a down-top LAN simulating system

LinYiFei

github.com/Apollo0317/network-communication-project

2025.12.6

Contents

1. Introduction

1.1 Background

This system is a simulation of the real world local-area network. In this system, we:

- build a simplified, **asynchronous**, LabView-like simulation engine which treats everything a game entity and update them in a finite loop in every frame;

- Divide networks at different levels into several individual protocol stack objects which is fairly easy to add, remove and expand, achieves a high degree of decoupling of network protocol stacks;

- implement full stack that a real-world LAN would use, including physics layer, mac layer, transport layer and application layer.

For the task part, we realize the basic part which is composed of level1 and level2, in level3, we implement:

- a tcp-like reliable Transport layer with port multipliexing function and a POSIX-like socket interface

- an classic channel coding method, (7,4) Hamming code and CRC32 checksum, which provide the ability of error decection and correction in physics layer and mac layer

- an useful application layer protocol, HTTP, with a series of standard client side interface including GET, and routing logic in sever side.

Next we will introduce all the part we mentioned above with detailed description, code and figures.

2. Feature

2.1 Simulation Engine

The simulation engine is the core of this system, which adopts the **Game Loop** pattern widely used in game development and real-time systems. This design allows us to simulate asynchronous network communication in a deterministic and reproducible manner.

2.1.1 Design Philosophy

Unlike event-driven simulators (e.g., ns-3), our engine uses a **tick-based** approach where time advances in fixed increments. Each tick represents a configurable physical time unit (default: 1 microsecond). This design has several advantages:

Feature	Game Loop (Ours)	Event-Driven
Determinism	High (reproducible)	Depends on event ordering
Parallelism	Natural support	Requires careful synchronization
Physical accuracy	Tick-level precision	Event-level precision
Complexity	Simple to implement	More complex

2.1.2 Core Architecture

The engine maintains a list of `SimulationEntity` objects and calls their `update()` method every tick:

```python
#core/simulator.py
class PhySimulationEngine:
    def __init__(self, time_step_us: float = 1.0):
        self.time_step_us = time_step_us
        self.entities: list[SimulationEntity] = []
        self.current_tick = 0

    def register_entity(self, entity: SimulationEntity):
        self.entities.append(entity)

    def run(self, duration_ticks: int):
        for tick in range(duration_ticks):
            self.current_tick = tick
            for entity in self.entities:
                entity.update(tick)
```

All network components (nodes, switches, cables) inherit from `SimulationEntity` and implement their own `update()` logic. This allows heterogeneous entities to coexist and interact naturally.

2.1.3 Entity Base Class

```python
#core/simulator.py
class SimulationEntity(abc.ABC):
    def __init__(self, name: str):
        self.name = name
        self.current_tick = 0

    def update(self, tick: int):
        """Called every simulation tick"""
        self.current_tick = tick

    def reset(self):
        """Reset entity state for re-simulation"""
        self.current_tick = 0
```
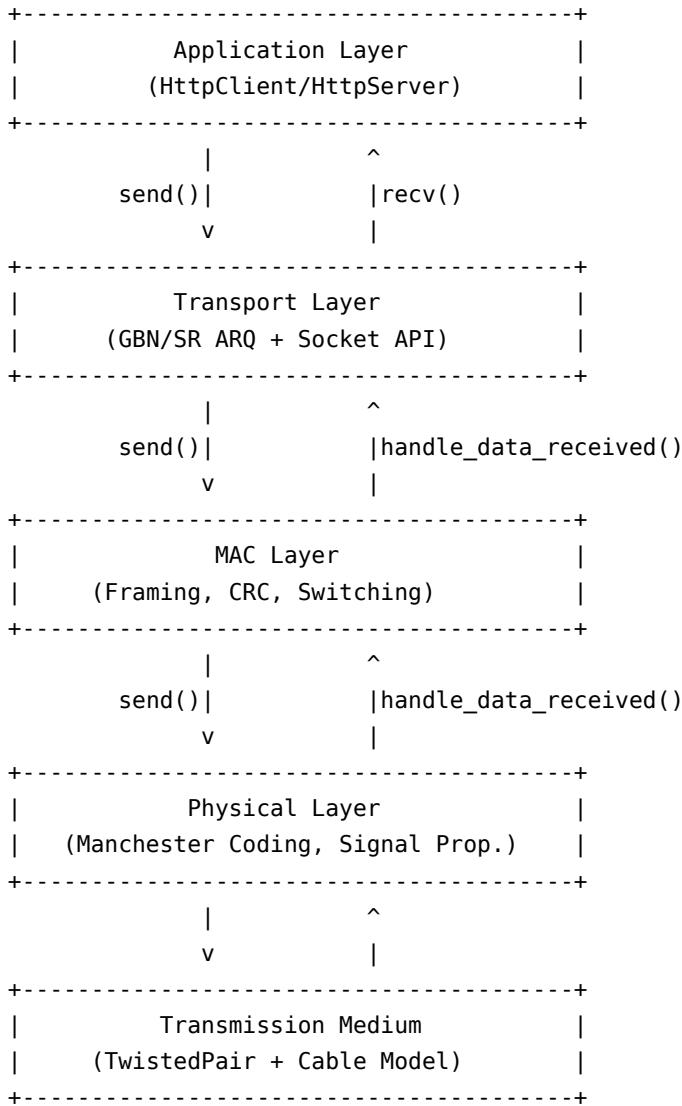
Entities can also signal early termination by returning `1` from `update()`, which is useful for testing scenarios where we want to stop simulation when certain conditions are met (e.g., all data transmitted).

2.2 Protocol Stack Architecture

2.2.1 Layered Design

Following the OSI model and real-world network implementations, we designed a modular protocol stack with clear layer separation:

```
+-----------------------------------------+
|            Application Layer             |
|         (HttpClient/HttpServer)          |
+-----------------------------------------+
            |             ^
      send()|             |recv()
            v             |
+-----------------------------------------+
|            Transport Layer               |
|        (GBN/SR ARQ + Socket API)         |
+-----------------------------------------+
            |             ^
      send()|             |handle_data_received()
            v             |
+-----------------------------------------+
|              MAC Layer                    |
|         (Framing, CRC, Switching)         |
+-----------------------------------------+
            |             ^
      send()|             |handle_data_received()
            v             |
+-----------------------------------------+
|             Physical Layer                |
|      (Manchester Coding, Signal Prop.)    |
+-----------------------------------------+
            |             ^
            v             |
+-----------------------------------------+
|            Transmission Medium            |
|        (TwistedPair + Cable Model)        |
+-----------------------------------------+
```

2.2.2 ProtocolLayer Base Class

Each protocol layer inherits from `ProtocolLayer`, which provides a unified interface for data encapsulation and transmission:

```python
#core/ProtocolStack.py
class ProtocolLayer(SimulationEntity):
    def __init__(self, lower_layer, simulator, name):
        super().__init__(name)
        self.lower_layer = lower_layer
        self.upper_layer = None
```

```python
        self.rx_queue = deque()

        # Automatically link layers
        if self.lower_layer:
            self.lower_layer.upper_layer = self

    def send(self, data):
        """Downward data flow: encapsulate and pass to lower layer"""
        encapsulated = self.Encapsulate(data)
        if self.lower_layer:
            self.lower_layer.send(encapsulated)
        else:
            self.send_to_phy(encapsulated)

    def handle_data_received(self, data):
        """Upward data flow: decapsulate and pass to upper layer"""
        decapsulated = self.Dencapsulate(data)
        if decapsulated is None:
            return  # Drop invalid packet
        if self.upper_layer:
            self.upper_layer.handle_data_received(decapsulated)
        else:
            self.rx_queue.append(decapsulated)

    def recv(self):
        """Top layer interface to retrieve received data"""
        if self.rx_queue:
            return self.rx_queue.popleft()
        return None

    @abc.abstractmethod
    def Encapsulate(self, data):
        """Add layer-specific header/trailer"""
        pass

    @abc.abstractmethod
    def Dencapsulate(self, data):
        """Remove and parse layer-specific header/trailer"""
        pass
```

2.2.3 Key Design Decisions

1. Bidirectional Layer Linking

Each layer maintains references to both upper and lower layers, enabling: - Downward flow via `send() -> Encapsulate() -> lower_layer.send()` - Upward flow via `handle_data_received() -> Dencapsulate() -> upper_layer.handle_data_received()`

2. Non-blocking Operations

All I/O operations are non-blocking to fit the tick-based simulation model: - `recv()` returns `None` if no data available (instead of blocking) - `send()` queues data for transmission (processed in future ticks)

3. Decoupling via Abstract Methods

Each layer implements its own `Encapsulate()` and `Dencapsulate()` methods, allowing: - Independent development and testing of each layer - Easy replacement of protocols (e.g., swap GBN for SR ARQ) - Clear separation of concerns

2.2.4 Data Flow Example

When an HTTP client sends a GET request, data flows through the stack:

```
HTTP Client: "GET /index.html"
     |
     v [Encapsulate: add HTTP headers]
TCP Layer: [TCP Header] + HTTP payload
     |
     v [Encapsulate: add MAC frame]
MAC Layer: [Preamble][Len][Src][Dst][TCP segment][CRC]
     |
     v [Encapsulate: Manchester encoding]
PHY Layer: [Encoded bit stream]
     |
     v [Transmit through cable]
   Cable: [Attenuated + noisy signal]
     |
     v [Receive and decode]
PHY Layer: [Decoded bytes]
     |
     v [Dencapsulate: verify CRC, extract payload]
MAC Layer: [TCP segment]
     |
     v [Dencapsulate: process TCP header, manage session]
TCP Layer: [HTTP payload]
     |
     v [Dencapsulate: parse HTTP request]
HTTP Server: Handle "GET /index.html"
```

This layered architecture makes the system highly modular and extensible. Adding a new protocol layer (e.g., IP routing) only requires implementing `Encapsulate()` and `Dencapsulate()` methods without modifying existing layers.

3. Realization

3.1 Physical Layer

In the Physical layer segment, we mainly:

- Implement the `Modulator` and `DeModulator` classes to convert byte streams into analog signals (`numpy.ndarray`) and vice versa

- Realize channel encoder and decoder using (7,4) Hamming code for error correction

3.1.1 Modulator (Level1)

We implement a **16-QAM (Quadrature Amplitude Modulation)** scheme, which maps 4 bits to one symbol using both amplitude and phase.

Constellation Mapping   16-QAM uses a 4x4 constellation with Gray coding. Each symbol carries 4 bits: - First 2 bits determine the In-phase (I) component - Last 2 bits determine the Quadrature (Q) component

```python
# phy/modulator.py
@staticmethod
def generate_QAM_mapping(order: int = 16) -> dict[str, list]:
    """Generate 16-QAM constellation mapping with Gray coding"""
    mapping = {}
    for code in range(order):
        bit_list = [(code >> i) & 1 for i in range(3, -1, -1)]
        I_bit, Q_bit = bit_list[:2], bit_list[2:]

        # Gray coding: 00->-3, 01->-1, 11->1, 10->3
        if I_bit == [0, 0]:   I_out = -3
        elif I_bit == [0, 1]: I_out = -1
        elif I_bit == [1, 1]: I_out = 1
        else:                 I_out = 3

        # Same mapping for Q channel
        if Q_bit == [0, 0]:   Q_out = -3
        elif Q_bit == [0, 1]: Q_out = -1
        elif Q_bit == [1, 1]: Q_out = 1
        else:                 Q_out = 3

        mapping[str(bit_list)] = [I_out, Q_out]
    return mapping
```

The constellation diagram looks like:

```
Q
^
|  (-3,3)  (-1,3)  (1,3)  (3,3)
|  (-3,1)  (-1,1)  (1,1)  (3,1)
|  (-3,-1) (-1,-1) (1,-1) (3,-1)
|  (-3,-3) (-1,-3) (1,-3) (3,-3)
+--------------------------------> I
```

Upconversion Process   The modulator performs quadrature modulation to shift baseband symbols to carrier frequency:

$$s(t) = I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t)$$

```python
# phy/modulator.py
def QAM_UpConverter(self, symbols: list[list], debug=False) -> np.ndarray:
    """Modulate baseband symbols to carrier frequency"""
    # Convert to complex symbols
    complex_symbols = np.array([complex(s[0], s[1]) for s in symbols])

    # Samples per symbol
    sps = self.sample_rate / self.symbol_rate
    I_baseband = np.real(complex_symbols)
    Q_baseband = np.imag(complex_symbols)

    # Pulse shaping (rectangular)
    I_n = np.repeat(I_baseband, repeats=sps)
    Q_n = np.repeat(Q_baseband, repeats=sps)

    # Generate carriers
    n = np.arange(len(I_n))
    carrier_cos = np.cos(2 * np.pi * self.fc * n / self.sample_rate)
    carrier_sin = np.sin(2 * np.pi * self.fc * n / self.sample_rate)

    # Quadrature modulation
    qam_signal = I_n * carrier_cos - Q_n * carrier_sin
    return qam_signal
```

Channel Estimation For the first transmission, 4 training symbols are prepended for channel estimation at the receiver:

```python
# phy/modulator.py
def QAM(self, byte_data: bytes) -> list:
    # ... symbol mapping ...

    # Add training symbols on first transmission
    if not self.has_estimated:
        train_symbols = [[1, 3], [3, 1], [-1, -3], [-3, -1]]
        symbols = train_symbols + symbols
        self.has_estimated = True
    return symbols
```

Demodulation Process The demodulator performs the inverse operations:

1. **Downconversion**: Multiply by local carriers to recover I/Q baseband
2. **Low-pass filtering**: Remove high-frequency components using Butterworth filter
3. **Symbol sampling**: Average samples within each symbol period
4. **Minimum distance detection**: Map received symbols to nearest constellation point

```python
# phy/modulator.py
def Detect_Symbol(self, symbols: list[list]) -> list[int]:
    """Symbol detection using minimum Euclidean distance"""
    bit_list = []

    # Channel estimation using training symbols
    if not self.has_estimated:
        std_train_symbols = [[1, 3], [3, 1], [-1, -3], [-3, -1]]
        recv_train_symbols = symbols[:4]
        symbols = symbols[4:]

        # Calculate amplitude loss coefficient
        std_power = sum(s[0]**2 + s[1]**2 for s in std_train_symbols)
        recv_power = sum(s[0]**2 + s[1]**2 for s in recv_train_symbols)
        self.amplitude_loss = math.sqrt(std_power / recv_power)
        self.has_estimated = True

    # Compensate channel loss and detect symbols
    for symbol in symbols:
        fixed_symbol = [s * self.amplitude_loss for s in symbol]
        # Find nearest constellation point
        idx = np.argmin([distance(fixed_symbol, ref) for ref in constellation])
        bit_list.extend(mapping[idx])

    return bit_list
```

3.1.2 Channel Coding (Level3 Part B)

We implement **(7,4) Hamming Code** for forward error correction (FEC). This code can: - Detect up to 2-bit errors - Correct single-bit errors

Hamming Code Theory The (7,4) Hamming code encodes 4 data bits into 7 bits by adding 3 parity bits. The encoding uses a generator matrix G^T:

$$\mathbf{c} = G^T \cdot \mathbf{m} \mod 2$$

where $\mathbf{m}$ is the 4-bit message and $\mathbf{c}$ is the 7-bit codeword.

```python
# phy/Coding.py
class Hamming:
    """Hamming (7,4) error correction code implementation"""

    # Generator matrix (transposed)
    _gT = numpy.matrix([
        [1, 1, 0, 1],  # p1
        [1, 0, 1, 1],  # p2
        [1, 0, 0, 0],  # d1
```

```python
        [0, 1, 1, 1],  # p3
        [0, 1, 0, 0],  # d2
        [0, 0, 1, 0],  # d3
        [0, 0, 0, 1],  # d4
    ])

    # Parity check matrix
    _h = numpy.matrix([
        [1, 0, 1, 0, 1, 0, 1],
        [0, 1, 1, 0, 0, 1, 1],
        [0, 0, 0, 1, 1, 1, 1]
    ])

    # Recovery matrix (extract original 4 bits)
    _R = numpy.matrix([
        [0, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 1],
    ])
```

Encoding Process   `encode(self, message)` method takes 4-bit group as input and output 7-bit group

```python
# phy/Coding.py
def encode(self, message: list[int]) -> list[int]:
    """Encode 4-bit message to 7-bit codeword"""
    message = numpy.matrix(message).transpose()
    encoded = numpy.dot(self._gT, message) % 2
    return encoded.flatten().tolist()[0]
```

Error Detection and Correction   The syndrome $\mathbf{s} = H \cdot \mathbf{r} \mod 2$ indicates the error position: - $\mathbf{s} = [0, 0, 0]$: No error - $\mathbf{s} \neq [0, 0, 0]$: Error at position indicated by syndrome (binary to decimal)

```python
# phy/Coding.py
def getOriginalMessage(self, message: list[int]) -> list[int]:
    """Decode 7-bit codeword, correct single-bit error if present"""
    message = numpy.matrix(message).transpose()

    # Calculate syndrome
    syndrome = numpy.dot(self._h, message) % 2
    error_pos = self._binary_to_decimal(syndrome)

    # Correct error if detected
    if error_pos > 0:
        message[error_pos - 1, 0] ^= 1  # Flip the error bit

    # Extract original 4 bits
```

```
    original = numpy.dot(self._R, message)
    return original.flatten().tolist()[0]
```

Channel Encoder/Decoder Interface   The ChannelEncoder class provides byte-level interface:

```python
# phy/Coding.py
class ChannelEncoder:
    def __init__(self):
        self.ham = Hamming()
        self.n = 7  # Codeword length
        self.k = 4  # Message length

    def encoding(self, data: bytes) -> bytes:
        """Encode bytes with Hamming code"""
        bit_list = self.bytes_to_bits(data)

        # Pad to align to 4-bit boundaries
        if len(bit_list) % self.k:
            bit_list.extend([0] * (self.k - len(bit_list) % self.k))

        # Encode each 4-bit group
        encoded_bits = []
        for i in range(0, len(bit_list), self.k):
            group = bit_list[i:i+4]
            encoded_bits.extend(self.ham.encode(group))

        return self.bits_to_bytes(encoded_bits)

    def decoding(self, data: bytes) -> bytes:
        """Decode bytes with error correction"""
        bit_list = self.bytes_to_bits(data)

        # Decode each 7-bit group
        decoded_bits = []
        for i in range(0, len(bit_list), self.n):
            group = bit_list[i:i+7]
            decoded_bits.extend(self.ham.getOriginalMessage(group))

        return self.bits_to_bytes(decoded_bits)
```

Code Rate and Overhead

Parameter	Value
Code rate	$\frac{4}{7} \approx 0.571$
Overhead	75% (7 bits transmitted per 4 data bits)
Error correction	1 bit per 7-bit block
Error detection	2 bits per 7-bit block

3.1.3 Test Results

Test Modulator (and DeModulator)   test code:

```python
def test_modulating():
    """
    Test modulation waveform generation
    """
    modulator = Modulator(
        scheme="16QAM", symbol_rate=1e6, sample_rate=50e6, fc=2e6, power_factor=1, debug=True
    )
    test_str = b"Hello!"
    print(f"Original data: {test_str}")

    signal = modulator.modulate(data=test_str)

    demodulator= DeModulator(
        scheme="16QAM", symbol_rate=1e6, sample_rate=50e6, fc=2e6, power_factor=1, debug=True
    )
    recovered_data = demodulator.demodulate(signal=signal)
    print(f"Recovered data: {recovered_data}")

    assert recovered_data.startswith(test_str), "Data recovery failed!"
```
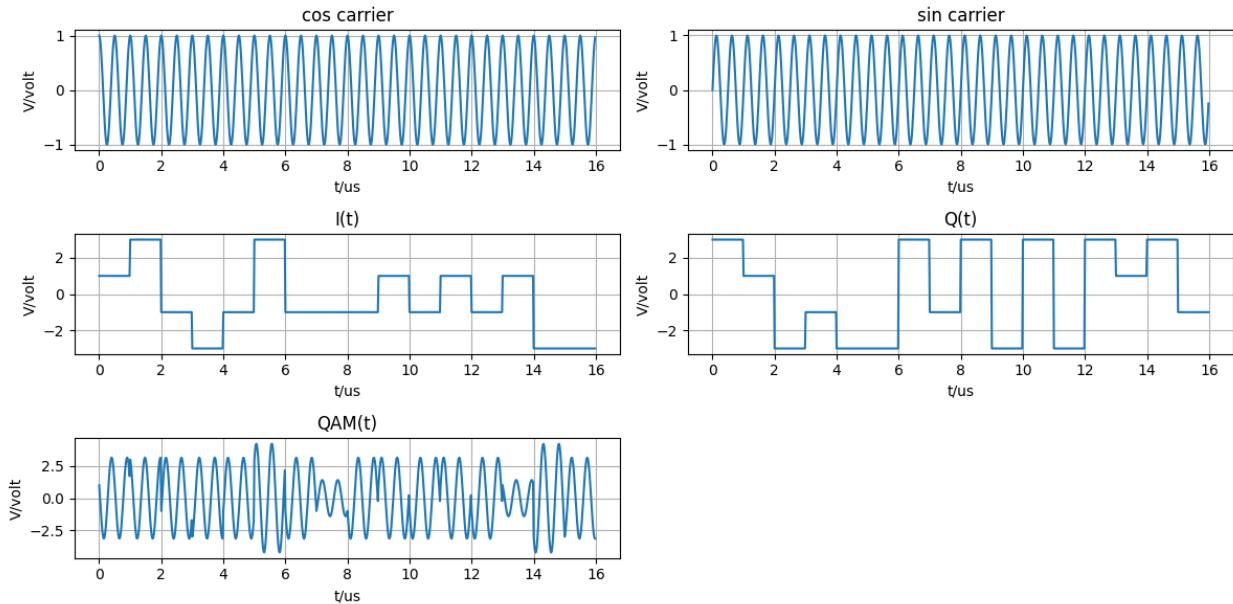

Figure 1: *modulated signal:*
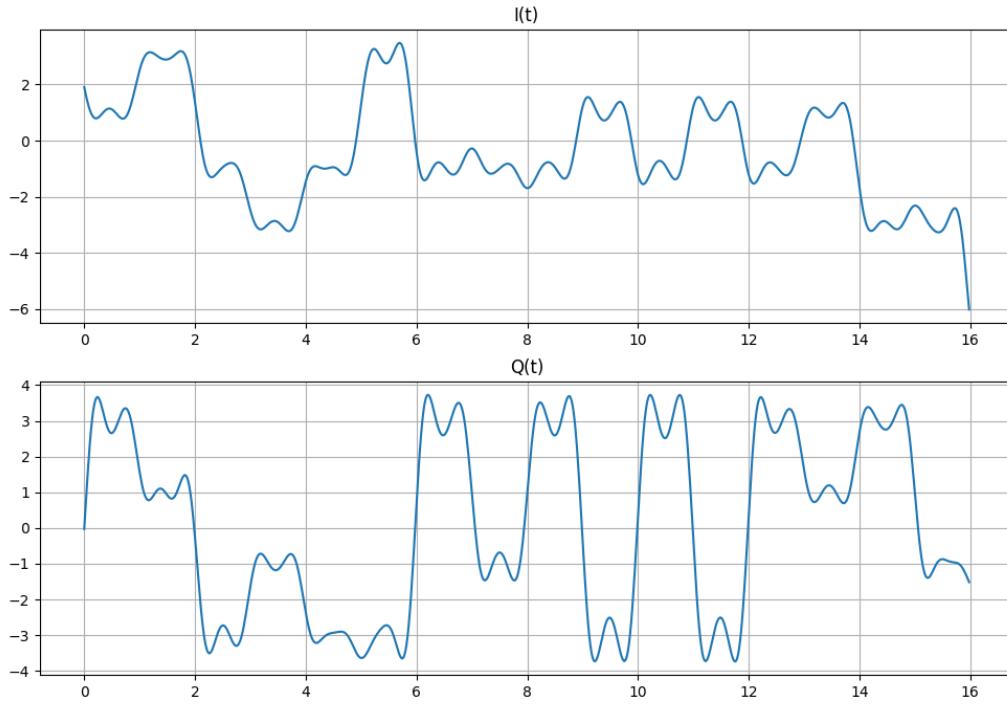
Figure 2: *demodulated IQ signal:*

Test Channel Coding   test code

```python
def test_channel_coding():
    channel_encoder = ChannelEncoder()
    message = b"Hello, Hamming Code!"
    print(f"Original Message: {message}")

    encoded_message = channel_encoder.encoding(data=message)
    print(f"Encoded Message: {encoded_message}")

    # Introduce a single-bit error for testing
    encoded_bits = channel_encoder.bytes_to_bits(byte_data=encoded_message)
    # Flip the 10th bit (for example)
    encoded_bits[10] ^= 1
    corrupted_encoded_message = channel_encoder.bits_to_bytes(bits=encoded_bits)
    print(f"Corrupted Encoded Message: {corrupted_encoded_message}")

    decoded_message = channel_encoder.decoding(data=corrupted_encoded_message)
    print(f"Decoded Message: {decoded_message}")

    assert message == decoded_message, "Decoded message does not match original!"
```

```
● (base) apollo•project/network-communication-project/phy(async ⚡)» uv run Coding.py          [17:30:47]
  Original Message: b'Hello, Hamming Code!'
  Encoded Message: b'\x99\xc32\\\xcf3<\xcd\xfdS\xc5@&p\xcd\xa75\\\xd5s\x19\xcc[0\xf5@&C\xcd\xff4\xcc\xc9Ui'
  Corrupted Encoded Message: b'\x99\xe32\\\xcf3<\xcd\xfdS\xc5@&p\xcd\xa75\\\xd5s\x19\xcc[0\xf5@&C\xcd\xff4\xcc\xc9
  Ui'
  Decoded Message: b'Hello, Hamming Code!'
```

Figure 3: *Hamming Coding*

3.2 Mac Layer

In the mac layer, we mainly:

- define the frame structure of our mac layer protocol with reference to the Ethernet frame structure, including all of its necessary fields;

- realize a robust switcher node with two-layer network, which has the ability of simultaneously forwarding as well as forwarding table learning.

3.2.1 Frame Structure (Level2)

Our MAC frame format is inspired by IEEE 802.3 Ethernet, with simplified fields suitable for simulation:

```
+----------+--------+--------+--------+--------+-------+
|  Header  | Length | Src MAC| Dst MAC| Payload |  CRC  |
+----------+--------+--------+--------+--------+-------+
| 2 bytes | 2 bytes| 1 byte | 1 byte | n bytes |4 bytes|
+----------+--------+--------+--------+--------+-------+
```

Field	Size	Description
Header	2B	Magic number `0xAABB` for frame synchronization
Length	2B	Total length of Src + Dst + Payload
Src MAC	1B	Source MAC address (0-255)
Dst MAC	1B	Destination MAC address (0-255)
Payload	nB	Upper layer data (TCP segment)
CRC32	4B	IEEE 802.3 CRC-32 checksum

Frame Encoding   The `NetworkInterface` class handles frame encapsulation:

```python
# mac/protocol.py
HEADER = b"\xaa\xbb"


class NetworkInterface:
    def __init__(self, mac_addr: int, mode: str = "node", name: str = "eth0"):
        self.mac_addr = mac_addr
        self.name = name
        self.header = HEADER
        self.mode = mode  # "node" or "switch"


    def encoding(self, dst_mac: int, data: bytes, src_mac: int) -> bytes:
```

14

```python
    """Encapsulate payload into MAC frame"""
    lens = 1 + 1 + 1 + len(data)
    header = struct.pack("!2sHBB", self.header, lens, src_mac, dst_mac)
    raw_frame = header + data
    crc = crc32_with_table(raw_frame)
    frame = raw_frame + struct.pack("!I", crc)
    return frame
```

One thing to note is that there are two modes of `NetworkInterface`, mode `'node'` would drop the frame whose destination mac is different from the host's mac while mode `'switch'` would keep all validate frame even if the destination mac is vary from the host's one because switcher need to forwarding frame to different host.

Frame Decoding with CRC Verification   The decoding process verifies frame integrity using CRC-32:

```python
# mac/protocol.py
def decoding(self, frame: bytes) -> tuple[int, int, bytes] | None:
    """Decode frame and verify CRC"""
    # Length check
    if len(frame) < 2 + 2 + 1 + 1 + 4:
        return None

    # Header check
    if frame[:2] != self.header:
        return None

    # CRC verification
    received_crc = struct.unpack("!I", frame[-4:])[0]
    calculated_crc = crc32_with_table(frame[:-4])
    if received_crc != calculated_crc:
        raise ValueError("CRC Mismatch")

    # Parse header fields
    _, lens, src_mac, dst_mac = struct.unpack("!2sHBB", frame[:6])

    # MAC address filtering (only for node mode)
    if dst_mac != self.mac_addr and self.mode == "node":
        raise ValueError("MAC Address Mismatch")

    # Extract payload
    data = frame[6:-4]
    return src_mac, dst_mac, data
```

CRC-32 Implementation (Level3 Part B)   We implement the IEEE 802.3 CRC-32 algorithm with lookup table optimization:

$$CRC(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

```python
# mac/crc.py
CRC32_POLYNOMIAL = 0xEDB88320  # Reflected polynomial


def crc32_with_table(data: bytes) -> int:
    """CRC-32 with pre-computed lookup table"""
    # Generate table on first call
    if not hasattr(crc32_with_table, '_table'):
        table = []
        for i in range(256):
            crc = i
            for _ in range(8):
                if crc & 1:
                    crc = (crc >> 1) ^ CRC32_POLYNOMIAL
                else:
                    crc >>= 1
            table.append(crc)
        crc32_with_table._table = table

    table = crc32_with_table._table
    crc = 0xFFFFFFFF

    for byte in data:
        crc = table[(crc ^ byte) & 0xFF] ^ (crc >> 8)

    return crc ^ 0xFFFFFFFF
```
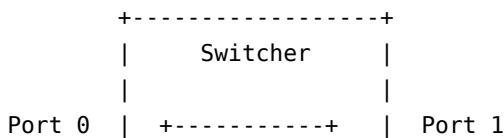
CRC Performance
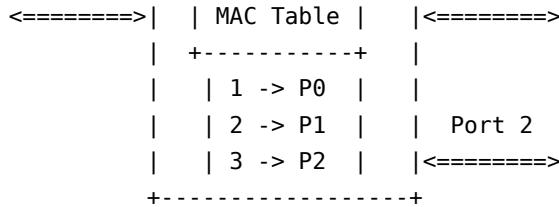
Implementation	1KB Data	Throughput
Basic (bit-by-bit)	0.45 ms	~2.2 MB/s
Table lookup	0.05 ms	~20 MB/s
Speedup	**~9x faster**	-

3.2.2 Switcher (Level2)

The Switcher class implements a Layer 2 switch with MAC address learning and frame forwarding capabilities.

Switch Architecture

```
        +------------------+
        |     Switcher     |
        |                  |
  Port 0 |  +-----------+  | Port 1
```

```
 <========>|  | MAC Table |   |<========>
           |  +-----------+   |
           |  | 1 -> P0 |    |
           |  | 2 -> P1 |    |  Port 2
           |  | 3 -> P2 |    |<========>
           +-----------------+
```

Each port has its own PHY and MAC layer stack:

```python
# mac/switcher.py
class Switcher(SimulationEntity):
    def __init__(self, simulator, mac_addr, port_num=3, name="switcher"):
        super().__init__(name=name)
        self.port_num = port_num
        self.socket_list = []  # MAC layer for each port
        self.port_list = []    # PHY layer for each port
        self.map = {}          # MAC address -> Port mapping

        # Create protocol stack for each port
        for i in range(port_num):
            phy_layer = PhyLayer(
                lower_layer=None,
                coding=True,
                simulator=simulator,
                name=f"{name}_port{i}"
            )
            mac_layer = MacLayer(
                lower_layer=phy_layer,
                simulator=simulator,
                mode='switch',  # Don't filter by dst MAC
                mac_addr=mac_addr,
                name=f"{name}_port{i}",
            )
            self.socket_list.append(mac_layer)
            self.port_list.append(phy_layer)

        simulator.register_entity(self)
```

MAC Address Learning   The switch passively learns MAC addresses by observing source addresses of incoming frames:

```
Time 1: Frame arrives at Port 0 (Src=A, Dst=B)
        -> Learn: MAC_A is at Port 0
        -> Flood to Port 1, 2 (unknown dst)

Time 2: Frame arrives at Port 1 (Src=B, Dst=A)
        -> Learn: MAC_B is at Port 1
        -> Forward to Port 0 (known dst)
```

```python
# mac/switcher.py
def update(self, tick: int):
    if tick % 10 == 0:  # Process every 10 ticks
        for i in range(self.port_num):
            socket_layer = self.socket_list[i]
            result = socket_layer.recv()

            if result:
                src_mac, dst_mac, data = result

                # Passive Learning
                if src_mac not in self.map:
                    self.map[src_mac] = i
                    print(f"Switcher learned MAC {src_mac} at port {i}")
                    if len(self.map) == self.port_num:
                        print(f"Switcher MAC table full: {self.map}")
                """
                or we can simply write as:
                self.map[src_mac] = i
                in case that the host that connected to the port may change
                """

                # Forwarding
                dst_port = self.map.get(dst_mac)

                if dst_port is not None:
                    # Unicast: known destination
                    self.socket_list[dst_port].send((src_mac, dst_mac, data))
                else:
                    # Flood: unknown destination
                    for j in range(self.port_num):
                        if j != i:
                            self.socket_list[j].send((src_mac, dst_mac, data))
```

Forwarding Logic

Forwarding Decision Table

Dst MAC	In MAC Table?	Action
Known	Yes	Unicast to mapped port
Unknown	No	Flood to all ports except source
Same as source port	Yes	Drop (same segment)

Switch Connection Interface   this interface need to specify the port that the switcher would like to use.

```python
# mac/switcher.py
def connect_to(self, port: int, twisted_pair: TwistedPair):
    """Connect a twisted pair cable to specified port"""
    phy_layer = self.port_list[port]
    twisted_pair.connect(
        tx_interface=phy_layer.tx_entity,
        rx_interface=phy_layer.rx_entity
    )
```

3.2.3 Test Results

Test CRC32   test code:

```python
def test_crc32_table_version():
    import zlib

    test_cases = [
        b"Hello, World!",
        b"123456789",
        bytes(range(256)),
        b"bulabulabula",
    ]

    print("=" * 60)
    print("CRC-32 Table Version Tests")
    print("=" * 60)

    all_passed = True
    for data in test_cases:
        basic_crc = crc32(data)
        table_crc = crc32_with_table(data)
        zlib_crc = zlib.crc32(data) & 0xFFFFFFFF

        match = (basic_crc == table_crc == zlib_crc)
        status = "pass" if match else "fail"
        if not match:
            all_passed = False

        print(f"{status}: basic={basic_crc:#010x} table={table_crc:#010x} zlib={zlib_crc:#010x}")

    assert all_passed, "CRC-32 table version tests fail"
```

```
(base) apollo•project/network-communication-project/mac(async ⚡)» uv run crc.py
 ============================================================
 CRC-32 Table Version Tests
 ============================================================
 pass: basic=0xec4ac3d0 table=0xec4ac3d0 zlib=0xec4ac3d0
 pass: basic=0xcbf43926 table=0xcbf43926 zlib=0xcbf43926
 pass: basic=0x29058c73 table=0x29058c73 zlib=0x29058c73
 pass: basic=0x414fa339 table=0x414fa339 zlib=0x414fa339

 All table version tests passed!
```

Figure 4: *crc32 checksum ok*

Test MultiNode Communication test code:

```python
def test_three_nodes_communicate_via_switcher():
    simulator = PhySimulationEngine(time_step_us=1)

    node1 = TestNode(simulator=simulator, mac_addr=1, name='node1')
    node2 = TestNode(simulator=simulator, mac_addr=2, name='node2')
    node3 = TestNode(simulator=simulator, mac_addr=3, name='node3')
    switcher = Switcher(simulator=simulator, mac_addr=0, port_num=3, name='switcher')

    cable = Cable(
        length=100,
        attenuation=4,
        noise_level=3,
        debug_mode=False,
    )
    print(f"\n{cable}")
    tp1 = TwistedPair(cable=cable, simulator=simulator, ID=0)
    tp2 = TwistedPair(cable=cable, simulator=simulator, ID=1)
    tp3 = TwistedPair(cable=cable, simulator=simulator, ID=2)

    node1.connect_to(tp1)
    switcher.connect_to(port=0, twisted_pair=tp1)
    node2.connect_to(tp2)
    switcher.connect_to(port=1, twisted_pair=tp2)
    node3.connect_to(tp3)
    switcher.connect_to(port=2, twisted_pair=tp3)

    test_msg1=  b"Hello from Node 1 to Node 2"
    test_msg2=  b"Hello from Node 3 to Node 2"

    print(f"\n[node1] Sending message to [node2]: {test_msg1}")
    node1.send(dst_mac=2, data=test_msg1)

    print(f"\n[node3] Sending message to [node2]: {test_msg2}")
```

```python
    node3.send(dst_mac=2, data=test_msg2)

    simulator.run(duration_ticks=3000)

    received_msgs_node2= node2.recvall()
    print(f"\n[node2] Received messages: {received_msgs_node2}")

    assert received_msgs_node2 is not None, "[node2] No messages received"
    assert len(received_msgs_node2) == 2, f"[node2] Expected 2 messages, got {len(received_msgs_node2)}"
    assert test_msg1 in received_msgs_node2, "[node2] Message from Node 1 missing"
    assert test_msg2 in received_msgs_node2, "[node2] Message from Node 3 missing"
```

```
● (base) apollo•project/network-communication-project/test(async ⚡)» uv run test_mac.py          [16:35:04]

Cable(length=100m, attenuation=4dB/m, noise_level=3, debug=False)
[node1] connected to TwistedPair_0(Channel A, Channel B)
[node2] connected to TwistedPair_1(Channel A, Channel B)
[node3] connected to TwistedPair_2(Channel A, Channel B)

[node1] Sending message to [node2]: b'Hello from Node 1 to Node 2'

[node3] Sending message to [node2]: b'Hello from Node 3 to Node 2'


==========================================================
Starting simulation for 3000 ticks
==========================================================

[TICK 0][switcher][MAC] Switcher learned MAC 1 is at port 0
[TICK 0][switcher][MAC] Switcher learned MAC 3 is at port 2
[node1] MAC Address Mismatch: dst 2, self 1
[TICK 12][node1][MAC] Frame decoding error: MAC Address Mismatch
[node3] MAC Address Mismatch: dst 2, self 3
[TICK 12][node3][MAC] Frame decoding error: MAC Address Mismatch


==========================================================
Simulation completed!
==========================================================

Simulation reset

[node2] Received messages: [b'Hello from Node 1 to Node 2', b'Hello from Node 3 to Node 2']
```

Figure 5: *MultiNode simultaneously communication*

Test Forwarding Table Learning   test code:

```python
def test_switcher_mac_learning():
    simulator = PhySimulationEngine(time_step_us=1)

    node1 = TestNode(simulator=simulator, mac_addr=1, name='node1')
    node2 = TestNode(simulator=simulator, mac_addr=2, name='node2')
    switcher = Switcher(simulator=simulator, mac_addr=0, port_num=2, name='switcher')

    cable = Cable(
        length=100,
        attenuation=4,
        noise_level=3,
```

```python
    debug_mode=False,
)
print(f"\n{cable}")
tp1 = TwistedPair(cable=cable, simulator=simulator, ID=0)
tp2 = TwistedPair(cable=cable, simulator=simulator, ID=1)

node1.connect_to(tp1)
switcher.connect_to(port=0, twisted_pair=tp1)
node2.connect_to(tp2)
switcher.connect_to(port=1, twisted_pair=tp2)

print(f"\nSwitcher MAC Table: {switcher.map}")

test_msg1=  b"Message 1 from Node 1 to Node 2"
test_msg2=  b"Message 2 from Node 2 to Node 1"

print(f"\n[node1] Sending message to [node2]: {test_msg1}")
node1.send(dst_mac=2, data=test_msg1)

simulator.run(duration_ticks=2000)

received_msgs_node2= node2.recvall()
print(f"\n[node2] Received messages: {received_msgs_node2}")

print(f"\n[node2] Sending message to [node1]: {test_msg2}")
node2.send(dst_mac=1, data=test_msg2)

simulator.run(duration_ticks=2000)

received_msgs_node1= node1.recvall()
print(f"\n[node1] Received messages: {received_msgs_node1}")


print(f"\nSwitcher MAC Table: {switcher.map}")
assert switcher.map.get(1) == 0, "Switcher did not learn MAC 1 correctly"
assert switcher.map.get(2) == 1, "Switcher did not learn MAC 2 correctly"

print("\n=== Switcher MAC Learning Test Passed ===")
```

```
Switcher MAC Table: {}

[node1] Sending message to [node2]: b'Message 1 from Node 1 to Node 2'


===========================================================
Starting simulation for 2000 ticks
===========================================================

[TICK 10][switcher][MAC] Switcher learned MAC 1 is at port 0


===========================================================
Simulation completed!
===========================================================

Simulation reset

[node2] Received messages: [b'Message 1 from Node 1 to Node 2']

[node2] Sending message to [node1]: b'Message 2 from Node 2 to Node 1'


===========================================================
Starting simulation for 2000 ticks
===========================================================

[TICK 10][switcher][MAC] Switcher learned MAC 2 is at port 1
[TICK 10][switcher][MAC] Switcher MAC table full: {1: 0, 2: 1}


===========================================================
Simulation completed!
===========================================================

Simulation reset

[node1] Received messages: [b'Message 2 from Node 2 to Node 1']

Switcher MAC Table: {1: 0, 2: 1}

=== Switcher MAC Learning Test Passed ===
```

Figure 6: *switcher learnt the fowarding table*

3.3 Transport Layer

In the transport layer, we mainly:

- Implement classic pipeline reliable transport protocols, including GBN and SR, and realize the flow control feature for the latter one;

- provide the utility of port multiplexing and demultiplexing;

- encapsulate a series of easy-to-use socket interface with POSIX style, including functions like: `send`, `recv`, `connect`, `listen`, `accept` and `close`.

3.3.1 Segment Structure

Our transport layer segment format:

```
+------+----------+----------+-----+-----+----------+
| Type | Src Port | Dst Port | Seq | Ack |  Payload |
+------+----------+----------+-----+-----+----------+
```

```
| 1B |    2B    |     2B    | 2B | 2B |     nB    |
+------+----------+----------+-----+-----+----------+
```

Field	Size	Description
Type	1B	0=DATA, 1=ACK
Src Port	2B	Source port number
Dst Port	2B	Destination port number
Seq	2B	Sequence number
Ack	2B	Acknowledgment number
Payload	nB	Upper layer data (up to MSS=1024 bytes)

For SR ACK packets, an additional field is included:

```
+------+----------+----------+-----+-----+------+---------+
| Type | Src Port | Dst Port | Seq | Ack | RWND | Payload |
+------+----------+----------+-----+-----+------+---------+
| 1B  |    2B    |     2B    | 2B | 2B | 2B  |    nB    |
+------+----------+----------+-----+-----+------+---------+
```

The `RWND` (Receive Window) field enables flow control in SR protocol.

```python
# tcp/TransportLayer.py
HEADER_FMT = "!BHHHH"        # Type(1B) | Src_Port(2B) | Dst_Port(2B) | Seq(2B) | Ack(2B)
HEAD_FMT_ACK = "!BHHHHH"    # Additional RWND(2B) for SR ACK
HEADER_SIZE = struct.calcsize(HEADER_FMT)
ACK_HEADER_SIZE = struct.calcsize(HEAD_FMT_ACK)


TYPE_DATA = 0
TYPE_ACK = 1


TIMEOUT_TICKS = 128   # Retransmission timeout
WINDOW_SIZE = 12      # Sliding window size
MSS = 1024            # Maximum Segment Size
```

3.3.2 Session Management

Each connection is identified by a 4-tuple: (local_mac, local_port, remote_mac, remote_port). The `TransportSession` class maintains per-connection state:

```python
# tcp/TransportLayer.py
class TransportSession:
    """Maintains transport state for a specific connection"""

    def __init__(self, local_mac, local_port, remote_mac, remote_port):
        self.local_mac = local_mac
        self.local_port = local_port
        self.remote_mac = remote_mac
        self.remote_port = remote_port
```

```python
        # Sequence numbers
        self.seq_out = 0    # Next sequence number to send
        self.seq_in = 0     # Expected sequence number to receive

        # Sender state
        self.unacked_packets: OrderedDict[int, Tuple[int, bytes]] = {}
        self.send_buffer = b""
        self.tx_window_size = WINDOW_SIZE

        # Receiver state (for SR)
        self.receive_buffer: list[tuple[int, bytes]] = []
        self.rx_window_size = WINDOW_SIZE
```

Sessions are created on-demand and cached:

```python
# tcp/TransportLayer.py
def _get_session(self, local_mac, local_port, remote_mac, remote_port):
    key = (local_mac, local_port, remote_mac, remote_port)
    if key not in self.sessions:
        self.sessions[key] = TransportSession(
            local_mac, local_port, remote_mac, remote_port
        )
    return self.sessions[key]
```

3.3.3 Reliable Transmission - Go-Back-N (Level3 Part A)

Go-Back-N (GBN) is a sliding window protocol with cumulative acknowledgments.

GBN Principle

```
Sender Window (size=4):
+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---+---+---+---+---+---+---+---+
  ^           ^
  |           |
 base       nextseq

- Packets 0-3: Sent, waiting for ACK
- Packets 4-7: Not yet sent
- On ACK(2): Slide window, base=2
- On timeout: Retransmit ALL packets from base
```

Sender Implementation   The sender maintains a buffer and flushes data within the window:

```python
# tcp/TransportLayer.py
class TransportLayer_GBN(ProtocolLayer):
    def send(self, data: tuple) -> TransportSession:
        """Application sends data"""
```

```python
        src_port, dst_mac, dst_port, payload = data
        src_mac = self.lower_layer.mac_addr
        session = self._get_session(src_mac, src_port, dst_mac, dst_port)

        # Buffer the data
        session.send_buffer += payload

        # Try to send within window
        self._flush_send_buffer(session)
        return session

    def _flush_send_buffer(self, session):
        """Send data while window allows"""
        while (len(session.send_buffer) > 0 and
                len(session.unacked_packets) < session.tx_window_size):
            # Segment data
            chunk = session.send_buffer[:MSS]
            session.send_buffer = session.send_buffer[MSS:]

            # Encapsulate and send
            encapsulated = self.Encapsulate(session, chunk)
            if encapsulated and self.lower_layer:
                self.lower_layer.send(encapsulated)

    def Encapsulate(self, session, data):
        """Add transport header"""
        header = struct.pack(
            HEADER_FMT,
            TYPE_DATA,
            session.local_port,
            session.remote_port,
            session.seq_out,
            session.seq_in,
        )
        packet = header + data

        # Cache for retransmission
        session.unacked_packets[session.seq_out] = (
            self.simulator.current_tick, packet
        )
        session.seq_out += 1

        return (session.remote_mac, packet)
```

Receiver Implementation   The receiver uses cumulative ACK - acknowledges all packets up to the received sequence:

```python
# tcp/TransportLayer.py
def Dencapsulate(self, data):
    """Process received segment"""
    src_mac, dst_mac, frame = data
    msg_type, src_port, dst_port, seq, ack = struct.unpack(
        HEADER_FMT, frame[:HEADER_SIZE]
    )
    payload = frame[HEADER_SIZE:]

    session = self._get_session(dst_mac, dst_port, src_mac, src_port)

    # Process cumulative ACK
    acked_seqs = [s for s in session.unacked_packets if s < ack]
    for s in acked_seqs:
        del session.unacked_packets[s]
    self._flush_send_buffer(session)

    if msg_type == TYPE_ACK:
        return None  # Pure ACK, don't pass up

    elif msg_type == TYPE_DATA:
        if seq == session.seq_in:
            # In-order packet
            session.seq_in += 1
            self._send_ack(session)
            # Store data for application
            if session not in self.session2data:
                self.session2data[session] = payload
            else:
                self.session2data[session] += payload
            return None

        elif seq < session.seq_in:
            # Duplicate packet, resend ACK
            self._send_ack(session)
            return None

        else:
            # Out-of-order packet, drop and resend ACK
            self._send_ack(session)
            return None
```

Timeout Retransmission   GBN retransmits ALL unacked packets on timeout:

```python
# tcp/TransportLayer.py
def update(self, tick):
    """Check for timeouts each tick"""
    super().update(tick)
```

```python
    for _, session in self.sessions.items():
        if not session.unacked_packets:
            continue

        # Check oldest packet
        _, (base_tick, _) = next(iter(session.unacked_packets.items()))

        if tick - base_tick > TIMEOUT_TICKS:
            # Retransmit ALL unacked packets
            for seq, (sent_tick, packet) in session.unacked_packets.items():
                if self.lower_layer:
                    self.lower_layer.send((session.remote_mac, packet))
                session.unacked_packets[seq] = (tick, packet)
            break
```

3.3.4 Reliable Transmission - Selective Repeat (Level3 Part A)

Selective Repeat (SR) improves upon GBN by only retransmitting lost packets.

SR vs GBN Comparison

Feature	Go-Back-N	Selective Repeat
ACK Type	Cumulative	Individual
Retransmission	All from base	Only lost packet
Receiver Buffer	Not needed	Required
Efficiency	Lower (more retrans)	Higher
Complexity	Simple	More complex

SR Receiver with Buffering   SR receiver buffers out-of-order packets:

```python
# tcp/TransportLayer.py
class TransportLayer_SR(ProtocolLayer):
    def Dencapsulate(self, data):
        src_mac, dst_mac, frame = data
        msg_type = struct.unpack("!B", frame[:1])[0]

        if msg_type == TYPE_ACK:
            msg_type, seq, ack, rwnd = struct.unpack(
                HEAD_FMT_ACK, frame[:ACK_HEADER_SIZE]
            )
            # Update sender window based on receiver's RWND
            session.tx_window_size = max(min(WINDOW_SIZE, rwnd), 1)
            if session.unacked_packets.get(ack):
                del session.unacked_packets[ack]
            self._flush_send_buffer(session)
            return None
```

```python
        elif msg_type == TYPE_DATA:
            # Always ACK received packet
            self._send_ack(src_mac, seq)

            if seq == session.seq_in:
                # In-order: deliver and check buffer
                combined_data = payload
                session.seq_in += 1

                # Deliver buffered packets in order
                while (session.receive_buffer and
                        session.receive_buffer[0][0] == session.seq_in):
                    _, buffered_payload = session.receive_buffer.pop(0)
                    session.seq_in += 1
                    combined_data += buffered_payload

                return combined_data

            elif seq > session.seq_in:
                # Out-of-order: buffer if within window
                if seq < session.rx_window_size + session.seq_in:
                    # Check for duplicates
                    if not any(s == seq for s, _ in session.receive_buffer):
                        session.receive_buffer.append((seq, payload))
                        session.receive_buffer.sort(key=lambda x: x[0])
                return None
```

Flow Control with RWND SR implements flow control by advertising receive window size:
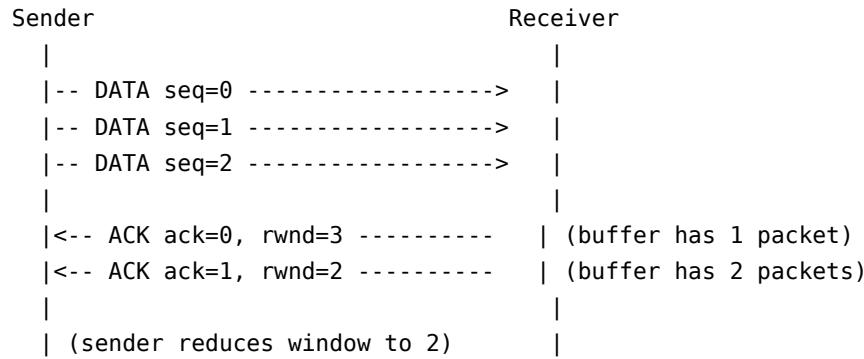
```python
# tcp/TransportLayer.py
def _send_ack(self, dst_mac, ack_num):
    """Send ACK with receive window advertisement"""
    session = self._get_session(dst_mac)

    # Calculate available buffer space
    rwnd = WINDOW_SIZE - len(session.receive_buffer)

    header = struct.pack(
        HEAD_FMT_ACK,
        TYPE_ACK,
        0,
        ack_num,
        rwnd,  # Advertise receive window
    )
    if self.lower_layer:
        self.lower_layer.send((dst_mac, header))
```

The sender adjusts its window based on RWND:

```
Sender                              Receiver
  |                                   |
  |-- DATA seq=0 ----------------->   |
  |-- DATA seq=1 ----------------->   |
  |-- DATA seq=2 ----------------->   |
  |                                   |
  |<-- ACK ack=0, rwnd=3 ----------   | (buffer has 1 packet)
  |<-- ACK ack=1, rwnd=2 ----------   | (buffer has 2 packets)
  |                                   |
  | (sender reduces window to 2)      |
```

3.3.5 Socket Interface

We provide a POSIX-style socket API for application layer use.

```python
# tcp/socket.py
class socket:
    def __init__(self, tcp_layer: TransportLayer_GBN):
        self.local_port = 8080
        self.tcp_layer = tcp_layer
        self.session: TransportSession = None
        self.is_listening = False
        self.established_sessions = []
```

Socket Class Overview

```python
# tcp/socket.py
def bind(self, port: int):
    """Bind socket to local port"""
    self.local_port = port
```

bind() - Assign Local Port

```python
# tcp/socket.py
def listen(self, num: int = 5):
    """Start listening for connections"""
    self.is_listening = True
    self.listen_num = num

def accept(self) -> Optional[socket]:
    """Accept incoming connection (non-blocking)"""
    if not self.is_listening:
        raise LookupError("accept before listen")

    # Find new sessions on this port
```

```python
    my_sessions = [
        s for s in self.tcp_layer.session2data.keys()
        if s.local_port == self.local_port
    ]

    for session in my_sessions:
        if session not in self.established_sessions:
            self.established_sessions.append(session)

            # Remove greeting data
            buffer = self.tcp_layer.session2data.get(session, b"")
            if buffer.startswith(self.greeting_data):
                buffer = buffer[len(self.greeting_data):]
                self.tcp_layer.session2data[session] = buffer

            # Create new socket for this connection
            new_socket = socket(tcp_layer=self.tcp_layer)
            new_socket.bind(self.local_port)
            new_socket.session = session
            return new_socket

    return None  # No pending connections
```

listen() and accept() - Server Side

```python
# tcp/socket.py
def connect(self, dst_mac: int, dst_port: int):
    """Connect to remote host (initiates session)"""
    self.session = self.tcp_layer.send(
        (self.local_port, dst_mac, dst_port, self.greeting_data)
    )
```

connect() - Client Side

```python
# tcp/socket.py
def send(self, data: bytes):
    """Send data over established connection"""
    if not self.session:
        raise LookupError("send before connection")

    dst_mac = self.session.remote_mac
    dst_port = self.session.remote_port
    self.tcp_layer.send((self.local_port, dst_mac, dst_port, data))

def recv(self, length: int = 128) -> Optional[bytes]:
    """Receive data (non-blocking)"""
```

```python
    if self.is_listening:
        raise LookupError("listening socket cannot recv directly")
    if not self.session:
        raise LookupError("recv before connection")

    if self.session not in self.tcp_layer.session2data:
        return None

    buffer = self.tcp_layer.session2data[self.session]
    if not buffer:
        return None

    # Return up to 'length' bytes
    data = buffer[:length]
    remaining = buffer[length:]

    if remaining:
        self.tcp_layer.session2data[self.session] = remaining
    else:
        del self.tcp_layer.session2data[self.session]

    return data
```

send() and recv() - Data Transfer

```python
# Server side
server_sock = socket(tcp_layer=server_tcp)
server_sock.bind(80)
server_sock.listen(5)

# In simulation loop
client_sock = server_sock.accept()
if client_sock:
    data = client_sock.recv(1024)
    if data:
        client_sock.send(b"HTTP/1.1 200 OK\r\n\r\nHello")

# Client side
client_sock = socket(tcp_layer=client_tcp)
client_sock.bind(12345)
client_sock.connect(dst_mac=1, dst_port=80)
client_sock.send(b"GET / HTTP/1.1\r\n\r\n")

# In simulation loop
response = client_sock.recv(1024)
```
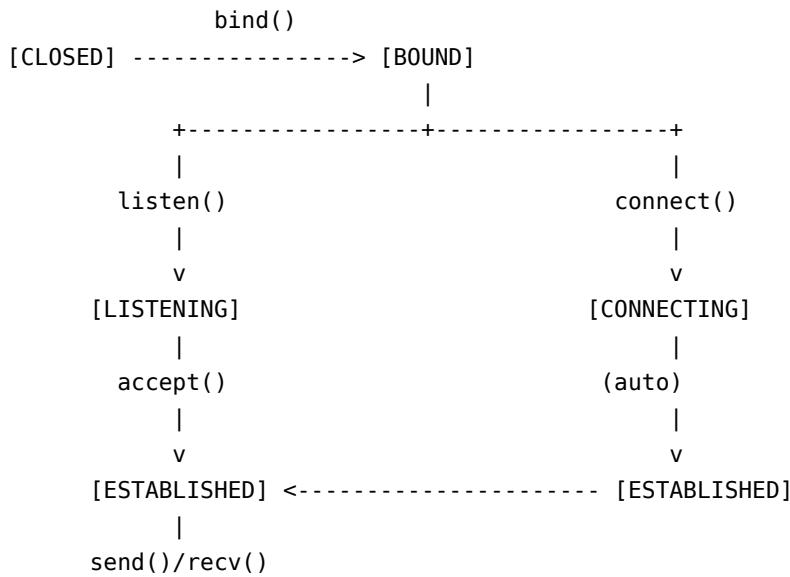
Usage Example

Socket State Diagram

```
                    bind()
    [CLOSED] ----------------> [BOUND]
                                   |
            +----------------+----------------+
            |                                 |
         listen()                          connect()
            |                                 |
            v                                 v
       [LISTENING]                       [CONNECTING]
            |                                 |
         accept()                           (auto)
            |                                 |
            v                                 v
       [ESTABLISHED] <--------------------- [ESTABLISHED]
            |
       send()/recv()
```

3.3.6 Test Results

TODO

Test Socket   test code:

```python
def test_two_client_sockets_to_one_server():
    simulator = PhySimulationEngine(time_step_us=1)

    # 三个节点，其中 node2 做"服务器"
    node1 = TestNode(simulator=simulator, mac_addr=1, name='node1')
    node2 = TestNode(simulator=simulator, mac_addr=2, name='node2')
    node3 = TestNode(simulator=simulator, mac_addr=3, name='node3')
    switcher = Switcher(simulator=simulator, mac_addr=0, port_num=3, name='switcher')

    # 信道与连接
    cable = Cable(
        length=100,
        attenuation=3,
        noise_level=4,
        debug_mode=False,
    )
    print(f"\n{cable}")
    tp1 = TwistedPair(cable=cable, simulator=simulator, ID=0)
    tp2 = TwistedPair(cable=cable, simulator=simulator, ID=1)
    tp3 = TwistedPair(cable=cable, simulator=simulator, ID=2)

    node1.connect_to(tp1)
    switcher.connect_to(port=0, twisted_pair=tp1)
    node2.connect_to(tp2)
```

```python
switcher.connect_to(port=1, twisted_pair=tp2)
node3.connect_to(tp3)
switcher.connect_to(port=2, twisted_pair=tp3)

# --- 准备 socket ---
# node2: 作为服务器，在 8080 端口监听
server_listen = node2.socket
server_listen.listen(num=5)

# node1: 两个客户端 socket，端口不同，分别连到 node2:8080
client_sock1 = socket(tcp_layer=node1.tcp_layer)
client_sock1.bind(10000)
client_sock1.setmode('debug')
client_sock2 = socket(tcp_layer=node1.tcp_layer)
client_sock2.bind(10001)
client_sock2.setmode('debug')
# --- 两个客户端依次发起连接并发送不同的数据 ---
msg1 = b'hello from client 1'
msg2 = b'hello from client 2'

client_sock1.connect(dst_mac=node2.mac_addr, dst_port=8080)
client_sock1.send(msg1)

client_sock2.connect(dst_mac=node2.mac_addr, dst_port=8080)
client_sock2.send(msg2)

# 运行一段时间，让报文都到达 node2
simulator.run(duration_ticks=5000)

# --- 服务器侧：accept 两次，拿到两个不同的连接 socket ---
conn1 = server_listen.accept()
conn2 = server_listen.accept()

print(f'conn1 session: {conn1.session}')
print(f'conn2 session: {conn2.session}')

assert conn1 is not None, "first accept() should return a connection socket"
assert conn2 is not None, "second accept() should return a connection socket"
assert conn1 is not conn2, "two accepted sockets should be different objects"
assert conn1.session is not conn2.session, "two connections should have different sessions"

# --- 从两个连接 socket 分别 recv，检查数据是否对应 ---
data1 = conn1.recv()
data2 = conn2.recv()

print("server conn1 recv:", data1)
print("server conn2 recv:", data2)
```

```python
# 注意：顺序取决于谁先到达，这里只断言"集合相等"，不要求顺序
recv_set = {data1, data2}
expect_set = {msg1, msg2}
assert recv_set == expect_set, f"server should receive {expect_set}, but got {recv_set}"

# --- 可选：再从服务器回一条消息，看客户端是否各自能收到 ---
resp1 = b'response to client 1'
resp2 = b'response to client 2'
conn1.send(resp1)
conn2.send(resp2)

simulator.run(duration_ticks=5000)

c1_resp = client_sock1.recv()
c2_resp = client_sock2.recv()
print("client1 recv:", c1_resp)
print("client2 recv:", c2_resp)

client_sock1.send(b'ack from client 1')
client_sock2.send(b'ack from client 2')
simulator.run(duration_ticks=5000)

print("server conn1 recv after ack:", conn1.recv())
print("server conn2 recv after ack:", conn2.recv())

assert c1_resp == resp1
assert c2_resp == resp2
```

```
[TICK 0][node2] ACCEPT sock_port=8080 new_session=[session:(2:8080) -> (1:10000)] peer=1:10000 pending_data_len=
19
[TICK 0][node2] ACCEPT sock_port=8080 new_session=[session:(2:8080) -> (1:10001)] peer=1:10001 pending_data_len=
19
conn1 session: [session:(2:8080) -> (1:10000)]
conn2 session: [session:(2:8080) -> (1:10001)]
[TICK 0][node2] RECV sock_port=8080 peer=1:10000 len=19 preview=b'hello from client 1'
[TICK 0][node2] RECV sock_port=8080 peer=1:10001 len=19 preview=b'hello from client 2'
server conn1 recv: b'hello from client 1'
server conn2 recv: b'hello from client 2'
[TICK 0][node2] SEND sock_port=8080 peer=1:10000 len=20 preview=b'response to clie'
[TICK 0][node2] SEND sock_port=8080 peer=1:10001 len=20 preview=b'response to clie'
```

Figure 7: *step1: server socket received message*

```
[TICK 0][node1] RECV sock_port=10000 peer=2:8080 len=20 preview=b'response to client 1'
[TICK 0][node1] RECV sock_port=10001 peer=2:8080 len=20 preview=b'response to client 2'
client1 recv: b'response to client 1'
client2 recv: b'response to client 2'
[TICK 0][node1] SEND sock_port=10000 peer=2:8080 len=17 preview=b'ack from client '
[TICK 0][node1] SEND sock_port=10001 peer=2:8080 len=17 preview=b'ack from client '
```

Figure 8: *step2: client socket received message*

3.4 Application Layer
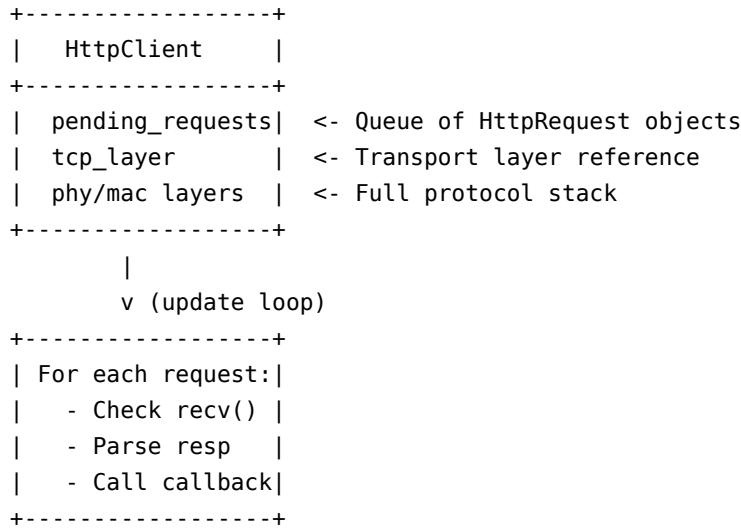
In this segment, we realize:

- an asynchronous HTTP client, which has the capability of sending multiple HTTP requests and triggering the callback function automatically once the response is received

- a non-blocking HTTP server, which uses route table to distinguish different requests and construct corresponding HTTP response

3.4.1 HTTP Client (Level3 Part C)

The HTTP client is designed for non-blocking operation, allowing multiple concurrent requests.

Client Architecture

```
+------------------+
|   HttpClient     |
+------------------+
|  pending_requests|  <- Queue of HttpRequest objects
|  tcp_layer       |  <- Transport layer reference
|  phy/mac layers  |  <- Full protocol stack
+------------------+
        |
        v (update loop)
+------------------+
| For each request:|
|   - Check recv() |
|   - Parse resp   |
|   - Call callback|
+------------------+
```

```python
# app/client.py
class RequestState(Enum):
    IDLE = 0        # Not yet sent
    PENDING = 1     # Sent, waiting for response
    COMPLETED = 2   # Response received

class HttpRequest:
```

```python
    """HTTP request wrapper"""
    def __init__(self, method: str, path: str, headers: dict = None, body: bytes = b''):
        self.method = method
        self.path = path
        self.headers = headers or {}
        self.body = body
        self.state = RequestState.IDLE
        self.response: Optional[dict] = None
        self.callback: Optional[Callable] = None
        self.sock: Optional[socket] = None
```

Request State Machine

Client Initialization   Each client has its own complete protocol stack:

```python
# app/client.py
class HttpClient(SimulationEntity):
    def __init__(self, simulator: PhySimulationEngine, mac_addr: int, name: str = 'http_client'):
        super().__init__(name=name)
        self.simulator = simulator
        self.mac_addr = mac_addr

        # Protocol stack
        self.phy_layer = PhyLayer(lower_layer=None, coding=True, simulator=simulator, name=name)
        self.mac_layer = MacLayer(
            lower_layer=self.phy_layer,
            simulator=simulator, mode='node',
            mac_addr=mac_addr, name=name
        )
        self.tcp_layer = TransportLayer_GBN(
            lower_layer=self.mac_layer, simulator=simulator, name=name
        )

        # Port allocation for multiple connections
        self._next_port = 10000

        # Pending request queue
        self.pending_requests: list[HttpRequest] = []

        simulator.register_entity(self)
```

```python
# app/client.py
def _build_http_request(self, method: str, path: str, host: str,
                        headers: dict = None, body: bytes = b'') -> bytes:
    """Build HTTP/1.1 request bytes"""
    request = f"{method} {path} HTTP/1.1\r\n"
    request += f"Host: {host}\r\n"
```

```python
    if headers:
        for key, value in headers.items():
            request += f"{key}: {value}\r\n"

    if body:
        request += f"Content-Length: {len(body)}\r\n"

    request += "Connection: close\r\n"
    request += "\r\n"

    return request.encode('utf-8') + body
```

Building HTTP Requests

Async GET/POST Methods   The client provides async-style request methods with callback
support:

```python
# app/client.py
def get(self, dst_mac: int, dst_port: int, path: str,
        callback: Callable = None) -> HttpRequest:
    """Send GET request (async)"""
    return self._send_request('GET', dst_mac, dst_port, path, callback=callback)


def post(self, dst_mac: int, dst_port: int, path: str,
         body: bytes = b'', callback: Callable = None) -> HttpRequest:
    """Send POST request (async)"""
    return self._send_request('POST', dst_mac, dst_port, path, body=body, callback=callback)


def _send_request(self, method: str, dst_mac: int, dst_port: int,
                  path: str, body: bytes = b'', callback: Callable = None) -> HttpRequest:
    """Internal method to send HTTP request"""
    req = HttpRequest(method, path, body=body)
    req.callback = callback
    req.state = RequestState.PENDING

    # Create socket and connect
    sock = socket(tcp_layer=self.tcp_layer)
    sock.bind(self._allocate_port())
    sock.connect(dst_mac=dst_mac, dst_port=dst_port)
    req.sock = sock

    # Build and send HTTP request
    http_data = self._build_http_request(method, path, f"{dst_mac}:{dst_port}", body=body)
    sock.send(http_data)

    self.pending_requests.append(req)
    return req
```

```python
# app/client.py
def _parse_http_response(self, data: bytes) -> Optional[dict]:
    """Parse HTTP response"""
    try:
        text = data.decode('utf-8', errors='ignore')
        lines = text.split('\r\n')

        # Parse status line: HTTP/1.1 200 OK
        status_line = lines[0].split(' ', 2)
        status_code = int(status_line[1])
        status_msg = status_line[2] if len(status_line) > 2 else ''

        # Parse headers
        headers = {}
        body_start = 0
        for i, line in enumerate(lines[1:], 1):
            if line == '':
                body_start = i + 1
                break
            if ':' in line:
                key, value = line.split(':', 1)
                headers[key.strip()] = value.strip()

        body = '\r\n'.join(lines[body_start:])

        return {
            'status_code': status_code,
            'status_msg': status_msg,
            'headers': headers,
            'body': body,
            'raw': data
        }
    except Exception as e:
        return None
```

Response Parsing

Non-blocking Update Loop   The client polls for responses in its update loop:

```python
# app/client.py
def update(self, tick):
    """Non-blocking update loop - poll for responses"""
    super().update(tick)

    for req in self.pending_requests[:]:  # Copy list to allow modification
        if req.state != RequestState.PENDING:
            continue
```

```python
        # Try to receive response (non-blocking)
        data = req.sock.recv(length=4096)
        if data:
            response = self._parse_http_response(data)
            req.response = response
            req.state = RequestState.COMPLETED

            # Trigger callback if provided
            if req.callback:
                req.callback(response)

        self.pending_requests.remove(req)
```
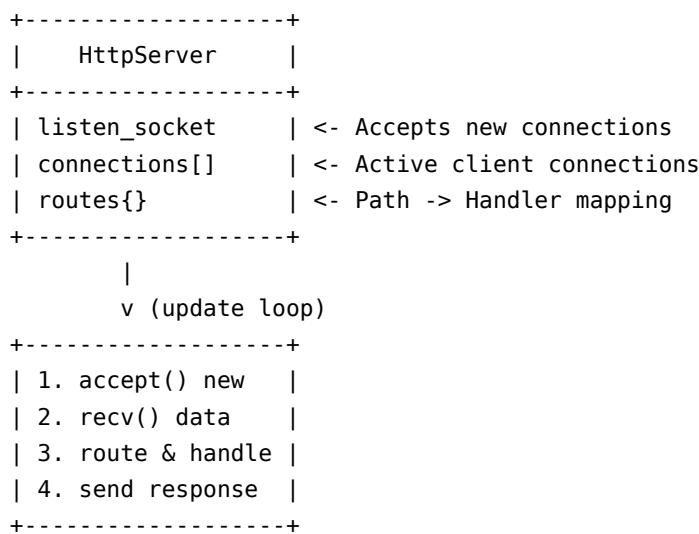
3.4.2 HTTP Server (Level3 Part C)

The HTTP server handles incoming connections and routes requests to appropriate handlers.

Server Architecture

```
+-------------------+
|    HttpServer     |
+-------------------+
| listen_socket     | <- Accepts new connections
| connections[]     | <- Active client connections
| routes{}          | <- Path -> Handler mapping
+-------------------+
        |
        v (update loop)
+-------------------+
| 1. accept() new   |
| 2. recv() data    |
| 3. route & handle |
| 4. send response  |
+-------------------+
```

```python
# app/server.py
class HttpServer(SimulationEntity):
    def __init__(self, simulator: PhySimulationEngine, mac_addr: int,
                 name: str = 'http_server', port: int = 80):
        super().__init__(name=name)
        self.simulator = simulator
        self.mac_addr = mac_addr

        # Protocol stack
        self.phy_layer = PhyLayer(lower_layer=None, coding=True, simulator=simulator, name=name)
        self.mac_layer = MacLayer(
```

```
            lower_layer=self.phy_layer,
            simulator=simulator, mode='node',
            mac_addr=mac_addr, name=name
        )
        self.tcp_layer = TransportLayer_GBN(
            lower_layer=self.mac_layer, simulator=simulator, name=name
        )

        # Listening socket
        self.listen_socket = socket(tcp_layer=self.tcp_layer)
        self.listen_socket.bind(port)
        self.listen_socket.listen(num=10)

        # Active connections
        self.connections: list[socket] = []

        # Route table: path -> handler(request) -> response_body
        self.routes: Dict[str, callable] = {}
        self._setup_default_routes()

        simulator.register_entity(self)
```

Server Initialization

```
# app/server.py
def _setup_default_routes(self):
    """Setup default routes"""
    self.routes['/'] = lambda req: b"<html><body><h1>Welcome!</h1></body></html>"
    self.routes['/hello'] = lambda req: b"<html><body><h1>Hello, World!</h1></body></html>"
    self.routes['/status'] = lambda req: b'{"status": "ok"}'

def add_route(self, path: str, handler: callable):
    """Add custom route"""
    self.routes[path] = handler
```

Route Table and Default Routes

```
# app/server.py
def _parse_http_request(self, data: bytes) -> Optional[dict]:
    """Parse HTTP request"""
    try:
        text = data.decode('utf-8', errors='ignore')
        lines = text.split('\r\n')

        # Parse request line: GET /path HTTP/1.1
        request_line = lines[0].split(' ')
```

```python
        method = request_line[0]
        path = request_line[1]

        # Parse headers
        headers = {}
        body_start = 0
        for i, line in enumerate(lines[1:], 1):
            if line == '':
                body_start = i + 1
                break
            if ':' in line:
                key, value = line.split(':', 1)
                headers[key.strip()] = value.strip()

        body = '\r\n'.join(lines[body_start:])

        return {
            'method': method,
            'path': path,
            'headers': headers,
            'body': body,
            'raw': data
        }
    except Exception as e:
        return None
```

Request Parsing

```python
# app/server.py
def _build_http_response(self, status_code: int, body: bytes,
                         content_type: str = 'text/html') -> bytes:
    """Build HTTP response"""
    status_messages = {
        200: 'OK',
        404: 'Not Found',
        500: 'Internal Server Error'
    }
    status_msg = status_messages.get(status_code, 'Unknown')

    response = f"HTTP/1.1 {status_code} {status_msg}\r\n"
    response += f"Content-Type: {content_type}\r\n"
    response += f"Content-Length: {len(body)}\r\n"
    response += "Connection: close\r\n"
    response += "\r\n"

    return response.encode('utf-8') + body
```

Response Building

```python
# app/server.py
def _handle_request(self, conn: socket, request: dict):
    """Handle HTTP request and send response"""
    path = request['path']
    method = request['method']

    print(f"[{self.name}] {method} {path}")

    # Route lookup
    if path in self.routes:
        try:
            body = self.routes[path](request)
            content_type = 'application/json' if body.startswith(b'{') else 'text/html'
            response = self._build_http_response(200, body, content_type)
        except Exception as e:
            response = self._build_http_response(500, f"Error: {e}".encode())
    else:
        response = self._build_http_response(404, b"<h1>404 Not Found</h1>")

    conn.send(response)
```

Request Handling with Routing

```python
# app/server.py
def update(self, tick):
    """Non-blocking update loop"""
    super().update(tick)

    # 1. Try to accept new connections
    new_conn = self.listen_socket.accept()
    if new_conn is not None:
        self.connections.append(new_conn)
        print(f"[{self.name}] New connection accepted")

    # 2. Process data from existing connections
    for conn in self.connections[:]:
        data = conn.recv(4096)
        if data:
            request = self._parse_http_request(data)
            if request:
                self._handle_request(conn, request)
```

Non-blocking Update Loop

3.4.3 Usage Example

```python
# test_http.py
from core import PhySimulationEngine
from phy import TwistedPair, Cable
from app.server import HttpServer
from app.client import HttpClient

# Create simulation
simulator = PhySimulationEngine(time_step_us=1)

# Create HTTP server and client
server = HttpServer(simulator=simulator, mac_addr=1, name='server', port=80)
client = HttpClient(simulator=simulator, mac_addr=2, name='client')

# Add custom route
server.add_route('/api/data', lambda req: b'{"value": 42}')

# Connect via cable
cable = Cable(length=100, attenuation=3, noise_level=1)
tp = TwistedPair(cable=cable, simulator=simulator, ID=0)
server.connect_to(tp)
client.connect_to(tp)

# Define callback
def on_response(resp):
    if resp:
        print(f"Got response: {resp['status_code']} - {resp['body']}")

# Send async requests
client.get(dst_mac=1, dst_port=80, path='/', callback=on_response)
client.get(dst_mac=1, dst_port=80, path='/api/data', callback=on_response)
client.get(dst_mac=1, dst_port=80, path='/notfound', callback=on_response)

# Run simulation
simulator.run(duration_ticks=5000)
```

3.4.4 HTTP Message Format Summary

Request Format

```
GET /path HTTP/1.1\r\n
Host: server:port\r\n
Content-Length: n\r\n
Connection: close\r\n
\r\n
[body]
```

Response Format

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: n\r\n
Connection: close\r\n
\r\n
[body]
```

3.4.5 Test Results

TODO

Test GET method   test code:

```python
def test_http_communication():
    """ 测试 HTTP 客户端和服务器通信"""
    simulator = PhySimulationEngine(time_step_us=1)

    server = HttpServer(simulator=simulator, mac_addr=3, name='server', port=80)
    client = HttpClient(simulator=simulator, mac_addr=2, name='client')

    switcher = Switcher(simulator=simulator, mac_addr=0, port_num=2, name='switcher')

    cable = Cable(length=100, attenuation=3, noise_level=3, debug_mode=False)
    tp1 = TwistedPair(cable=cable, simulator=simulator, ID=0)
    tp2 = TwistedPair(cable=cable, simulator=simulator, ID=1)

    server.connect_to(tp1)
    switcher.connect_to(port=0, twisted_pair=tp1)
    client.connect_to(tp2)
    switcher.connect_to(port=1, twisted_pair=tp2)

    server.add_route('/api/data', lambda req: b'{"data": "test_value", "count": 42}')

    responses = []

    def on_response(resp):
        responses.append(resp)
        if resp:
            print(f"Callback with {len(resp['raw'])} byte: status={resp['status_code']}, body={resp['body']

    req1 = client.get(dst_mac=3, dst_port=80, path='/', callback=on_response)
    req2 = client.get(dst_mac=3, dst_port=80, path='/hello', callback=on_response)
    req3 = client.get(dst_mac=3, dst_port=80, path='/api/data', callback=on_response)
    req4 = client.get(dst_mac=3, dst_port=80, path='/not_exist_?', callback=on_response)

    simulator.run(duration_ticks=10000)
```

```
    print("\n=== Test Results ===")
    for i, resp in enumerate(responses):
        if resp:
            print(f"Response {i+1}: {resp['raw']}")

    assert len(responses) == 4, f"Should receive 4 responses but got {len(responses)}"

    print("\n=== HTTP Test Passed ===")
```

```
● (base) apollo•project/network-communication-project/test(async ⚡)» uv run test_http.py

 [server] HTTP Server started on port 80
 [client] HTTP Client initialized
 [server] connected to TwistedPair_0(Channel A, Channel B)
 [client] connected to TwistedPair_1(Channel A, Channel B)
 [client] GET request sent to 3:80/
 [client] GET request sent to 3:80/hello
 [client] GET request sent to 3:80/api/data
 [client] GET request sent to 3:80/not_exist_?
```

Figure 9: *client raises four parallel GET requests*

```
[TICK 10][switcher][MAC] Switcher learned MAC 2 is at port 1
[server] New connection accepted at 80
[TICK 20][switcher][MAC] Switcher learned MAC 3 is at port 0
[TICK 20][switcher][MAC] Switcher MAC table full: {2: 1, 3: 0}
[server] GET /
[server] send Response: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-'...
[client] Response received: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-Length: 43\r\nConnection: cl
ose\r\n\r\n<html><body><h1>Welcome!</h1></body></html>'

Callback with 126 byte: status=200, body=<html><body><h1>Welcome!</h1></body></html>
[server] New connection accepted at 80
[server] GET /hello
[server] send Response: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-'...
[server] New connection accepted at 80
[server] GET /api/data
[server] send Response: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-'...
[server] New connection accepted at 80
[server] GET /not_exist_?
[server] send Response: b'HTTP/1.1 404 Not Found\r\nContent-Type: text/html\r\nC'...
[client] Response received: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-Length: 48\r\nConnection: cl
ose\r\n\r\n<html><body><h1>Hello, World!</h1></body></html>'
```

Figure 10: *server parses and routes the requests respectively*

```
=== Test Results ===
Response 1: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-Length: 43\r\nConnection: close\r\n\r\n<html
><body><h1>Welcome!</h1></body></html>'
Response 2: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-Length: 48\r\nConnection: close\r\n\r\n<html
><body><h1>Hello, World!</h1></body></html>'
Response 3: b'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-Length: 35\r\nConnection: close\r\n\r\n{"dat
a": "test_value", "count": 42}'
Response 4: b'HTTP/1.1 404 Not Found\r\nContent-Type: text/html\r\nContent-Length: 22\r\nConnection: close\r\n\r
\n<h1>404 Not Found</h1>'

=== HTTP Test Passed ===
```

Figure 11: *client receives html pages*

appendix A : project architecture

```
network-communication-project/
|-- core/          # define engine and some base class
|-- phy/           # physics layer
|-- mac/           # MAC layer
|-- tcp/           # transport layer
|-- app/           # app layer
|-- test/          #test files
```

reference

1. Wireless Communication
2. IEEE 802.3 Ethernet Standard
3. RFC 793 - TCP Protocol
4. Computer Networking - a top-down approach