URP (UDP-based Reliable Protocol) Implementation Report

Course: COMP3331/9331 Computer Networks and Applications
Term: Term 3, 2025
Programming Language: Python 3.11
Author: Bin Huang **ZID:** z5610468

---

1. Project Overview

1.1 Programming Language and Environment

- **Language:** Python 3.11
- **Development Environment:** VLAB (CSE Linux)
- **Testing Platform:** CSE Linux machines

1.2 File Organization

```
project/
├── sender.py              # Sender implementation (main program)
├── receiver.py            # Receiver implementation (main program)
├── urp.py                 # URP segment structure and utilities
├── timer.py               # Independent timer module
├── plc.py                 # Packet Loss and Corruption module
├── logger.py              # Logging module
└── logs/                  # Log files directory (auto-generated)
    ├── sender_log.txt
    └── receiver_log.txt
```

1.3 Execution Instructions

Start Receiver:

```
python3 receiver.py <receiver_port> <sender_port> <output_file> <max_win>
```

Start Sender:

```
python3 sender.py <sender_port> <receiver_port> <input_file> <max_win> <rto> <flp> <rlp> <fcp> <rcp>
```

Example:

```
# Terminal 1
python3 receiver.py 56007 59606 received.txt 3000

# Terminal 2
python3 sender.py 59606 56007 test.txt 3000 100 0.1 0.05 0.1 0.05
```

---

2. URP Protocol Implementation

2.1 Segment Format

The URP segment consists of a 6-byte fixed header followed by optional data payload:

```
1                 2
0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 8
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Sequence Number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Reserved        |A|S|F|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Data (0-1000 bytes)     |
+                                 +
```

Field Descriptions:

- **Sequence Number (16 bits):** Byte-stream position or acknowledgment number
- **Reserved (13 bits):** Set to zero
- **Control Flags (3 bits):** ACK, SYN, FIN (mutually exclusive)
- **Checksum (16 bits):** Error detection field
- **Data (0-1000 bytes):** Payload data (MSS = 1000 bytes)

Implementation (`urp.py`):

```python
def pack(self):
    reserved_flags = self.flags & 0b00000111
    header = struct.pack("!HHH", self.seq_num, reserved_flags, 0)
    segment = header + self.data
    self.checksum = self._calculate_checksum(segment)
    header = struct.pack("!HHH", self.seq_num, reserved_flags, self.checksum)
    return header + self.data
```

2.2 Error Detection Mechanism

Algorithm: 16-bit One’s Complement Checksum (TCP/UDP style)

Calculation Process:

1. Divide segment (header + data) into 16-bit words
2. Sum all words with end-around carry
3. Take one’s complement of the sum

Implementation:

```python
def _calculate_checksum(data):
    if len(data) % 2 == 1:
        data += b"\x00"  # Pad odd-length data

    total_sum = 0
    for i in range(0, len(data), 2):
        word = (data[i] << 8) + data[i + 1]
        total_sum += word
        total_sum = (total_sum & 0xFFFF) + (total_sum >> 16)

    return ~total_sum & 0xFFFF
```

Verification:

```python
def verify_checksum(self):
    reserved_flags = self.flags & 0b00000111
    header = struct.pack("!HHH", self.seq_num, reserved_flags, 0)
```

```python
    checksum = self._calculate_checksum(header + self.data)
    return checksum == self.checksum
```

Advantages:

- Detects all single-bit errors
- Detects most burst errors
- Low computational overhead
- Industry-standard (used in TCP/UDP)

---

3. Program Design

3.1 Overall Architecture

```
┌──────────────────────────────────────────────────────────────┐
│                    Sender Architecture                       │
├──────────────────────────────────────────────────────────────┤
│  Main Thread: Connection setup/teardown, coordination        │
│                                                              │
│  Thread 1: Send Thread                                       │
│    - Monitor window space availability                       │
│    - Send DATA segments when window permits                  │
│    - Triggered by: window_available_event                    │
│                                                              │
│  Thread 2: Receive Thread                                    │
│    - Continuously receive ACK segments                       │
│    - Validate checksum                                       │
│    - Update sliding window                                   │
│    - Trigger fast retransmission on 3 duplicate ACKs         │
│                                                              │
│  Thread 3: Timer Thread (in timer module)                    │
│    - Monitor timeout for oldest unACKed segment              │
│    - Trigger timeout retransmission                          │
│    - Auto-restart after timeout                              │
│                                                              │
│  PLC Module: Packet Loss and Corruption emulation            │
│    - Forward direction: DATA, SYN, FIN segments              │
│    - Reverse direction: ACK segments                         │
└──────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────┐
│                  Receiver Architecture                       │
├──────────────────────────────────────────────────────────────┤
│  Main Thread: Single-threaded event loop                     │
│    - Receive and validate segments                           │
│    - Process SYN, DATA, FIN segments                         │
│    - Generate and send ACKs immediately                      │
│    - Manage receive buffer for out-of-order data             │
│    - Write in-order data to file                             │
│                                                              │
│  Timer Thread: TIME_WAIT timer                               │
│    - Started when receiving FIN                              │
│    - Duration: 2 * MSL = 2 seconds                           │
│    - Transition to CLOSED after expiration                   │
└──────────────────────────────────────────────────────────────┘
```

3.2 Data Structure Design

3.2.1 Sender Data Structures

Send Buffer (Sliding Window):

```python
self.send_buffer = {
    seq_num1: (segment, send_time),
    seq_num2: (segment, send_time),
    ...
}
```

- **Purpose:** Store unacknowledged segments for retransmission
- **Key:** Sequence number
- **Value:** Tuple of (segment object, send timestamp)
- **Thread Safety:** Protected by `buffer_lock`

Window Variables:

```python
self.base = ...           # Left edge of window (oldest unACKed)
self.next_seq_num = ...   # Right edge of window (next to send)
self.max_win = ...        # Maximum window size in bytes
```

State Variables:

```python
self.state = ...          # Connection state (CLOSED/SYN_SENT/ESTABLISHED/CLOSING/FIN_WAIT)
self.isn = ...            # Initial sequence number (random 0-65535)
self.dup_ack_count = ...  # Counter for duplicate ACKs
self.last_ack_num = ...   # Last ACK received
```

3.2.2 Receiver Data Structures

Receive Buffer:

```python
self.recv_buffer = {
    seq_num1: data1,
    seq_num2: data2,
    ...
}
```

- **Purpose:** Store out-of-order DATA segments
- **Key:** Sequence number
- **Value:** Payload data bytes

State Variables:

```python
self.state = ...               # Connection state (CLOSED/LISTEN/ESTABLISHED/TIME_WAIT)
self.expected_seq_num = ...    # Next expected byte number
self.received_seq_num = set()  # Set of received sequence numbers (for duplicate detection)
```

3.3 Modular Design

3.3.1 Timer Module (`timer.py`)

Key Features:

- **Decoupled Design:** Independent module, not tied to Sender
- **Thread-Safe:** Internal locking mechanism
- **Auto-Restart Option:** Configurable automatic restart after timeout
- **Simple API:** `start()` , `stop()` , `restart()`

Usage in Sender:

```python
self.timer = Timer(
    timeout=self.rto,
    callback=self.handle_timeout,
    auto_restart=True
)

# Start timer when first segment is sent
self.timer.start()

# Restart timer when new ACK arrives
self.timer.restart()

# Stop timer when all segments are ACKed
self.timer.stop()
```

Benefits:

- Reduces Sender code by ~40 lines
- Improves testability
- Reusable for other components (e.g., TIME_WAIT timer in Receiver)

3.3.2 PLC Module (`plc.py`)

Purpose: Emulate unreliable channel behavior

Forward Direction Processing:

```python
def process_fd(self, data):
    if random.random() < self.flp:
        return None, "drp"  # Drop
    if random.random() < self.fcp:
        return corrupted_data, "cor"  # Corrupt
    return data, "ok"  # Pass through
```

Reverse Direction Processing:

```python
def process_bk(self, data):
    if random.random() < self.rlp:
        return None, "drp"  # Drop
    if random.random() < self.rcp:
        return corrupted_data, "cor"  # Corrupt
    return data, "ok"  # Pass through
```

Corruption Method:

- Skip first 4 bytes (for logging simplification)
- Randomly select a byte
- Randomly flip one bit in selected byte

3.3.3 Logger Module (`logger.py`)

Features:

- Real-time logging (flush after each write)
- Automatic timestamp calculation
- Consistent log format

Log Format:

```
<direction> <status> <time> <segment-type> <seq-number> <payload-length>
```

---

4. Sender Implementation Details

4.1 Connection Setup (Two-Way Handshake)

```
Sender                                 Receiver
  |                                       |
  |------- SYN (seq=ISN) ---------->|
  |                                       |
  |<------ ACK (seq=ISN+1) ---------|
  |                                       |
[ESTABLISHED]                    [ESTABLISHED]
```

Implementation:

```python
def connection_setup(self):
    self.state = STATE_SYN_SENT
    syn_segment = UrpSegment(self.isn, FLAG_SYN, b"")
    self.send_segment(syn_segment, is_new=True)

    while self.state != STATE_ESTABLISHED:
        self.check_for_ack(timeout=0.01)
        self.checktimeout()  # Retransmit SYN if timeout
```

4.2 Data Transfer (Sliding Window)

Window Management:

```python
def _calculate_window_usage(self):
    total_bytes = 0
    for seq_num, (segment, _) in self.send_buffer.items():
        if segment.flags == FLAG_DATA:
            total_bytes += len(segment.data)
    return total_bytes
```

Send Thread Logic:

```python
def send_data_thread(self):
    while self.running:
        self.window_available_event.wait(timeout=0.01)
        if self.state != STATE_ESTABLISHED:
            break
        self.try_send_data()
```

Incremental File Reading:

```python
def try_send_data(self):
    """Try to send data"""
    assert self.file_handle is not None
    while True:
        # Check if the window has available space
        with self.buffer_lock:
            window_used = self._calculate_window_usage()
            if window_used >= self.max_win:
                self.window_available_event.clear()
                return

        # Check if there is any more data to send
        with self.file_lock:
            if self.file_pos >= self.file_size:
                self.all_data_sent = True
                return
```

```
        # Read data
        read_length = min(MSS, self.file_size - self.file_pos)
        data = self.file_handle.read(read_length)
        self.file_pos += len(data)
        ...
```

> Key Design:
>
> - File remains open throughout transmission
> - Data read on-demand (1000 bytes at a time)
> - Memory usage: O(window_size) instead of O(file_size)
> - Supports arbitrarily large files

4.3 Reliability Mechanisms

4.3.1 Timeout Retransmission

Single Timer Approach:

- Only one timer for the oldest unACKed segment
- When timeout occurs, retransmit oldest segment only
- Timer automatically restarts after retransmission

```python
def handle_timeout(self):
    with self.buffer_lock:
        if not self.send_buffer:
            return

        # Retransmit oldest segment (at self.base)
        segment, _ = self.send_buffer[self.base]
        self.send_segment(segment, is_new=False)
        self.stats['timeout_retrans'] += 1

        # Update send time
        self.send_buffer[self.base] = (segment, time.time())
```

4.3.2 Fast Retransmission

Trigger: 3 duplicate ACKs

```python
def process_ack(self, segment):
    ack_num = segment.seq_num

    if self._is_new_ack(ack_num):
        # Update window, restart timer
        self.base = ack_num
        self.dup_ack_count = 0
        self.timer.restart()
    else:
        # Duplicate ACK
        if ack_num == self.last_ack_num:
            self.dup_ack_count += 1
            if self.dup_ack_count == 3:
                self.retrans_oldest("fast_retrans")
                self.dup_ack_count = 0
```

4.3.3 Cumulative Acknowledgment

Processing:

```python
# Remove all segments with seq_num < ack_num
segments_remove = []
for seq in self.send_buffer.keys():
    if self._is_before(seq, ack_num):
        segments_remove.append(seq)

for seq in segments_remove:
    del self.send_buffer[seq]
```

Advantage: Single ACK can acknowledge multiple segments

4.4 Connection Teardown

```
Sender                               Receiver
  |                                     |
  |------- FIN (seq=X) ------------>|
  |                                     |
  |<------ ACK (seq=X+1) -----------|
  |                                     |
[CLOSED]                            [TIME_WAIT]
                                        |
                                    (wait 2 seconds)
                                        |
                                      [CLOSED]
```

Implementation:

```python
def connection_close(self):
    # Wait for all data to be ACKed
    while self.send_buffer:
        time.sleep(0.01)

    # Send FIN
    self.state = STATE_FIN_WAIT
    fin_segment = UrpSegment(self.next_seq_num, FLAG_FIN, b"")
    self.send_segment(fin_segment, is_new=True)

    # Wait for ACK (with retransmission on timeout)
    while self.state != STATE_CLOSED:
        self.check_for_ack(timeout=0.01)
        self.checktimeout()
```

---

5. Receiver Implementation Details

5.1 Single-Threaded Design

Rationale:

- Receiver logic is simpler than Sender
- No need for concurrent transmission
- Single event loop sufficient for receiving and processing

Main Loop:

```python
def run(self):
    self.state = STATE_LISTEN
    while self.running and self.state != STATE_CLOSED:
        self.receive_segment()
        time.sleep(0.001)
```

5.2 State Machine

LISTEN → ESTABLISHED:

```python
def handle_listen(self, segment):
    if segment.flags & FLAG_SYN:
        self.expected_seq_num = (segment.seq_num + 1) % MAX_SEQ_NUM
        self.send_ack(self.expected_seq_num)
        self.state = STATE_ESTABLISHED
```

ESTABLISHED → TIME_WAIT:

```python
def handle_established(self, segment):
    if segment.flags & FLAG_FIN:
        ack_num = (segment.seq_num + 1) % MAX_SEQ_NUM
        self.send_ack(ack_num)
        self.state = STATE_TIME_WAIT
        self.timer.start()  # 2-second timer
```

5.3 Out-of-Order Data Handling

Strategy:

1. In-order data → Write to file immediately
2. Out-of-order data → Buffer until gap is filled
3. Flush buffer when contiguous data available

Implementation:

```python
def process_data_segment(self, segment):
    if segment.seq_num == self.expected_seq_num:
        # In-order: write immediately
        self.output_file.write(segment.data)
        self.expected_seq_num = (self.expected_seq_num + len(segment.data)) % MAX_SEQ_NUM
        self.flush_buffer()
    else:
        # Out-of-order: buffer
        self.recv_buffer[segment.seq_num] = segment.data

    self.send_ack(self.expected_seq_num)

def flush_buffer(self):
    while self.expected_seq_num in self.recv_buffer:
        data = self.recv_buffer[self.expected_seq_num]
        self.output_file.write(data)
        del self.recv_buffer[self.expected_seq_num]
        self.expected_seq_num = (self.expected_seq_num + len(data)) % MAX_SEQ_NUM
```

5.4 Immediate ACK Generation

Policy: Generate ACK immediately upon receiving any segment (no delayed ACKs)

```python
def send_ack(self, ack_num):
    ack_segment = UrpSegment(ack_num, FLAG_ACK, b"")
    segment_data = ack_segment.pack()
    self.sock.sendto(segment_data, (self.server_ip, self.sender_port))
    self.log.log_segment("snd", "ok", ack_segment, 0)
    self.stats["total_acks_sent"] += 1
```

6. Thread Synchronization

6.1 Locks Used in Sender

Lock	Purpose	Protected Resources
`state_lock`	Protect connection state	`self.state`
`buffer_lock`	Protect send buffer	`self.send_buffer` , `self.base` , `self.next_seq_num`
`ack_lock`	Protect ACK variables	`self.dup_ack_count` , `self.last_ack_num`
`file_lock`	Protect file reading	`self.file_pos` , `self.file_data`
`log_lock`	Protect log file writes	`self.log.log_file`
`stats_lock`	Protect statistics	`self.stats` dictionary

6.2 Events Used

Event	Purpose	Set By	Waited By
`window_available_event`	Signal window space available	Receive thread (when ACK arrives)	Send thread
`timer._cancel_event`	Cancel timer	Timer.stop()	Timer thread

6.3 Deadlock Prevention

Strategy:

- Consistent lock ordering
- Use `RLock` (reentrant lock) for nested locking
- Release locks before calling external functions
- Short critical sections

Example:

```python
# Correct: acquire locks in consistent order
with self.buffer_lock:
    with self.stats_lock:
        # Process data

# Avoid: calling callback while holding lock
with self.lock:
    # ... do work ...
# Release lock before callback
self.callback()
```

7. Testing and Validation

7.1 Testing Strategy

Progressive Testing Approach:

1. **Phase 1:** Stop-and-Wait (max_win=1000) + Reliable channel (all probabilities=0)
2. **Phase 2:** Stop-and-Wait + Loss only (flp/rlp > 0)

3. **Phase 3:** Stop-and-Wait + Corruption only (fcp/rcp > 0)
4. **Phase 4:** Stop-and-Wait + Combined loss and corruption
5. **Phase 5:** Sliding Window + Reliable channel
6. **Phase 6:** Sliding Window + Unreliable channel

7.2 Verification Methods

File Integrity:

```
diff test.txt received.txt
echo $?  # Should output 0 if files are identical
```

Log File Analysis:

- Check sequence numbers are sequential
- Verify retransmissions occur after timeouts/3 dup ACKs
- Confirm statistics match log entries

Example Test:

```
# Create test file
python3 -c "print('A' * 3500)" > test.txt

# Start receiver
python3 receiver.py 56007 59606 received.txt 3000 &

# Start sender
python3 sender.py 59606 56007 test.txt 3000 100 0.1 0.05 0.1 0.05

# Verify
diff test.txt received.txt
```

7.3 Edge Cases Tested

- **Sequence number wraparound:** File size > 32768 bytes
- **Window full:** max_win < file size
- **High loss rate:** flp/rlp = 0.3
- **High corruption rate:** fcp/rcp = 0.3
- **Combined loss and corruption:** All probabilities > 0
- **Minimum window:** max_win = 1000 (Stop-and-Wait)
- **Large files:** Files > 100MB to verify incremental reading
- **Memory efficiency:** Verified constant memory usage regardless of file size

---

8. Limitations and Known Issues

8.1 Working Features

✅ Two-way connection setup (SYN/ACK)
✅ Sliding window protocol with adjustable window size
✅ Timeout retransmission with single timer
✅ Fast retransmission on 3 duplicate ACKs
✅ Cumulative acknowledgment
✅ Out-of-order data buffering
✅ Error detection via 16-bit checksum
✅ PLC module for loss and corruption emulation
✅ Complete logging with real-time statistics

✅ One-way connection teardown (FIN/ACK)
✅ TIME_WAIT state (2 seconds)
✅ Sequence number wraparound handling ✅ Incremental reading file

8.2 Not Implemented (As Per Assignment Spec)

- Flow control
- Congestion control
- Timeout estimation (fixed RTO provided as argument)
- Timeout interval doubling
- Delayed ACKs (immediate ACKs used instead)

8.3 Known Limitations

Scalability:

- Single timer may not scale to very high-speed networks
- Thread creation overhead for each connection
- Mitigation: Sufficient for assignment test cases

Error Handling:

- Limited handling of unexpected protocol violations
- Program terminates on critical errors
- Mitigation: Assumption of correct operation as per spec

8.4 Program Works Correctly Under These Conditions

- File size: Any size
- Window size: 1000 bytes to 32768 bytes (MSS multiples)
- Loss probability: 0% to 30% (higher rates may cause excessive delay)
- Corruption probability: 0% to 30%
- RTO: 50ms to 1000ms
- Platform: Python 3.11 on CSE Linux (VLAB)

9. Code References and Acknowledgments

9.1 External Code References

All code is original implementation based on:

- **Textbook:** Computer Networking: A Top-Down Approach (8th Edition) by Kurose and Ross, Section 3.5 (TCP)
- **Course Materials:** COMP3331 Week 4/5 lecture notes on reliable data transfer
- **Python Documentation:** Official Python 3 documentation for:
  - `socket` module
  - `struct` module
  - `threading` module

9.2 Design Inspiration

The following design patterns were inspired by standard practice:

- **Single Timer:** TCP timeout mechanism (RFC 793)
- **Fast Retransmission:** TCP fast retransmit algorithm
- **Sliding Window:** TCP sliding window protocol
- **Checksum:** TCP/UDP checksum algorithm

No code was copied from external sources. All implementations are written from scratch based on understanding of the protocols.

10. Conclusion

This project successfully implements a complete UDP-based reliable transport protocol (URP) with the following key achievements:

1. **Comprehensive Protocol Features:** Connection setup/teardown, sliding window, timeout retransmission, fast retransmission, and error detection
2. **Multi-threaded Architecture:** Efficient concurrent handling of send, receive, and timeout events
3. **Modular Design:** Independent timer, PLC, and logger modules for better code organization
4. **Robust Testing:** Progressive testing strategy ensures correctness under various network conditions
5. **Complete Logging:** Detailed logs facilitate debugging and performance analysis

The implementation demonstrates a deep understanding of reliable transport protocols and provides hands-on experience with the building blocks used in real-world protocols like TCP and QUIC.

Learning Outcomes

Through this assignment, I gained practical experience in:

- **Transport Layer Protocols:** Understanding reliability mechanisms
- **Socket Programming:** UDP sockets, concurrent I/O
- **Concurrent Programming:** Multi-threading, synchronization, deadlock prevention
- **Network Simulation:** Emulating packet loss and corruption
- **Protocol Testing:** Systematic validation of complex systems

Appendix: Statistics Example

For a 3500-byte file transfer with max_win=3000, rto=100ms, flp=0.1, rlp=0.05, fcp=0.1, rcp=0.05:

Sender Log Statistics:

```
Original data sent:             3500
Total data sent:                5500
Original segments sent:         6
Total segments sent:            11
Timeout retransmissions:        5
Fast retransmissions:           0
Duplicate acks received:        1
Corrupted acks discarded:       1
PLC forward segments dropped:   1
PLC forward segments corrupted: 2
PLC reverse segments dropped:   2
PLC reverse segments corrupted: 1
```

Receiver Log Statistics:

```
Original data received:         3500
Total data received:            3500
Original segments received:     6
Total segments received:        10
Corrupted segments discarded:   2
Duplicate segments received:    2
Total acks sent:                8
Duplicate acks sent:            4
```

These statistics demonstrate correct protocol operation with packet loss and corruption.