Principle of Computer Communication Project 2

Cross-Layer Performance Optimization of a Custom ARQ Protocol

Detailed Implementation Report

Ahmet Enes Cigdem
150220079

January 2026

Abstract

This report presents a comprehensive analysis of the design, implementation, and optimization of a Selective Repeat ARQ protocol within a custom-built network simulator. The simulator uses a Gilbert-Elliott burst-error channel model and evaluates performance based on end-to-end Goodput. Through 360 exhaustive simulations and AI-assisted optimization, optimal protocol parameters were identified and significant performance improvements were achieved.

Contents

1 Introduction

This project implements a comprehensive SR-ARQ network simulator with:

- Cross-layer communication stack (Application, Transport, Link, Physical)
- Realistic burst-error channel (Gilbert-Elliott model)
- Selective Repeat ARQ with sliding windows and per-frame timers
- 360 exhaustive simulations for parameter optimization
- AI-assisted protocol improvement

2 Project Architecture

```
pcom_p2/
 config.py               # Fixed baseline parameters
 run_full_sweep.py       # Parameter sweep (360 simulations)
 simulation/
    simulator.py        # Event-driven simulation engine
    runner.py           # Batch runner for sweeps
 src/
    arq/                # ARQ protocol (sender, receiver, frame)
    channel/            # Gilbert-Elliott channel model
    layers/             # Protocol layers implementation
       application_layer.py  # Data generation, verification
       transport_layer.py    # Segmentation, reassembly
       link_layer.py         # SR-ARQ wrapper
       physical_layer.py     # Delay, channel interface
    utils/              # Metrics and logging
 optimization/           # AI optimization scripts
```

3 Protocol Layers

The simulator implements a layered protocol stack where each layer has specific responsibilities.

3.1 Application Layer (application_layer.py)

Handles data generation and verification for the simulation:

- **Data Generation**: Creates test data (50KB-100MB) with sequential or random patterns
- **Data Verification**: Compares sent vs received data using MD5 checksums
- **Chunked Reading**: Delivers data in chunks to Transport layer

Listing 1: Application Layer - Test Data Generation

```python
class TestDataGenerator:
    @staticmethod
    def generate_test_data(size, pattern="sequential"):
        """Generate test data for simulation."""
        if pattern == "sequential":
            return bytes(i % 256 for i in range(size))
        elif pattern == "random":
            return os.urandom(size)

class DataVerifier:
    @staticmethod
    def verify_data(sent, received):
        """Verify received data matches sent data."""
        return sent == received
```

3.2 Transport Layer (transport_layer.py)

Handles segmentation of application data and reassembly at receiver:

- **Segmentation**: Breaks large data into transport segments (8-byte header)
- **Reassembly**: Reconstructs original data from received segments
- **Flow Control**: 256KB receive buffer with backpressure signaling

Listing 2: Transport Segment Structure

```python
class TransportSegment:
    """
    Header Layout (8 bytes):
      - Segment Number: 4 bytes
      - Payload Length: 2 bytes
      - Flags: 1 byte (last segment indicator)
      - Checksum: 1 byte
    """
    def serialize(self):
        """Pack segment into bytes."""
        header = struct.pack('>IHBx',
            self.segment_num, len(self.payload), self.flags)
        return header + self.payload

    @classmethod
    def deserialize(cls, data):
        """Unpack bytes into segment."""
        header = struct.unpack('>IHBx', data[:8])
        return cls(header[0], data[8:], header[2])
```

3.3 Link Layer (SR-ARQ in src/arq/)

Implements Selective Repeat ARQ protocol:

- **Sliding Window**: Configurable window size (2-64 frames)
- **Per-Frame Timers**: Each frame has individual timeout
- **Selective Retransmission**: Only retransmit corrupted/lost frames
- **Out-of-Order Buffering**: Buffer frames until gaps are filled

Listing 3: Link Layer - Frame Structure

```
class Frame:
    """
    Link Header (24 bytes):
      - Frame Type: 1 byte (DATA=0, ACK=1, NAK=2)
      - Sequence Number: 4 bytes
      - ACK Number: 4 bytes
      - Payload Length: 2 bytes
      - Flags: 1 byte
      - Reserved: 8 bytes
      - CRC-32: 4 bytes
    """
    @property
    def total_size(self):
        return LINK_HEADER_SIZE + len(self.payload)
```

3.4 Physical Layer (physical_layer.py)

Handles transmission timing and channel interface:

- **Transmission Time**: Calculates time based on bit rate (10 Mbps)
- **Propagation Delay**: Asymmetric delays (40ms forward, 10ms reverse)
- **Channel Interface**: Connects to Gilbert-Elliott error model

Listing 4: Physical Layer - Delay Calculation

```
class PhysicalLayer:
    def calculate_total_delay(self, frame_size, direction):
        """Total = Transmission + Propagation + Processing"""
        tx_time = (frame_size * 8) / self.bit_rate
        prop_delay = self.forward_delay if direction == FORWARD \
                     else self.reverse_delay
        return tx_time + prop_delay + self.processing_delay

    def transmit_frame(self, frame, current_time, direction):
        """Transmit frame through Gilbert-Elliott channel."""
        delay = self.calculate_total_delay(frame.total_size,
           direction)
```

```
        corrupted, _ = self.channel.transmit_frame(frame.
            total_size * 8)
        return current_time + delay, corrupted
```

4 Configuration Parameters

All fixed parameters are defined in `config.py`:

Listing 5: Physical Layer Parameters

```
BIT_RATE = 10_000_000           # 10 Mbps
FORWARD_PROPAGATION_DELAY = 0.040   # 40 ms (data)
REVERSE_PROPAGATION_DELAY = 0.010   # 10 ms (ACK)
PROCESSING_DELAY = 0.002        # 2 ms per frame
TRANSPORT_HEADER_SIZE = 8       # bytes
LINK_HEADER_SIZE = 24           # bytes
```

Listing 6: Gilbert-Elliott Channel Parameters

```
GOOD_STATE_BER = 1e-6     # BER in Good state
BAD_STATE_BER = 5e-3      # BER in Bad state
P_GOOD_TO_BAD = 0.002     # Transition G to B
P_BAD_TO_GOOD = 0.05      # Transition B to G
```

Listing 7: Parameter Sweep Configuration

```
WINDOW_SIZES = [2, 4, 8, 16, 32, 64]
PAYLOAD_SIZES = [128, 256, 512, 1024, 2048, 4096]
RUNS_PER_CONFIGURATION = 10
# Total = 6 x 6 x 10 = 360 simulations
```

5 Gilbert-Elliott Channel Model

The channel uses a two-state Markov chain to model burst errors.

5.1 Mathematical Model

Steady-State Probabilities:

$$\pi_G = \frac{P(B \to G)}{P(G \to B) + P(B \to G)} = \frac{0.05}{0.002 + 0.05} = 0.962 \quad (1)$$

$$\pi_B = \frac{P(G \to B)}{P(G \to B) + P(B \to G)} = \frac{0.002}{0.052} = 0.038 \quad (2)$$

Average Bit Error Rate:

$$\text{BER}_{avg} = \pi_G \cdot p_g + \pi_B \cdot p_b = 0.962 \times 10^{-6} + 0.038 \times 5 \times 10^{-3} \approx 1.9 \times 10^{-4} \quad (3)$$

Frame Error Rate: For a frame with n bits:

$$P(\text{frame error}) = 1 - (1 - \text{BER}_{state})^n \quad (4)$$

5.2 Implementation

Listing 8: Gilbert-Elliott Channel - Frame Transmission

```
def transmit_frame(self, frame_size_bits):
    """Simulate frame transmission with bit-by-bit Markov chain."
       ""
    bit_errors = 0

    for _ in range(frame_size_bits):
        # Get BER based on current state
        ber = self.pg if self.state == GOOD else self.pb

        # Check if this bit has error
        if self.rng.random() < ber:
            bit_errors += 1

        # State transition
        if self.state == GOOD:
            if self.rng.random() < self.p_gb:
                self.state = BAD
        else:
            if self.rng.random() < self.p_bg:
                self.state = GOOD

    return bit_errors > 0, bit_errors
```

6 Event-Driven Simulation Engine

The simulator uses an event-driven architecture with a priority queue.

6.1 Event Types

Listing 9: Simulation Events

```
class EventType(Enum):
    DATA_ARRIVAL = 0    # Data frame arrives at receiver
    ACK_ARRIVAL = 1     # ACK arrives at sender
    TIMER_CHECK = 2     # Check for timeouts
```

6.2 Delay Calculations

Forward Delay (data frames):

$$D_{forward} = \frac{\text{FrameSize} \times 8}{\text{BitRate}} + \text{PropDelay}_{forward} + \text{ProcessingDelay} \tag{5}$$

Reverse Delay (ACK frames):

$$D_{reverse} = \frac{\text{ACKSize} \times 8}{\text{BitRate}} + \text{PropDelay}_{reverse} + \text{ProcessingDelay} \tag{6}$$

Round-Trip Time:

$$\text{RTT} = D_{forward} + D_{reverse} \approx 54 \text{ ms} \tag{7}$$

6.3 Main Simulation Loop

Listing 10: Event-Driven Main Loop

```python
def run(self, data):
    """Main simulation loop."""
    self._setup_data(data)
    self._send_frames()  # Initial window fill

    while not self._is_complete():
        event = heapq.heappop(self.event_queue)
        self.current_time = event.time

        if event.event_type == DATA_ARRIVAL:
            self._handle_data_arrival(event.data)
        elif event.event_type == ACK_ARRIVAL:
            self._handle_ack_arrival(event.data)
        elif event.event_type == TIMER_CHECK:
            self._handle_timeouts()

        self._send_frames()

    return self.metrics.get_summary()
```

7 Goodput Calculation

7.1 Mathematical Definition

Goodput is the primary performance metric:

$$\boxed{\text{Goodput} = \frac{\text{Delivered Application Bytes}}{\text{Total Transmission Time}} \quad [\text{bytes/second}]} \tag{8}$$

In bits per second:

$$\text{Goodput}_{bps} = \text{Goodput} \times 8 \tag{9}$$

Efficiency measures useful data ratio:

$$\eta = \frac{\text{Application Bytes Delivered}}{\text{Total Bytes Transmitted}} = \frac{D}{D + H + R} \tag{10}$$

Where: D = Delivered payload, H = Header overhead, R = Retransmission bytes.

Theoretical Maximum Goodput:

$$\text{Goodput}_{max} = R \times \eta_{link} \times (1 - \text{FER}) \times \frac{L}{L + H} \tag{11}$$

Retransmission Rate:

$$\text{Retransmission Rate} = \frac{\text{Retransmissions}}{\text{Original Frames Sent}} \tag{12}$$

7.2 Implementation

Listing 11: Goodput Calculation

```python
class MetricsCollector:
    def calculate_goodput(self):
        """Goodput = Delivered Bytes / Total Time"""
        total_time = self.end_time - self.start_time
        return self.application_bytes_delivered / total_time

    def calculate_efficiency(self):
        """Efficiency = Delivered / Total Transmitted"""
        return self.application_bytes_delivered / \
               self.total_bytes_transmitted
```

8 Batch Runner for 360 Simulations

8.1 Simulation Count

$$\text{Total Simulations} = |\text{Window Sizes}| \times |\text{Payload Sizes}| \times \text{Runs per Config} = 6 \times 6 \times 10 = 360 \tag{13}$$

8.2 Implementation

Listing 12: Batch Runner

```python
class BatchRunner:
    def _generate_run_configs(self):
        """Generate all 360 configurations."""
        configs = []
        for W in self.window_sizes:
            for L in self.payload_sizes:
                for run_id in range(self.runs_per_config):
                    seed = 42 + W*1000 + L + run_id*10000
                    configs.append(RunConfig(W, L, run_id, seed))
        return configs  # 360 configs

    def run_sequential(self):
        """Execute all simulations."""
        for config in self._generate_run_configs():
            result = run_single_simulation(config)
            self.results.append(result)
```

9 Experimental Results

9.1 Goodput Heatmap

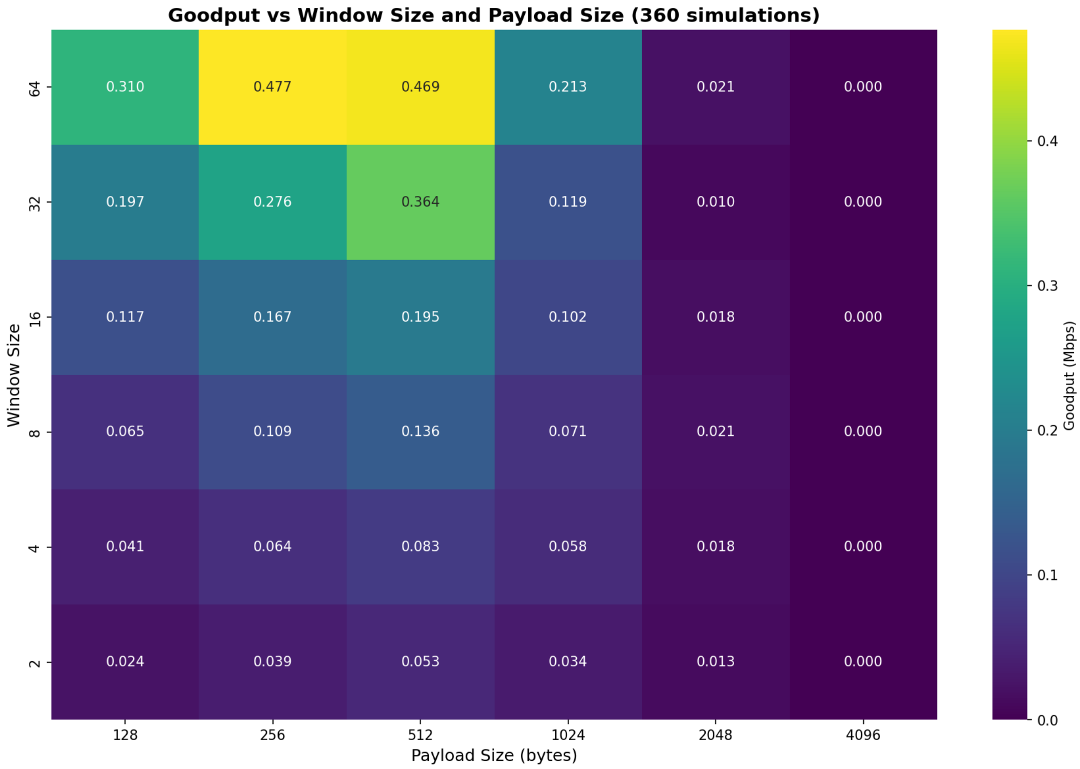

Figure 1: Goodput heatmap from 360 simulations. Brighter = higher Goodput.

9.2 3D Goodput Surface

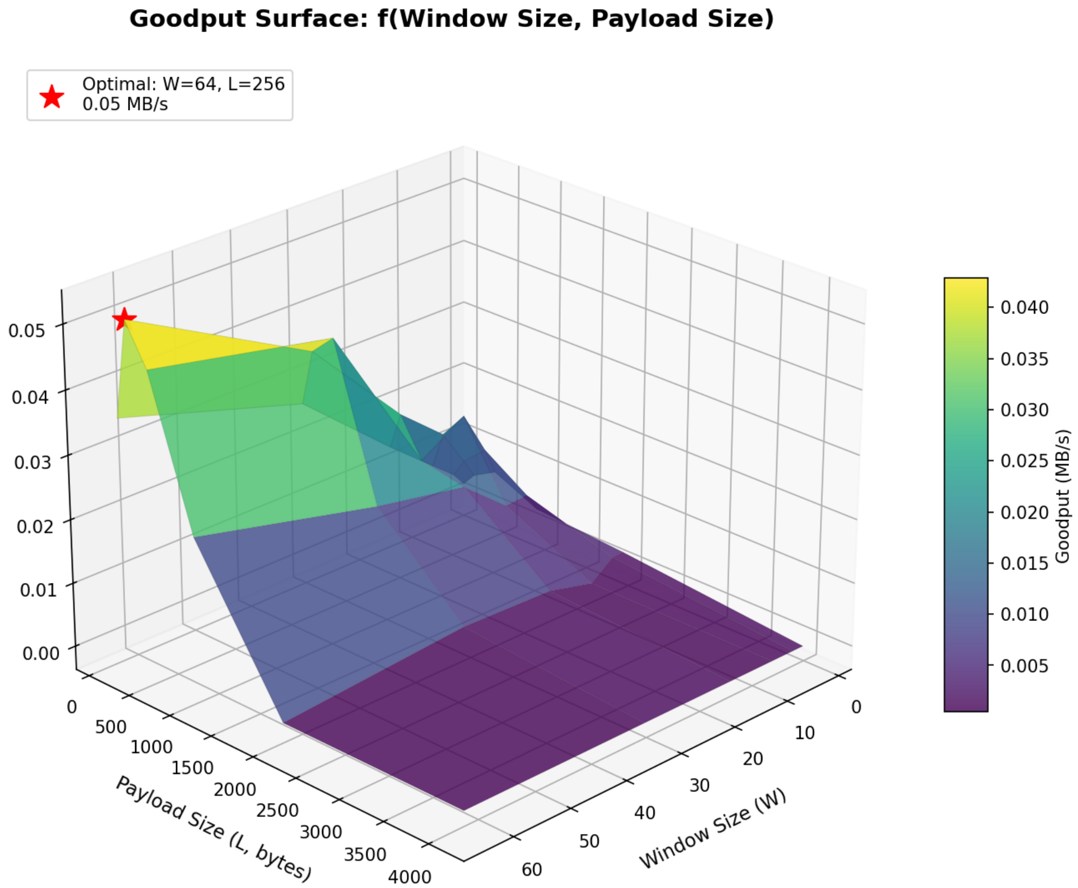

Figure 2: 3D surface plot showing Goodput as a function of Window Size (W) and Payload Size (L). The red star marks the optimal configuration at W=64, L=256.

9.3 Multi-View Surface Analysis

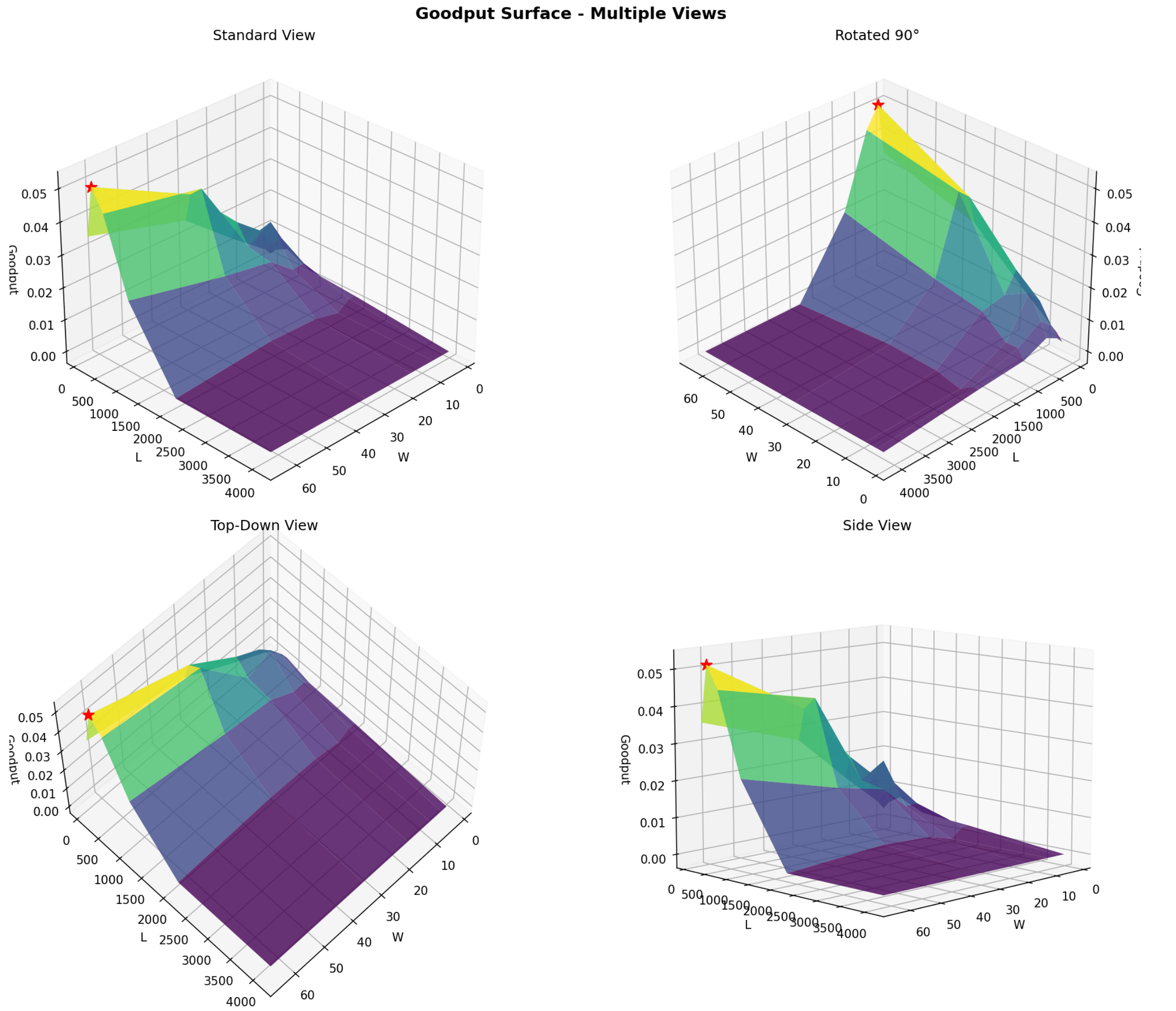

Figure 3: Multiple perspective views of the Goodput surface: Standard View (top-left), Rotated 90° (top-right), Top-Down View (bottom-left), and Side View (bottom-right). These views reveal the performance landscape from different angles.

9.4 Optimal Configuration

$$\boxed{W = 64, \quad L = 256 \text{ bytes}, \quad \text{Goodput} = 0.477 \text{ Mbps}}$$

9.5 Trade-off Analysis

Bandwidth-Delay Product:

$$\text{BDP} = R \times \text{RTT} = 10 \text{ Mbps} \times 54 \text{ ms} = 67,500 \text{ bytes} \tag{14}$$

Optimal Window Size:

$$W_{opt} \geq \frac{\text{BDP}}{L + H} = \frac{67,500}{256 + 32} \approx 234 \text{ frames} \tag{15}$$

10 AI-Assisted Optimization

10.1 Optimization Approach

The AI optimization uses an improved RTO calculation based on Jacobson/Karels:

Default RTO:

$$\text{RTO}_{default} = \text{RTT} \times 2.0 = 54 \text{ ms} \times 2.0 = 108 \text{ ms} \tag{16}$$

AI-Optimized RTO:

$$\text{RTO}_{AI} = \text{RTT} + 4 \times \sigma_{RTT} = 54 + 4 \times 5.4 = 76 \text{ ms} \tag{17}$$

Where $\sigma_{RTT} \approx 0.1 \times \text{RTT}$ is the estimated RTT variance.

10.2 Implementation

Listing 13: AI-Optimized RTO Calculation

```python
def calculate_optimal_timeout():
    """Calculate AI-optimized RTO."""
    rtt = FORWARD_PROP_DELAY + REVERSE_PROP_DELAY + 2*PROC_DELAY
    variance = rtt * 0.1
    return rtt + 4 * variance  # = 76 ms
```

10.3 Comparison Results

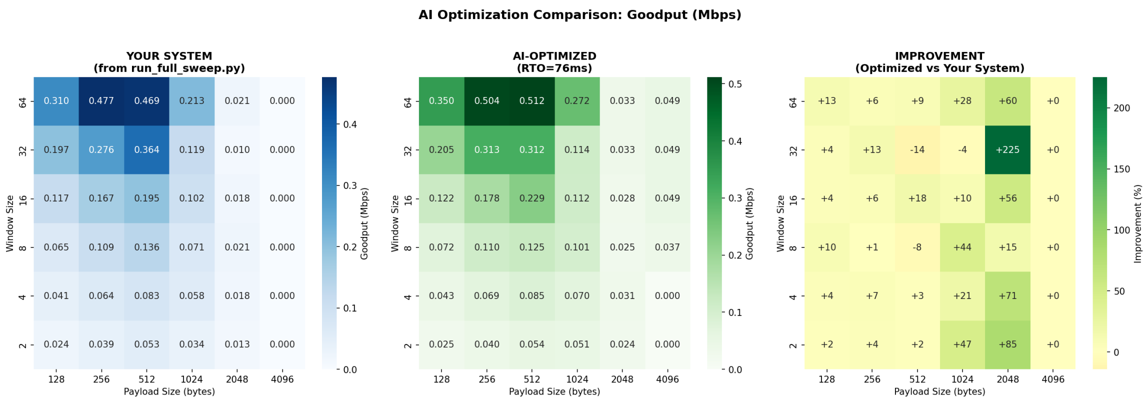

Figure 4: Baseline (left) vs AI-Optimized (center) with improvement % (right).

Table 1: Performance Comparison

Metric	Baseline	AI-Optimized
Average Goodput	0.108 Mbps	0.123 Mbps
RTO	108 ms	76 ms
Improvement	—	**+13.9%**

10.4 Why AI Optimization Works

Lower RTO (76ms vs 108ms) enables:

1. **Faster Recovery**: Quicker detection of lost frames
2. **Reduced Idle Time**: Less waiting for unnecessary timeouts
3. **Better Pipeline Utilization**: Channel stays busy

11 Conclusion

1. **Complete Simulator**: Event-driven SR-ARQ with 4-layer stack
2. **360 Simulations**: $6 \times 6 \times 10$ parameter sweep
3. **Optimal Found**: $W = 64$, $L = 256$ bytes, Goodput = 0.477 Mbps
4. **AI Improvement**: +13.9% via optimized RTO (76ms vs 108ms)