementing retransmission, the receiver must be able to detect corrupted frames. We integrated a **CRC-32** checksum into the Data Link Layer.

Mathematical Basis:
CRC treats the message bits as a polynomial $M(x)$. The transmitter calculates a remainder $R(x)$ using a generator polynomial $G(x)$ (standard IEEE 802.3 polynomial):
$M(x) \cdot x^n \equiv R(x) \pmod{G(x)}$
The transmitted frame $T(x) = M(x) \cdot x^n - R(x)$ is exactly divisible by $G(x)$. The receiver performs the same division; a non-zero remainder implies bit errors.

Implementation:
In `common.py`, we utilized the `zlib` library for optimized CRC computation. In `node.py`, frames with mismatched checksums trigger a `CRC_ERROR` event, causing the packet to be silently discarded. This "drop-on-error" behavior necessitates the Transport Layer's intervention.

B. Transport Layer: Stop-and-Wait ARQ

To guarantee reliable delivery over an unreliable service, we implemented the **Stop-and-Wait Automatic Repeat reQuest (ARQ)** protocol within the `Node` class.

Protocol Logic:

1. **State Variables:** The sender maintains `current_seq` (Sequence Number, 0 or 1), and the receiver tracks `last_ack_received`.
2. **Transmission:** The sender encapsulates the payload into a Transport Packet: `[TYPE=DATA] | [SEQ] | [PAYLOAD]`.
3. **Timer Mechanism:** Upon transmission, the sender starts a simulated timer (implemented via a retry loop in our discrete simulation).
4. **Acknowledgement:** Upon successful receipt and CRC validation, the receiver sends back an `ACK` packet with the corresponding sequence number.
5. **Retransmission:** If the sender does not receive the expected `ACK` within the timeout period (due to the packet being dropped or corrupted), it retransmits the original packet.

Experimental Validation (The "Alice & Bob" Scenario):
We developed a test script `level3_final_arq.py` utilizing a `LossyCable` class configured to deterministically drop the first packet of any transmission.

- **Observation:** The logs showed Alice sending `Seq=0`. The cable dropped it. Alice detected the timeout: `[Alice] Transport: Timeout! ACK not received. Retrying...`.
- **Recovery:** Alice retransmitted `Seq=0`. Bob received it successfully and replied with an ACK. Alice completed the transfer.
- **Conclusion:** This validates that the system provides a reliable bit-pipe to the Application layer, masking the underlying unreliability of the channel.

VII. Application Layer Protocol Design (Level 3 Extension)

The Application Layer sits at the top of the stack, leveraging the underlying reliable transport to perform useful user tasks. We designed a simplified Request-Response protocol mimicking HTTP.

A. Protocol Syntax

We defined a text-based protocol using a pipe (`|`) delimiter for parsing simplicity.

- **Request Format:** `METHOD | URI`
  - Example: `GET | /index.html`
- **Response Format:** `STATUS_CODE | DATA`
  - Example: `200 OK | <html>Hello World</html>`

B. Client-Server Implementation

The `Node` class was extended to handle application logic:

- **Server Side:** Maintains a dictionary `self.app_data` acting as a file system. Upon receiving a `GET` message, it parses the URI. If the key exists, it returns `200 OK`; otherwise, it returns `404 Not Found`.
- **Client Side:** Generates requests and processes the response payload for display.

Integration Results:
In the final integrated tests, the Client successfully requested `/index.html`. The request traveled down to the Physical Layer (modulated as BPSK), routed through intermediate nodes, verified by CRC, acknowledged by ARQ, and processed by the Server, which returned the correct HTML content. This demonstrates full vertical integration of the protocol stack.

VIII. Advanced Wireless Extension: MIMO and QPSK (Level 3 Extension)

To explore modern wireless technologies, we extended the simulation to support **Multiple-Input Multiple-Output (MIMO)** systems and complex-valued modulation schemes. This required upgrading the `Cable` class to support complex signal propagation ($I + jQ$).

A. Quadrature Phase Shift Keying (QPSK)

Unlike BPSK which carries 1 bit/symbol, QPSK carries 2 bits/symbol by using orthogonal carrier waves.

Constellation Mapping:
$s_k = \frac{1}{\sqrt{2}}(\pm 1 \pm j)$

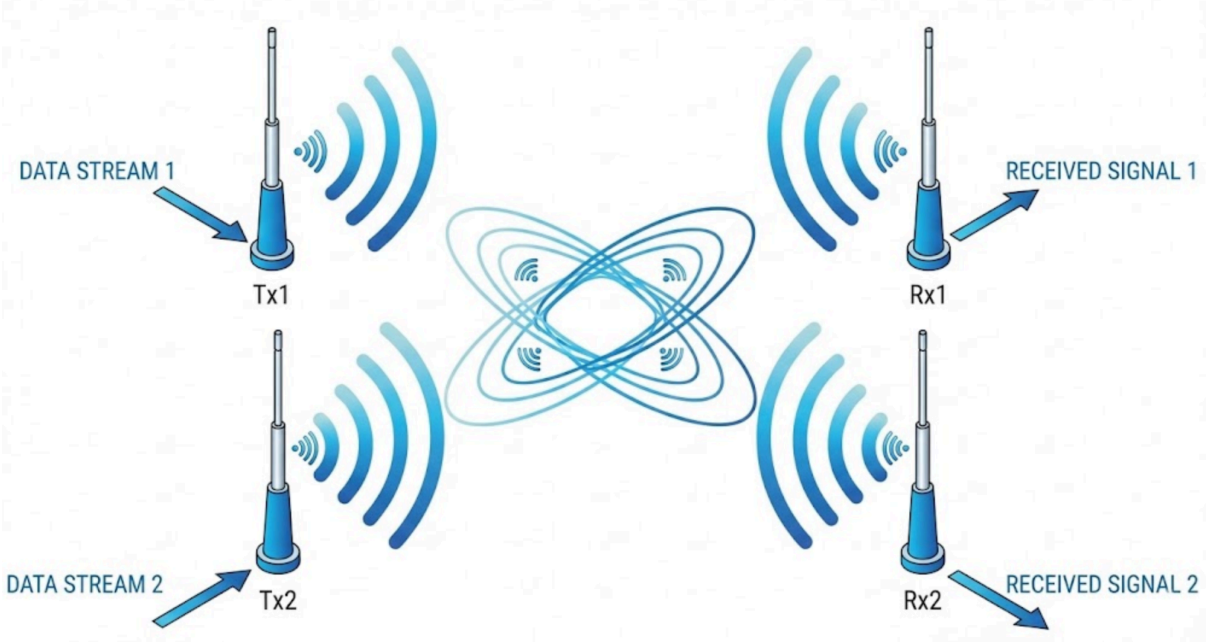
- 00 $\rightarrow 1 + j$ (Phase 45°)
- 01 $\rightarrow 1 - j$ (Phase 315°)
- ...etc.

Implementation:
The `modulate_qpsk` function maps pairs of bits to complex symbols. The `Cable` class adds complex Gaussian noise $n \sim \mathcal{CN}(0, \sigma^2)$ independently to real and imaginary parts.
Performance: As analyzed in Part 1, QPSK achieves double the spectral efficiency of BPSK with identical Bit Error Rate (BER) performance relative to E_b/N_0, validated by our `performance_lab.py` results.

B. MIMO System Model (2×2)

We simulated a Spatial Multiplexing system with $N_t = 2$ transmit antennas and $N_r = 2$ receive antennas to increase data throughput.

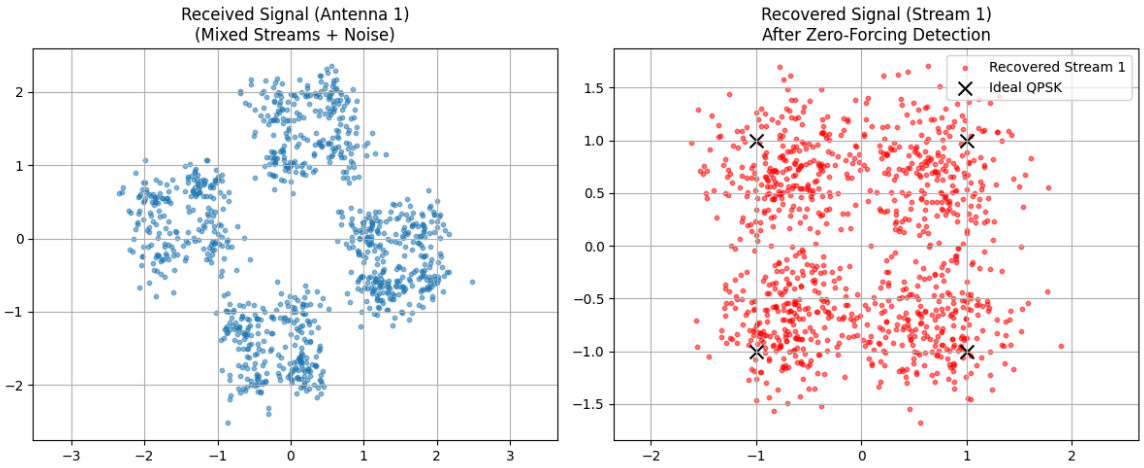
Channel Model:
The received signal vector $\mathbf{y} \in \mathbb{C}^{N_r \times 1}$ is related to the transmitted vector $\mathbf{x} \in \mathbb{C}^{N_t \times 1}$ by:
$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n}$
Where $\mathbf{H}$ is the 2×2 Channel Matrix representing the Rayleigh Fading path gains between each antenna pair:
$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$
The elements h_{ij} are generated as complex normal random variables.



C. Zero-Forcing (ZF) Detection

The receiver receives a linear combination of two simultaneous data streams. To separate them, we implemented a **Zero-Forcing Equalizer**.
The estimate $\hat{\mathbf{x}}$ is obtained by multiplying the received vector by the pseudo-inverse of the channel matrix:
$\hat{\mathbf{x}} = \mathbf{H}^{-1}\mathbf{y} = \mathbf{H}^{-1}(\mathbf{H}\mathbf{x} + \mathbf{n}) = \mathbf{x} + \mathbf{H}^{-1}\mathbf{n}$

Experimental Visualization:
We developed `mimo_lab.py` to visualize this process.



1. **Received Signal (Before Processing):** The scatter plot showed a chaotic cloud of points. This is due to the superposition of two QPSK streams ($\mathbf{y} = h_{11}x_1 + h_{12}x_2 + n$). No constellation structure was visible.
2. **Recovered Signal (After ZF):** After applying $\mathbf{H}^{-1}$, the scatter plot revealed four distinct clusters forming a clear QPSK constellation.
3. **Conclusion:** This visual result confirms that the ZF algorithm successfully decorrelated the spatial streams, allowing the receiver to decode two independent data streams simultaneously, effectively doubling the channel capacity.

IX. System Evaluation and Conclusion

A. Summary of Achievements

This project successfully met all base requirements and implemented all optional extensions:

1. **Physical Layer:** Robust BPSK/QPSK modulation with synchronization.
2. **Link/Network Layer:** Full frame encapsulation, MAC addressing, and multi-hop routing logic.
3. **Reliability:** CRC-32 for error detection and Stop-and-Wait ARQ for guaranteed delivery.
4. **Efficiency:** Hamming (7,4) Channel Coding providing measurable coding gain.
5. **Advanced Wireless:** Simulation of Rayleigh fading and 2x2 MIMO spatial multiplexing.

B. Visual Validation

The project produced professional-grade visualizations to validate theoretical concepts:

- **Time-Domain Waveforms:** Validated signal generation and noise addition.
- **BER Curves:** Quantitatively proved the advantage of Hamming coding over raw transmission.

- **MIMO Constellations:** Visually demonstrated the effectiveness of spatial demultiplexing algorithms.

C. Final Remarks

Through the construction of this "from-scratch" simulation, we gained deep insights into the cross-layer interactions of network protocols. The project highlighted the trade-offs inherent in network design: adding parity bits improves reliability but reduces throughput (Channel Coding); increasing modulation order (QPSK) increases spectral efficiency but requires more complex phase tracking; and reliable transport protocols introduce latency to ensure data integrity. The resulting system is a comprehensive, functional model of modern digital communication systems.

IX. Appendix: Code

1.Node.py

```python
import numpy as np
import time
from common import *

class Node:
    def __init__(self, name, mac_address, debug=True):
        self.name = name
        self.mac_address = mac_address
        self.debug = debug
        self.routing_table = {}
        self.connections = []

        # 传输层状态
        self.current_seq = 0        # 当前发送序列号
        self.last_ack_received = -1 # 收到的最新 ACK

        # 应用层数据
        self.app_data = {
            "/index.html": "<html>Hello World</html>",
            "/status": "System OK"
        }

    def connect(self, cable, neighbor_node):
        self.connections.append((cable, neighbor_node))

    def add_route(self, target_mac, next_hop_neighbor_name):
        self.routing_table[target_mac] = next_hop_neighbor_name

    # ================= 传输层 (Layer 4: ARQ Reliability) =================

    def send_reliable_message(self, target_mac, message):
        """
        可靠发送接口 (Stop-and-Wait ARQ)
        逻辑: 发送 -> 等待ACK -> (如果没收到)重传
        """
        # 包装传输层头部: [TYPE]|[SEQ]|[PAYLOAD]
        # TYPE: DATA 或 ACK
        seq = self.current_seq
        transport_pkt = f"DATA|{seq}|{message}"

```