a down-top LAN simulating system

Apollo

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

Feature	Game Loop (Ours)	Event-Driven
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
        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
        [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

```
# 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)

```
# 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:

```
# 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

Test Results

```
Testing Hamming (7,4) Code:
==================================================
Test: Single bit error correction
Original:  [1, 0, 1, 1]
Encoded:   [0, 0, 1, 1, 0, 1, 1]
Corrupted: [0, 0, 1, 1, 0, 0, 1]  (bit 5 flipped)
Decoded:   [1, 0, 1, 1]
Result:    PASS (error corrected)
==================================================
```

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:
        """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:

```
# 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
    )
```

Test Results

```
==================================================
Switch Learning Test
==================================================
[Tick 100] Node1 (MAC=1) sends to Node2 (MAC=2)
  Switcher learned MAC 1 at port 0
  Flooding to port 1, 2 (unknown dst)

[Tick 200] Node2 (MAC=2) replies to Node1 (MAC=1)
  Switcher learned MAC 2 at port 1
  Forwarding to port 0 (known dst)

[Tick 300] Node1 sends again to Node2
  Forwarding to port 1 (known dst)

MAC Table: {1: 0, 2: 1}
==================================================
```

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

        # 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:

```
# 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:

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

```
# 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
        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:

```
# 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
    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

```
# 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

```
# 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)"""
    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

```

# 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

```
==================================================
Transport Layer Test (GBN)
==================================================
[TICK 0] Client CONNECT to 1:80
[TICK 0] NEW SESSION 2:12345 -> 1:80
[TICK 0] SEND DATA seq=0 len=2 (greeting)

[TICK 50] Server RECV DATA seq=0
[TICK 50] SEND ACK ack=1
[TICK 50] ACCEPT new_session peer=2:12345

[TICK 100] Client SEND len=1024
[TICK 100] SEND DATA seq=1 len=1024

[TICK 150] Server RECV DATA seq=1
[TICK 150] SEND ACK ack=2
[TICK 150] Server RECV len=1024

Throughput: ~6.8 KB/s (simulated)
Retransmissions: 0
==================================================
```

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:
    """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"

    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:

```
# 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
```

```
# 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:

```
# 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

        # 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(' ')
        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

```
# 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

```
# 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

```
# 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

```
# 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')
```

```python
# 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

```
==================================================
HTTP Communication Test
==================================================
[server] HTTP Server started on port 80
[client] HTTP Client initialized
```

```
[TICK 100] client sends GET /
[TICK 150] server receives: GET / HTTP/1.1
[TICK 150] server sends: HTTP/1.1 200 OK
[TICK 200] client receives response:
           Status: 200 OK
           Body: <html><body><h1>Welcome!</h1></body></html>

[TICK 300] client sends GET /api/data
[TICK 350] server receives: GET /api/data HTTP/1.1
[TICK 350] server sends: HTTP/1.1 200 OK
[TICK 400] client receives response:
           Status: 200 OK
           Body: {"value": 42}

[TICK 500] client sends GET /notfound
[TICK 550] server receives: GET /notfound HTTP/1.1
[TICK 550] server sends: HTTP/1.1 404 Not Found
[TICK 600] client receives response:
           Status: 404 Not Found
           Body: <h1>404 Not Found</h1>

All requests completed successfully!
==================================================
```

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