# Computer Networks Major Task

Weight: 25% (course project) | Group size: 4–6 students | Submission: LMS (one submission per group)

## 1. Learning Outcomes

By completing this project, students will be able to:

- Design a compact application-layer protocol satisfying concrete requirements.
- Implement client-server (and where applicable, peer-to-peer) applications using sockets.
- Instrument code and run controlled experiments (delay/loss/jitter) and analyze results.
- Write an RFC-like protocol specification and a reproducible technical report.
- Work effectively in teams and communicate technical results professionally.

## 2. High-level Expectations & Constraints

1. Primary transport must be UDP for the custom protocol (TCP may be used only for administrative control channels if justified).
2. Your code must be cross-platform: it should run on Linux and Windows with minimal changes (Python 3 recommended).
3. Provide scripts to run the experiments reproducibly; use deterministic seeds when randomization is involved.
4. Log all sent/received packets with timestamps and relevant metadata; include pcap traces for at least two runs per test scenario.
5. Repeat each measurement at least 5 times; report median and range (min/median/max) and explain observed variance.

## 3. Application-Layer Protocols (Each group must choose one project):

### Project 1: IoT Telemetry Protocol (Sensor Reporting)

In this project, you will **design and implement a custom lightweight telemetry protocol** for constrained IoT sensors that periodically send small readings (e.g., temperature, humidity, or voltage) to a central collector. Your goal is to design a compact, efficient, and robust application-layer protocol operating **over UDP**, and to evaluate its performance under varying network conditions experimentally.

**1. Functional Requirements**

**1.1 Core Design**

Your protocol must:

- Use **UDP** as its transport layer.
- Support at least two message types:

    - DATA: carries one or more sensor readings.
    - HEARTBEAT: sent periodically when no new data is available, to indicate device liveness.
- Not use per-packet retransmission (loss-tolerant by design).
- Use a **compact binary header** (≤ 12 bytes recommended).
- Support an **optional batching mode**, where multiple readings (up to N) are grouped into a single packet.
- Include in each message:
    - **Device ID**
    - **Sequence number**
    - **Timestamp**
    - **Message type**
    - **Optional fields** (chosen by your team, e.g., flags, checksum, etc.)

Each group defines its own **protocol name** (e.g., *TinyTelemetry v1*), version, and header format. The header layout and encoding must be fully documented in your Mini-RFC.

### 1.2 Server Responsibilities

Your collector (server) must:

- Maintain per-device state (e.g., last sequence number, timestamps).
- Perform **duplicate suppression** (ignore repeated sequence numbers).
- Detect and **report sequence gaps** when packets are lost.
- Reorder delayed packets by timestamp for analysis.
- (For measurement purposes) Log all received data to a CSV file with fields: device_id, seq, timestamp, arrival_time, duplicate_flag, gap_flag

---

## 2. Operational Constraints (Testable)

| Parameter | Description | Values / Limits |
|---|---|---|
| payload_limit_bytes | Maximum UDP application payload | 200 bytes |
| reporting_intervals | Configurable periodicity of sensor reporting | 1s, 5s, 30s (test all) |
| loss_tolerance | Must tolerate 5% random loss (no retransmission) | Detect but not recover |
| batching | Optional; group up to N readings per packet | N chosen by your team and justified experimentally |

---

## 3. Test Scenarios & Acceptance Criteria

All tests use Linux **netem** for network impairment simulation. Each team must provide scripts to automate these tests.

| Scenario | netem Command | Acceptance Criteria |
|---|---|---|
| **Baseline (no impairment)** | none | ≥99% of reports received (1s interval, 60s test); sequence numbers in order. |
| **Loss 5%** | sudo tc qdisc add dev <IF> root netem loss 5% | Server detects sequence gaps; duplicate suppression works; duplicate rate ≤1%. |
| **Delay + Jitter (100ms ±10ms)** | sudo tc qdisc add dev <IF> root netem delay 100ms 10ms | Server correctly reorders by timestamp; no buffer overrun or crash. |

---

**4. Metrics to Collect**

Each test run should produce the following metrics:

| Metric | Description |
|---|---|
| bytes_per_report | Average total bytes (payload + header) per reading |
| packets_received | Count of successfully received packets |
| duplicate_rate | Fraction of duplicate messages detected |
| sequence_gap_count | Number of missing sequences detected |
| cpu_ms_per_report | CPU time per reading processed |

---

**5. Deliverables**

Each team must submit the following items:

| Deliverable | Description |
|---|---|
| **Mini-RFC (≤3 pages)** | Formal description of your protocol, including: header table, field sizes, encoding format, message types, batching design, and rationale. |
| **Implementation** | Working client ("sensor") and server ("collector") prototypes in C, C++, or Python using UDP sockets. |
| **Testing Scripts** | Automated scripts (e.g., Bash/Python) that run all test scenarios using netem and produce .pcap + .csv outputs. |
| **Results & Plots** | Graphs showing:<br>- bytes_per_report vs reporting_interval<br>- duplicate_rate vs loss |
| **README** | Build instructions, usage examples, and a short explanation of your batching decision and field-packing strategy. |

## Project 2: Multiplayer Game State Synchronization

In this project, you will **design and implement a custom network protocol** to synchronize player positions and ephemeral game events in a simplified multiplayer environment. Your protocol must maintain **low-latency synchronization** between clients and a central server while tolerating moderate network loss and delay. The focus is on **network behavior**, not on building a full game — a minimal 2D environment suffices.

---

**1. Functional Requirements**

**1.1 Core Design**

Your protocol must:

- Use **UDP** as the transport layer.
- The **server broadcasts state snapshots** (position and event updates) to all clients at a configurable frequency (e.g., 10–60 Hz).
- Each update must include:
  - **Sequence number**
  - **Snapshot ID**
  - **Server timestamp**
- Clients must **discard outdated updates** (those with older snapshot IDs).
- Implement one of the following reliability enhancement strategies:
  1. **Redundant updates** (e.g., include last *K* updates per packet),
  2. **Delta encoding** (send only changes since last acknowledged snapshot),
  3. **Selective reliability** (e.g., NACKs or selective ACKs for critical events).
- The system must handle **at least 4 concurrent clients** on the lab testbed.

Each team must define its own **protocol name, version, and header structure** and document them in a **Mini-RFC** (see Deliverables).

---

### 1.2 Protocol Structure

Each message (snapshot or event) must contain a **binary header**. The following fields are required (you may adjust sizes, but must justify them):

| Field | Description |
|---|---|
| protocol_id | 4-byte ASCII identifier unique to your protocol |
| version | 1 byte |
| msg_type | 1 byte (e.g., SNAPSHOT, EVENT, ACK) |
| snapshot_id | 4 bytes |
| seq_num | 4 bytes |
| server_timestamp | 8 bytes (ms since epoch or monotonic clock) |
| payload_len | 2 bytes |
| (Optional) checksum | 4 bytes (CRC32 recommended) |

Each team must ensure the total packet (header + payload) ≤ **1200 bytes**.

---

### 1.3 Client Responsibilities

Clients must:

- Discard outdated or duplicate snapshots.
- Apply updates in order by snapshot_id.
- Maintain and visualize the current positions of all players.
- Implement a **smoothing** technique to reduce perceived jitter. (method to be determined and documented in the mini-RFC)
- Handle **critical events** (e.g., item pickups, shooting) with selective reliability or redundancy.

---

### 1.4 Server Responsibilities

The server must:

- Broadcast periodic snapshots to all connected clients.
- Maintain per-client state (sequence tracking, last acknowledged snapshot).
- Support **4 concurrent clients** without exceeding CPU limits.
- Optionally send additional redundant or delta updates to improve reliability.
- Log performance metrics (CPU usage, update frequency, packet counts).

---

## 2. Operational Constraints

| Parameter | Description | Value |
|---|---|---|
| payload_limit_bytes | Maximum UDP payload size | 1200 bytes |
| latency_target_ms | Average end-to-end latency | ≤ 50 ms |
| clients_supported | Concurrent clients | 4 |
| loss_profile | Random loss rate for testing | 2% (LAN-like), 5% (WAN-like) |

---

## 3. Grid Clash Game

You can develop any multiplayer game that uses your protocol. It's shown here a very simple game called "Grid Clash".

### 3.1 Game Overview

Each player sees the same **shared n×n grid** (e.g., 20×20). Cells start in the **"unclaimed"** state. Players compete to **claim cells** by clicking on them as quickly as possible. When a player clicks an unclaimed cell, it becomes **acquired** by that player — turning to a unique color or ID visible to all. The **game ends** when all cells are acquired. The **winner** is the player with the most acquired cells.

This setup tests low-latency synchronization and conflict resolution across clients.

### 3.2 Sample Core Game Mechanics

| Element | Description |
|---|---|
| **Grid** | n×n cells (e.g., 20×20, total 400 cells). Each cell has a global ID (row, col). |
| **States** | Each cell can be: UNCLAIMED, PENDING, or ACQUIRED(player_id). |
| **Player Actions** | Click on a cell → send ACQUIRE_REQUEST(cell_id, timestamp) to the server. |
| **Conflict Handling** | If multiple requests arrive for the same cell, the server resolves based on **earliest timestamp or first received message**, and broadcasts the result. |
| **Game State Update** | The server periodically sends **state snapshots** to all clients (e.g., 20 updates/sec). Clients apply updates to maintain a synchronized view. |
| **End Condition** | All cells acquired → server sends GAME_OVER(winner_id, scoreboard) to all clients. |

## 4. Test Scenarios & Acceptance Criteria

All tests use **Linux netem** to simulate network impairments. Each team must automate tests with scripts (run_all_tests.sh) that execute the scenarios below, collect metrics, and produce CSV + plots.

### 4.1 Test Scenarios

| Scenario | netem Command | Acceptance Criteria |
|---|---|---|
| **Baseline (no impairment)** | None | Server sustains 20 updates/sec per client; avg latency ≤ 50 ms; avg CPU < 60% on grading VM. |
| **Loss 2% (LAN-like)** | sudo tc qdisc add dev <IF> root netem loss 2% | Mean perceived position error ≤ 0.5 units; 95th percentile ≤ 1.5 units; graceful interpolation. |
| **Loss 5% (WAN-like)** | sudo tc qdisc add dev <IF> root netem loss 5% | Critical events reliably delivered (≥99% within 200 ms); system remains stable. |
| **Delay 100ms (WAN delay)** | sudo tc qdisc add dev <IF> root netem delay 100ms | Clients continue functioning; selective reliability or redundancy prevents visible misbehavior. |

---

## 5. Metrics to Collect

Each test must produce a CSV with at least the following columns:

| Metric | Description |
|---|---|
| client_id | Numeric ID |
| snapshot_id | Snapshot identifier |
| seq_num | Packet sequence number |
| server_timestamp_ms | Timestamp at server send |
| recv_time_ms | Timestamp at client receive |
| latency_ms | recv_time - server_timestamp |

| | |
|---|---|
| jitter_ms | Variation in inter-arrival times |
| perceived_position_error | Euclidean distance between server's authoritative position and client's displayed position |
| cpu_percent | Server CPU utilization |
| bandwidth_per_client_kbps | Average measured bandwidth per client |

Students must report **mean**, **median**, and **95th percentile** values for latency, jitter, and error.

---

**6. Measuring "Perceived Position Error"**

To standardize evaluation:

1. The **server** logs authoritative positions with timestamps.
2. Each **client** logs displayed/interpolated positions with timestamps.
3. Post-process both logs to compute positional error at matching timestamps (interpolated if needed).
    - Metric = ||pos_server(t) - pos_client(t)||
4. Report:
    - Mean and 95th percentile errors.
    - Plots showing position error vs loss rate and update rate.

---

**7. Deliverables**

| **Deliverable** | **Description** |
|---|---|
| **Mini-RFC (≤3 pages)** | Defines protocol format, message types, header table, chosen reliability mechanism, and update rate rationale. Must include state diagram and field justification. |
| **Implementation** | Working server and client prototypes (C/C++/Go/Python) that can run 2–4 clients locally for demo. |
| **Testing Scripts** | Automated scripts to run all netem scenarios and collect CSV + PCAP logs. |
| **Results & Plots** | Graphs showing latency, jitter, and perceived error vs update rate and loss rate; bandwidth comparison with baseline. |
| **README** | Build instructions, how to run tests, and explanation of chosen design mechanisms. |

## 4. Cross-platform Tool Recommendations & How to Use Them

While the protocol implementation must be portable, below are recommended tools and brief usage notes for each OS:

### Linux (preferred testbed for grading)

Recommended tools:
- tc qdisc + netem: scriptable and precise network impairment (loss/delay/jitter/duplication/corruption). Use `tc` for all TA tests.
- tcpdump / tshark / Wireshark: packet capture and analysis. Save pcap for reproducibility.

- iperf3: baseline throughput tests when comparing streaming or file transfer.
- psutil (Python): measure CPU, memory usage programmatically.
- strace (advanced): debug system call issues if code behaves differently in Linux VM.
Notes: For reproducibility purposes, provide your netem commands in the README and include scripts that apply and remove the qdisc settings.

### Windows

Recommended tools:
- Clumsy (https://jagt.github.io/clumsy/): GUI tool to add loss, delay, duplication, and throttling for development. It is not as scriptable as tc, so for automated grading rely on Linux.
- Wireshark: packet capture, same as Linux.
- Windows Subsystem for Linux (WSL2): recommended for students who develop on Windows but want to run reproducible Linux tests locally. Use WSL2 to run netem on a Linux VM or use a Linux VM alongside Windows.
Notes: If you develop on Windows, ensure the code runs unchanged in a Linux VM before submission.
Provide clear instructions in your README for running.

## 5. Project Timeline & Phases (detailed)

### 1. Phase 1 — Prototype (Due: Week 7)

Deliverables (upload to LMS):

- Project proposal (max 3 pages) including: assigned scenario, short motivation, and proposed protocol approach.
- Mini-RFC draft (sections 1–3): Introduction, Protocol Architecture, Message Formats (header table + sample message).
- Working prototype demonstrating INIT and DATA exchanges over UDP (server + client).
- README with instructions to run locally.
- Automated script to run the 'Baseline local' test scenario for your assigned project.

Checklist: proposal clarity, feasibility, initial message format correctness, code runs locally, logs present, prototype demonstrates core functionality.

### 2. Phase 2 — Feature Completion & Tests (Due: Week 12)

Deliverables:

- Full implementation of mandatory scenario-specific features and handling of error cases.

- Evidence of running required test scenarios: logs, pcap traces, netem command list, and raw measurement CSVs.
- Updated Mini-RFC with sections 4–7 (procedures, reliability, experimental plan).

Checklist: mandatory features pass acceptance criteria, logs show tests, pcap snippets included.

### 3. Phase 3— Final Report & Presentation (Due: Week 14)

Deliverables:

- Final Mini-RFC (5–8 pages), full code repository with commit history, and reproducible scripts.
- Technical report (6–8 pages) that includes methodology, detailed plots, interpretation, and limitations.
- Presentation slides (8–10) and final demo video (≤10 minutes) uploaded online (Just make sure that its access is set to "anyone with the link can view"). Only submit the video link inside the README.txt file and make sure that it's working.

Checklist: report quality, reproducibility, comparison with baseline, and presentation clarity.

- **Some key points to be covered in video demo:**

- Explain your protocol header fields and why you chose their length.
- How does your protocol handle packet loss and reordering?
- Show an excerpt from your pcap that illustrates retransmission or recovery.
- Explain one experiment you ran and why the results look the way they do.

## 6. Mini-RFC Template:

Students must submit a Mini-RFC following this structure. Use exact headings and include required subsections. Below we give expanded guidance and examples.

### 1. Introduction

Brief description of protocol, use case, and why a new protocol is needed. State assumptions and constraints (e.g., max packet size, allowed loss).

### 2. Protocol Architecture

Describe entities, sequence flows, and provide a finite-state machine (FSM). Include diagrams (png/svg) or ASCII diagrams in the appendix. Example: Client -> Server sessionless telemetry vs sessionful file-transfer.

### 3. Message Formats

Provide a header field table. Example (IoT telemetry header):

Example: IoT Telemetry Header (total 10 bytes):

Field | Size (bits) | Description
---
Version | 4 | Protocol version
MsgType | 4 | Message type (0=INIT,1=DATA,2=ACK,3=ERROR)
DeviceID | 16 | Unique device id
SeqNum | 16 | Sequence number
Timestamp | 32 | Unix epoch seconds (or milliseconds truncated)
Flags | 8 | Flags for batching/priority

Students should provide byte offsets and struct packing format (e.g., struct.pack('!B B H H I B', ...)) for each header. If using variable-length payloads, document length fields and fragmentation rules.

### 4. Communication Procedures

Describe session start, normal data exchange, error recovery, and shutdown. Provide example step-by-step sequences.

### 5. Reliability & Performance Features

Explain retransmission strategy, timeout selection, windowing, or other techniques like FEC. If using timers, show how they were calculated (e.g., RTO = estimate_RTT + 4 * dev_RTT).

### 6. Experimental Evaluation Plan

Specify baselines, metrics, measurement methods, and how to simulate network conditions. Include exact netem commands for Linux and alternatives for other OSes. Provide scripts to automate runs and collect results.

### 7. Example Use Case Walkthrough

Provide an end-to-end trace example (timestamps, messages sent/received) for a single session. Include pcap excerpt (as appendix) and explanation.

### 8. Limitations & Future Work

Honest assessment of weaknesses and potential improvements; essential for high marks.

### 9. References

Reference any RFCs, papers, or libraries used. Include links or RFC numbers where applicable.

## 7. Originality and Academic Integrity

Each team must design an **original protocol**. Submissions will be checked for similarity against public repositories and AI-generated samples.

**Your design must include:**

1. A unique protocol name and version field.

2. At least **two original mechanisms or field definitions**, such as a timestamp encoding, field/data compression, or adaptive update rate.
3. A justification section in your Mini-RFC explaining how your design differs from known protocols (e.g., MQTT-SN).
4. No reuse of external networking/game libraries (e.g., ENet, RakNet, Lidgren). Only standard socket and serialization libraries are allowed.

Submissions that substantially replicate public code or lack experimental justification will not receive credit.

# 8. Appendices

## 1. Appendix A — Example netem commands (Linux)

Replace <IF> with your interface (e.g., eth0 or lo for loopback):
- Add 5% loss: sudo tc qdisc add dev <IF> root netem loss 5%
- Add 100ms delay with 10ms jitter: sudo tc qdisc add dev <IF> root netem delay 100ms 10ms
- Combine rate and delay (tbf + netem):
  sudo tc qdisc add dev <IF> root handle 1: tbf rate 2mbit burst 32k latency 400ms
  sudo tc qdisc add dev <IF> parent 1:1 netem delay 50ms
- Remove qdisc: sudo tc qdisc del dev <IF> root

## 2. Appendix B — Reproducible experiment checklist (recommended)

- Provide a script ./run_experiment.sh that sets up netem (optional) and runs the client/server for a single test.
- Store raw outputs (pcap, CSV) in a directory labeled with timestamp and seed.
- Provide a Jupyter notebook or Python script that loads CSVs and reproduces the plots in the report.
- Document exact commands and environment (OS, Python version, library versions).