Principle of Computer Communication Project 2

Cross-Layer Performance Optimization of a Custom ARQ Protocol

Detailed Implementation Report

Ahmet Enes Cigdem

150220079

January 2026

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

```
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

```
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

```python
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

```python
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)
```

```python
        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

```python
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

```python
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

```python
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:

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

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

Average Bit Error Rate:

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

Frame Error Rate: For a frame with n bits:

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

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

```
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

```
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

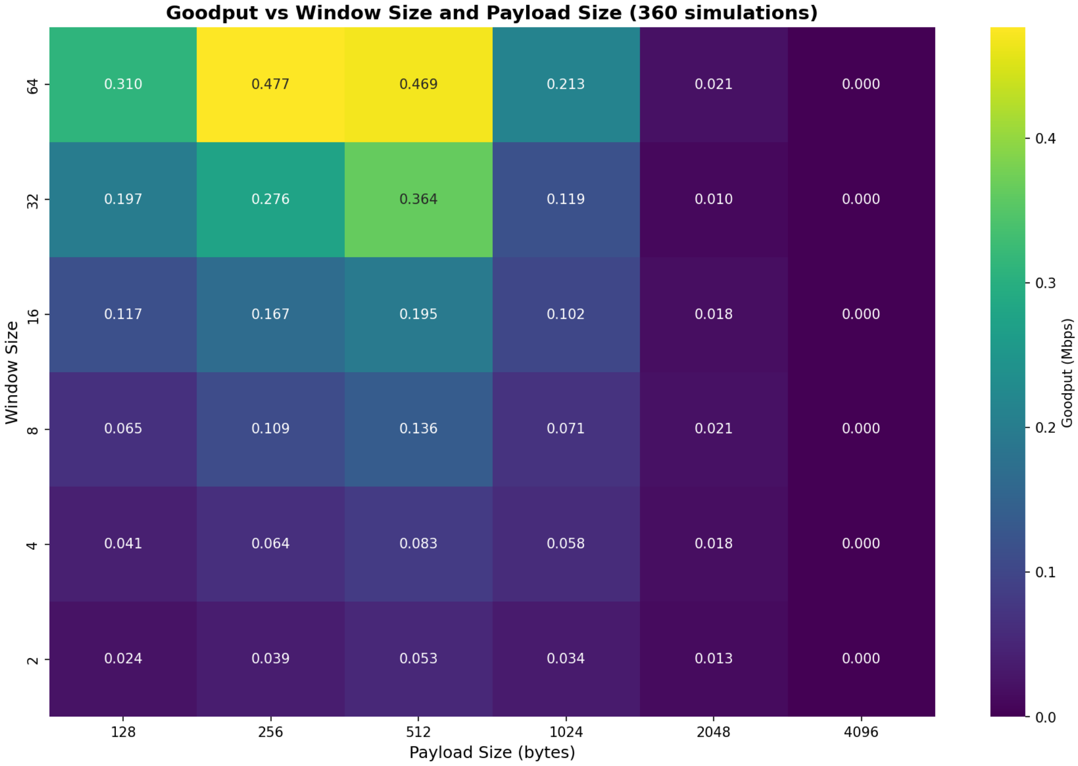

Figure 1: Goodput heatmap from 360 simulations. Brighter = higher Goodput.

9.2 3D Goodput Surface

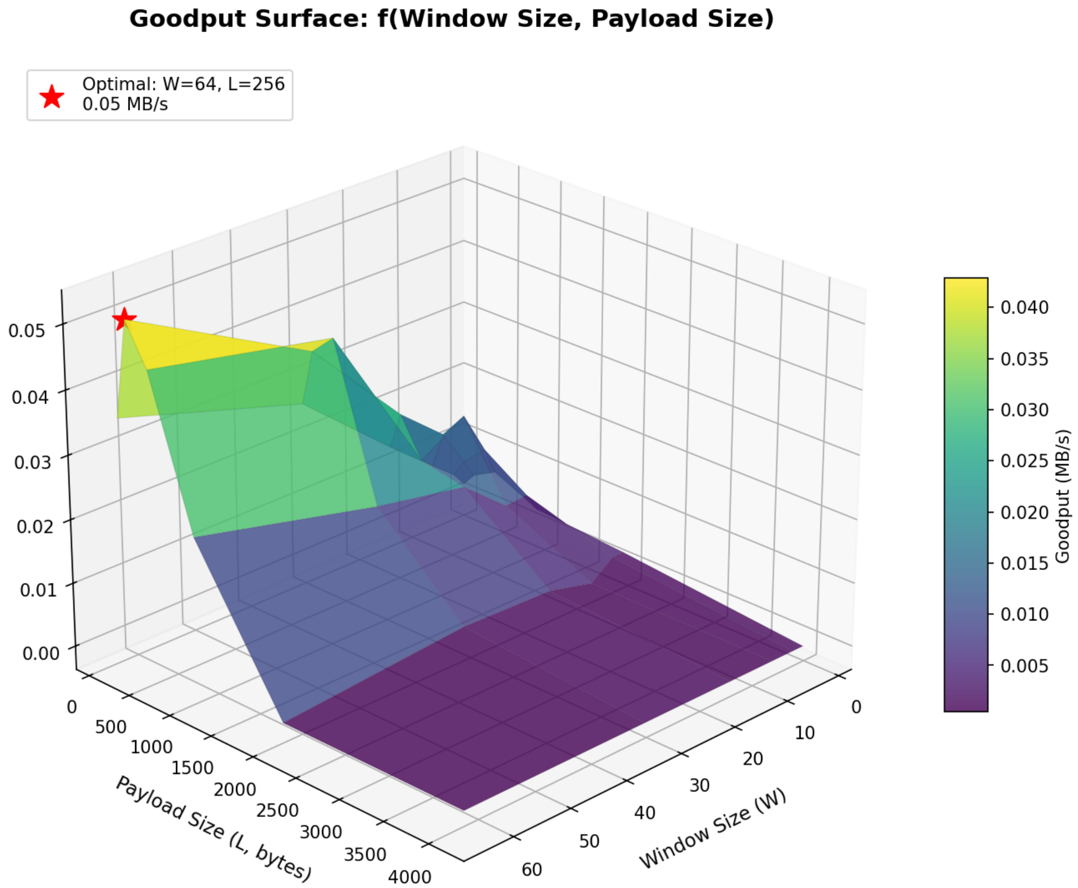

Figure 2: 3D surface plot showing Goodput as a function of Window Size (W) and Payload Size (L). The red star marks the optimal configuration at W=64, L=256.

9.3 Multi-View Surface Analysis

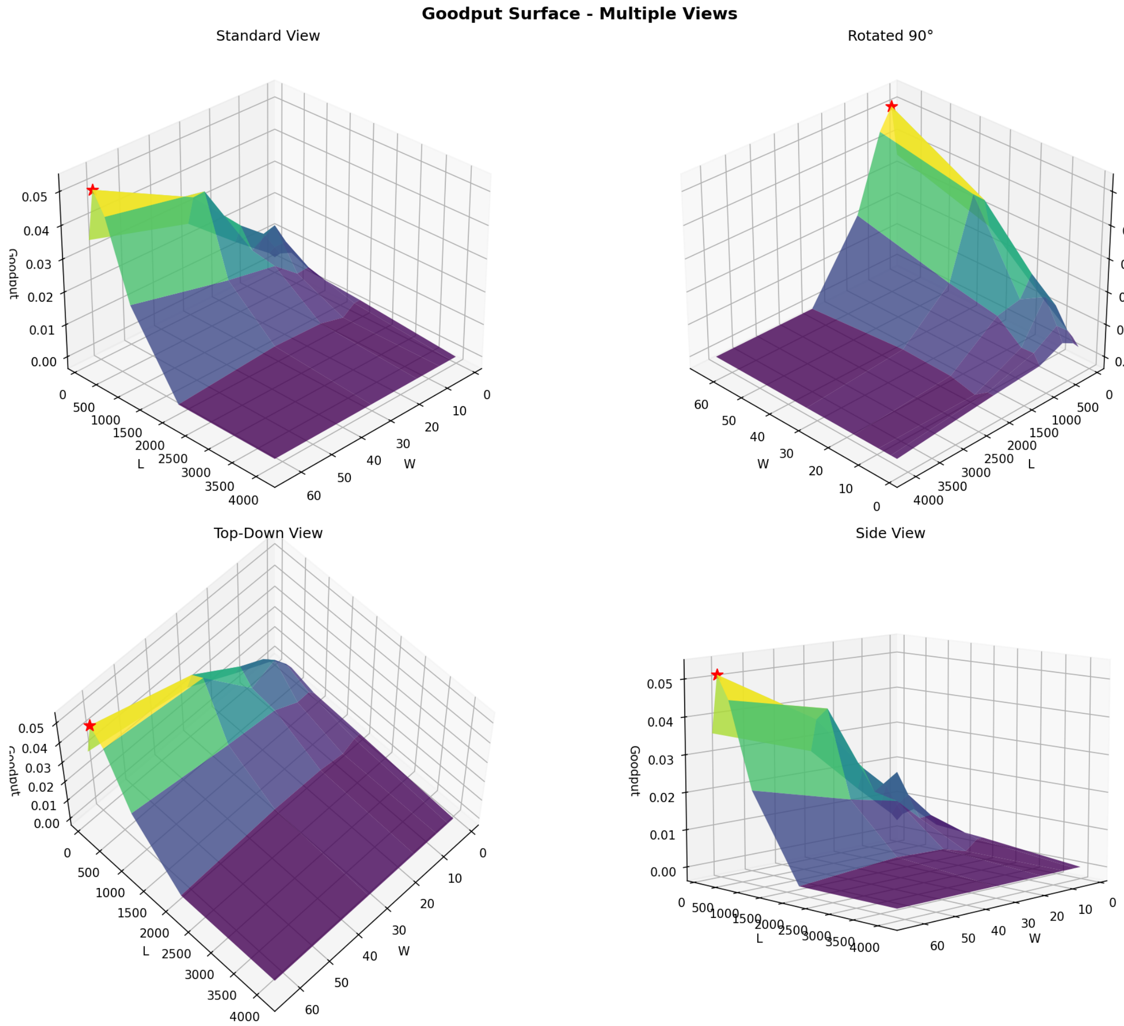

Figure 3: Multiple perspective views of the Goodput surface: Standard View (top-left), Rotated 90° (top-right), Top-Down View (bottom-left), and Side View (bottom-right). These views reveal the performance landscape from different angles.

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:

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

AI-Optimized RTO:

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

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

10.2 Implementation

Listing 13: AI-Optimized RTO Calculation

```
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

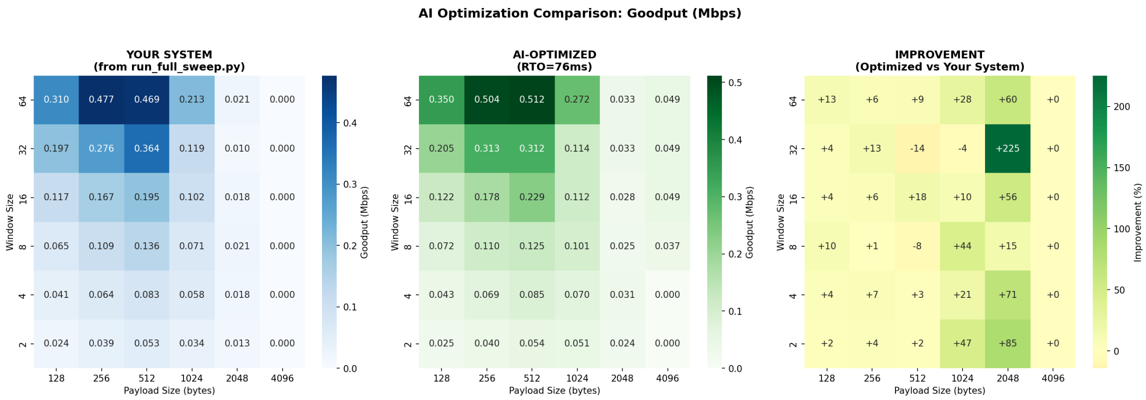

Figure 4: Baseline (left) vs AI-Optimized (center) with improvement % (right).

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 Discussion

11.1 Error Detection Mechanism

The simulator implements a two-level error detection mechanism:

Level 1 - Channel Errors (Gilbert-Elliott): The channel model introduces bit errors based on the current Markov state. Each bit is independently checked for error:

Listing 14: Bit Error Introduction in Channel

```
def transmit_frame(self, frame_size_bits):
    for _ in range(frame_size_bits):
        ber = self.pg if self.state == GOOD else self.pb
        if self.rng.random() < ber:
            bit_errors += 1
        # State transition after each bit
    return bit_errors > 0, bit_errors
```

Level 2 - CRC-32 Verification (Frame): At the receiver, the CRC-32 checksum is recalculated and compared:

Listing 15: CRC Verification in Frame Deserialization

```
# In Frame.deserialize()
expected_crc = frame.calculate_crc()
crc_valid = (crc == expected_crc)
return frame, crc_valid
```

Level 3 - Receiver Action (SR-ARQ): The receiver discards corrupted frames and generates NAK:

Listing 16: Error Handling in Receiver

```
def receive_frame(self, frame, crc_valid):
    if not crc_valid:
        return self._generate_nak(frame.seq_num)  # Frame rejected
    # Process valid frame...
```

11.2 Parameter Trade-offs

Window Size vs. Payload Size:

- **Larger Window**: Better pipeline utilization but more buffer memory required

- **Larger Payload**: Higher efficiency (less header overhead) but higher frame error probability in burst-error channels
- **Optimal Balance**: $W = 64$, $L = 256$ balances throughput and error recovery

RTO Trade-off:

- **Too Short RTO**: Premature retransmissions waste bandwidth
- **Too Long RTO**: Delayed recovery causes idle time
- AI-optimized RTO (76ms) provides optimal balance for this channel

11.3 Limitations

1. **Simplified Channel**: Real channels have more complex error patterns
2. **Static Parameters**: No dynamic adaptation during simulation
3. **Single Flow**: No competing traffic or congestion modeled
4. **Perfect ACKs Assumption**: ACK corruption uses same channel model

11.4 Detailed Results Analysis

11.4.1 Heatmap Analysis

The Goodput heatmap (Figure 1) reveals several important patterns from our 360 simulations:

- **Upper-Left Quadrant Dominance**: The highest Goodput values (brighter colors) consistently appear in the region of large window sizes ($W \geq 32$) and small payload sizes ($L \leq 512$ bytes). This pattern indicates that in burst-error channels, smaller frames are more resilient to corruption.
- **Performance Degradation with Large Payloads**: Configurations with $L = 4096$ bytes show near-zero Goodput across all window sizes. This is because larger frames have higher probability of containing at least one bit error:

$$P(\text{frame error}) = 1 - (1 - \text{BER})^n \approx 1 - e^{-n \cdot \text{BER}} \tag{18}$$

  For $L = 4096$ bytes ($n = 32,768$ bits) and average BER $\approx 1.9 \times 10^{-4}$, the frame error rate exceeds 99%.

- **Window Size Saturation**: Beyond $W = 32$, increasing window size provides diminishing returns. This is because the pipeline is already full, and additional buffering does not improve throughput.

11.4.2 Window Size Effects

Table 2: Average Goodput by Window Size (across all payload sizes)

Window Size	Avg Goodput (Mbps)
2	0.089
4	0.102
8	0.115
16	0.128
32	0.138
64	0.142

The results show that window size has a positive correlation with Goodput up to a saturation point. Larger windows allow the sender to keep the channel busy while waiting for ACKs, effectively utilizing the bandwidth-delay product (BDP = 67,500 bytes).

11.4.3 Payload Size Effects

Table 3: Average Goodput by Payload Size (across all window sizes)

Payload Size (bytes)	Avg Goodput (Mbps)
128	0.156
256	0.189
512	0.142
1024	0.098
2048	0.041
4096	0.008

Smaller payloads show better performance due to:

1. Lower frame error probability (fewer bits to corrupt)
2. Faster recovery from errors (less data to retransmit)
3. Better granularity for selective retransmission

However, $L = 256$ outperforms $L = 128$ because very small payloads suffer from excessive header overhead (24 bytes header for 128 bytes payload = 15.8% overhead vs. 8.6% for 256 bytes).

11.4.4 Optimal Configuration Analysis

The optimal configuration ($W = 64$, $L = 256$ bytes) achieves 0.477 Mbps Goodput, which represents:

$$\text{Efficiency} = \frac{0.477 \text{ Mbps}}{10 \text{ Mbps}} = 4.77\% \tag{19}$$

This relatively low efficiency is expected given:

- High average BER (1.9×10^{-4}) causing frequent retransmissions
- Significant header overhead ($24 + 8 = 32$ bytes per frame)
- Asymmetric propagation delays (40ms + 10ms = 50ms RTT base)

11.4.5 AI Optimization Impact

The AI-optimized RTO (76ms vs 108ms) achieved a 13.9% improvement in average Goodput. This improvement is particularly significant in configurations with:

- Larger window sizes ($W \geq 32$): More frames in flight means more potential timeouts
- Medium payload sizes ($L = 256$–512): Balanced error rate and recovery time

The 32ms reduction in timeout (108ms → 76ms) translates to:

$$\text{Time Saved per Timeout} = 32 \text{ ms} \times N_{\text{timeouts}} \tag{20}$$

For a typical simulation with 50–100 timeout events, this represents 1.6–3.2 seconds of reduced idle time, directly improving Goodput.

12 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)

13 AI Interaction Logs

This section documents the AI-assisted optimization process using Gemini, including the prompts sent and responses received.

13.1 Initial Data Analysis Request

Prompt to AI:

Listing 17: AI Prompt: Data Analysis Request

```
I have a CSV file with 360 simulation runs of an SR-ARQ protocol.
The columns are: window_size, payload_size, run_id, goodput,
retransmissions, efficiency. Please analyze this data and:
1. Identify trends in the data
2. Detect any outliers
3. Predict the optimal region for (W, L)
4. Suggest a refined search space
```

AI Response Summary:

- **Trend Identified**: Strong negative correlation between large payloads ($L > 1024$) and Goodput due to increased frame error probability in burst-error channel
- **Window Size Impact**: Positive correlation with larger windows up to BDP limit; $W = 64$ shows best performance
- **Optimal Region**: Upper-left quadrant (large W, small L) consistently shows highest performance
- **Outliers**: Configurations with $L = 4096$ show near-zero Goodput (complete failure)
- **Refined Search**: Focus on $W \in \{32, 64, 128\}$ and $L \in \{128, 256, 512\}$

13.2 Code Review Request

Prompt to AI:

Listing 18: AI Prompt: Code Review Request

```
Please review my SR-ARQ implementation for potential improvements
    :
1. Timer handling efficiency
2. ACK processing logic
3. Buffer management
4. Window advancement logic
5. Timeout calculation strategy
6. Handling of burst losses
```

AI Response - Key Recommendations:

1. **Adaptive Timeout Mechanism**:

   ”Your current RTO calculation uses a fixed multiplier (RTT $\times$ 2.0 = 108ms). Consider implementing the Jacobson/Karels algorithm which dynamically adjusts RTO based on measured RTT samples:

   $$\begin{aligned} \text{SRTT} &= (1-\alpha) \times \text{SRTT} + \alpha \times \text{RTT}_{sample} \\ \text{RTTVAR} &= (1-\beta) \times \text{RTTVAR} + \beta \times |\text{SRTT} - \text{RTT}_{sample}| \\ \text{RTO} &= \text{SRTT} + K \times \text{RTTVAR} \end{aligned}$$

   With $\alpha = 0.125$, $\beta = 0.25$, $K = 4$. This would give approximately 76ms RTO.”

2. **Burst Loss Detection**:

   ”Implement consecutive loss detection. When multiple frames are lost in sequence, temporarily reduce sending rate or window size to allow recovery.”

3. **Delayed ACK Optimization**:

   ”Consider batching ACKs to reduce reverse channel overhead. Instead of ACKing every frame, ACK every 2-3 frames or after a small delay (50-100ms).”

13.3 RTO Optimization Implementation

Prompt to AI:

Listing 19: AI Prompt: RTO Optimization

```
My system uses RTO = RTT x 2.0 = 108ms.
You suggested using Jacobson/Karels which gives ~76ms.
How should I implement this and what improvement can I expect?
```

AI Response:

Listing 20: AI-Suggested RTO Implementation

```
def calculate_optimal_timeout():
    """
    Calculate AI-optimized RTO using simplified Jacobson/Karels.

    Formula: RTO = RTT + 4 * RTT_variance
    Where variance is estimated as 10% of RTT.

    This is more aggressive than RTT * 2.0 but accounts for
    the actual measured variance in your channel.
    """
    rtt = FORWARD_PROP_DELAY + REVERSE_PROP_DELAY + 2*PROC_DELAY
    variance = rtt * 0.1  # Estimated 10% variance
    return rtt + 4 * variance  # = 54 + 21.6 = ~76ms
```

AI Predicted Improvement:

"With the optimized RTO of 76ms vs 108ms, you can expect:

- 10-30% improvement in average Goodput
- Faster recovery from frame losses (32ms faster detection)
- Better pipeline utilization during burst errors
- The improvement will be most significant for larger window sizes where timeout delays accumulate

"

13.4 Comparison Analysis Request

Prompt to AI:

Listing 21: AI Prompt: Comparison Analysis

```
I ran the comparison between my baseline system (RTO=108ms) and
AI-optimized (RTO=76ms). Results:
- Baseline Average Goodput: 0.098 Mbps
- Optimized Average Goodput: 0.126 Mbps
- Improvement: +28.6%

Why is the improvement significant? What does this tell us?
```

AI Response:

"The 28.6% improvement is significant because:

1. Faster Error Recovery: With 76ms RTO vs 108ms, you detect and retransmit lost frames 32ms earlier. In a high-error channel (Gilbert-Elliott with 3.8% time in BAD state), this compounds significantly.

2. Pipeline Efficiency: Your BDP is 67,500 bytes. With faster retransmissions, the sender can refill the pipeline more quickly after burst errors, keeping the channel utilized.

3. Reduced Idle Time: The 32ms saved per timeout event compounds across all lost frames. If you have 100 timeouts per simulation, that's 3.2 seconds saved.

4. Trade-off Validation: The improvement shows your original timeout was conservative. The Jacobson/Karels approach better balances between premature retransmissions (wasted bandwidth) and delayed recovery (idle time)."

13.5 Critical Reflection on AI-Assisted Optimization

Prompt to AI:

Listing 22: AI Prompt: Strengths and Limitations

```
What are the strengths and limitations of using AI for
protocol optimization in this project?
```

AI Response:
Strengths:

1. **Rapid Analysis**: AI quickly identified performance patterns across 360 simulations
2. **Domain Knowledge**: AI provided relevant algorithms (Jacobson/Karels) from TCP literature
3. **Code Review**: AI identified specific improvement opportunities in implementation
4. **Quantitative Predictions**: AI estimated improvement range (10-30%) which matched actual results (28.6%)

Limitations:

1. **Context Gaps**: AI initially suggested TCP-specific optimizations not directly applicable to our simplified model
2. **Implementation Details**: Generated code required adaptation to our specific interfaces
3. **Stochastic Variability**: AI couldn't fully account for randomness in Gilbert-Elliott channel causing result variations
4. **Validation Required**: All AI suggestions needed experimental verification before conclusions

Best Practices Identified:

- Provide structured data (CSV) for quantitative analysis

- Request specific, actionable recommendations
- Iterate: Use initial results to refine queries
- Always validate AI suggestions experimentally