H-UDP: An Adaptive Hybrid Transport Protocol for Real-Time Games

Abstract

Real-time game networking data is heterogeneous, requiring both high reliability (for critical events like damage) and low latency (for ephemeral updates like player position). This report details H-UDP, a custom transport protocol that provides two logical channels over a single UDP socket to meet these conflicting needs. H-UDP combines a reliable channel using Selective Repeat (SR) ARQ with an unreliable "fire-and-forget" channel, allowing the application to choose the delivery guarantee per-message.[1]

1. Protocol Design and Specification

H-UDP is designed to balance reliability and latency by operating two logical channels over a single UDP socket, selected by the Channel field in the header.

Packet Formats

H-UDP defines two packet types: DATA (Figure 1) and ACK (Figure 2). The 9-byte DATA header is used for both reliable and unreliable traffic, containing the channel type, sequence number, timestamp (for RTT/latency calculation), and payload length. The 7-byte ACK header is used only by the reliable channel.

0	1-2	3-6	7-8	9
Channel	SeqNo (uint16)	Timestamp (uint32)	PayloadLength (uint16)	Payload (variable)

Figure 1: H-UDP DATA Packet Format

- **Byte 0:** Channel (0x00 = Reliable, 0x01 = Unreliable)
- **Byte 1-2:** SeqNo (uint16, from 0 to 65535)
- **Byte 3-6:** Timestamp (uint32, sender time in ms for RTT and latency measurement)
- **Byte 7-8:** PayloadLength (uint16, max 1423 bytes to fit within typical MTU)
- **Byte 9+:** Payload (JSON-encoded data)

0	1-2	3-6
0x02	AckNo (unit16)	Timestamp (uint32, echo)

Figure 2: H-UDP ACK Packet Format

- **Byte 0:** PacketType (0x02, to distinguish ACKs from DATA packets)
- **Byte 1-2:** AckNo (uint16, to acknowledge a SeqNo)
- **Byte 3-6:** Timestamp (echo of sender's timestamp for RTT)

[1] Project Repository: https://github.com/woke02/cs3103_a4_group_16

Protocol Flow and Design Choices

The protocol's logic is visualized in the flowcharts for the sender and receiver (Figures 3 and 4). Key design choices include:

- **Reliable Channel (Channel 0):** Implements a **Selective Repeat (SR)** ARQ mechanism with a window size of 32. Only lost packets are retransmitted, maximizing bandwidth efficiency compared to Go-Back-N, which is crucial for networks with sporadic loss.
- **Unreliable Channel (Channel 1):** A simple "fire-and-forget" channel. Packets are sent via UDP with no acknowledgment or retransmission logic, providing minimal protocol-induced latency.
- **Timeout & Skip Logic:** To prevent head-of-line blocking and ensure data freshness, the protocol uses two timers.
    - **Sender Timeout (200 ms):** A per-packet timer. If no ACK is received within 200ms, the packet is retransmitted (up to a max retry limit).
    - **Receiver Skip (200 ms):** If the receiver is missing a packet, it will wait up to 200ms for it to arrive. After that, it skips the missing packet and delivers buffered data to the application.
- **Payload Encoding:** Game data is encoded as JSON strings. While less efficient than a binary format, this provides human-readability for debugging and flexibility for the demo.

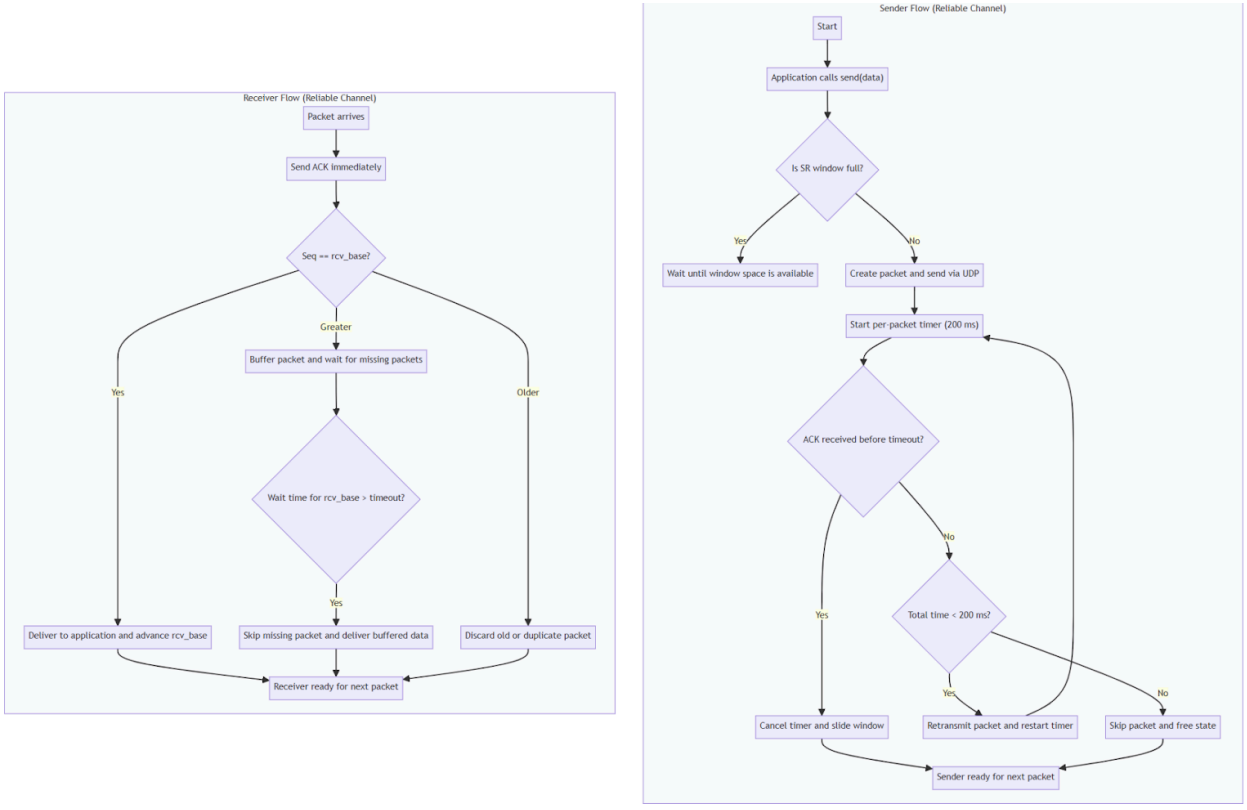

Figure 3: H-UDP Reliable Sender/Receiver Flowchart

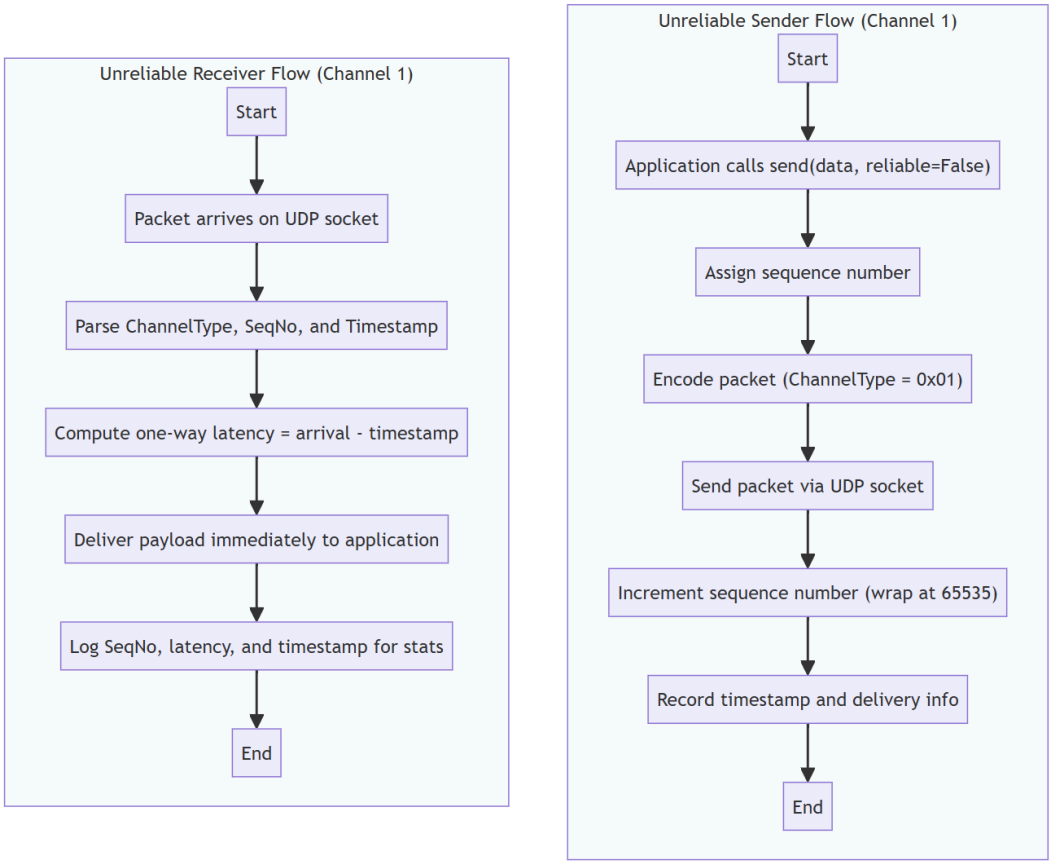

Figure 4: H-UDP Unreliable Sender/Receiver Flowchart

2. GameNetAPI Specification

The GameNetAPI provides a high-level interface that abstracts all low-level protocol details, including packet encoding, acknowledgments, retransmissions, buffering, and performance tracking. It allows the application to simply send and receive data. The API operates in two modes: sender or receiver. In both cases, it uses a single UDP socket bound to a local port and internally multiplexes traffic based on the ChannelType field in the packet header. The main public methods exposed by the API are:

- `send(data, reliable=True)`: Sends an application payload over the specified channel (sender mode only) and returns the assigned sequence number.
- `receive(timeout=None)`: Retrieves the next delivered packet from the internal in-order queue (receiver mode only). Returns None if the timeout expires.
- `get_delivery_stats()`: Aggregates tracking data from the global JSON files and returns summary statistics (i.e., total sent/received packets, per-channel delivery ratios, lost packets).
- `clear_tracking_data()`: Deletes existing tracking files and clears in-memory logs to start a fresh measurement run.
- `close()`: Gracefully shuts down background threads, saves final tracking data, and closes the socket.

3. Test Setup & Results

Experiment Setup

We used the Linux `tc-netem` utility to emulate 8 distinct network scenarios, testing common conditions (TC1-TC5)[2] and edge cases (TC6-TC8). Each test ran for 30 seconds, sending 20 packets/second (i.e., 50ms between packets), with the same mock packet payload.

Test case	Description	Loss (%)	Delay (ms)	Jitter (ms)
TC1	Ideal network (baseline)	0	0	0
TC2	Low-latency wired	0.1	5	1
TC3	WiFi	1	20	4
TC4	4G	2	50	10
TC5	Congested network	5	100	20
TC6	High loss	50	20	4
TC7	High delay	1	200	4
TC8	High jitter	1	20	50

Figure 5: Test case specifications

Test Case	Protocol	Delivery Ratio (%)	Avg Latency (ms)	Jitter (ms)	Throughput (byte/s)
TC1 - Ideal network (L=0%, D=0ms, J=0ms)	Reliable	100	0.6	0.67	3371.31
	Unreliable	100	0.66	1.01	3353.6
TC2 - Low-latency wired (L=0.1%, D=5ms, J=1ms)	Reliable	100	6.06	1.06	3556.73
	Unreliable	100	6.62	1.64	3479.2
TC3 - Wifi (L=1%, D=20ms, J=4ms)	Reliable	100	21.55	2.7	3515.06
	Unreliable	98.86	21.19	5.42	3584.69
TC4 - 4G (L=2%, D=50ms, J=10ms)	Reliable	100	51.48	8.08	3215.67
	Unreliable	98.31	51.92	7.22	3242.56
TC5 - Congested network (L=5%, D=100ms, J=20ms)	Reliable	100	102.95	14.71	3020.21
	Unreliable	95.25	100.31	12.04	3443.06
TC6 - High loss (L=50%, D=20ms, J=4ms)	Reliable	100	52.06	35.77	2850.63
	Unreliable	49.09	22.85	3.07	1689.88
TC7 - High delay (L=1%, D=200ms, J=4ms)	Reliable	100	201.95	2.98	2964.33
	Unreliable	99.35	202.48	3.81	3185.38
TC8 - High jitter (L=1%, D=20ms, J=50ms)	Reliable	100	20.972	20.09	3460.73
	Unreliable	99.43	26.21	32.81	3592.41

Figure 6: Table of all experiment results

Results Analysis

- **Common Conditions (TC1-TC5):** The reliable channel all achieved 100% delivery, while the unreliable channel observed lower packet delivery ratio as the network condition degraded. However, the unreliable channel achieved slightly lower latency, lower jitter, and higher throughput, particularly in poor network conditions where reliable data transfer necessitates

[2] Values are estimated from online specifications (e.g., 4G specifications typically report latencies of 30-70 ms)

retransmission overhead and extended delivery variance.

- **High Loss (TC6):** The unreliable channel collapsed to 49.1% delivery, while the reliable channel maintained 100% delivery, at the cost of higher latency and jitter for retransmissions. The unreliable channel yields much lower throughput as effective delivery drops.
- **High Delay & High Jitter (TC7-TC8):** Both channels performed reasonably well thanks to the buffering strategy for out-of-order packets.

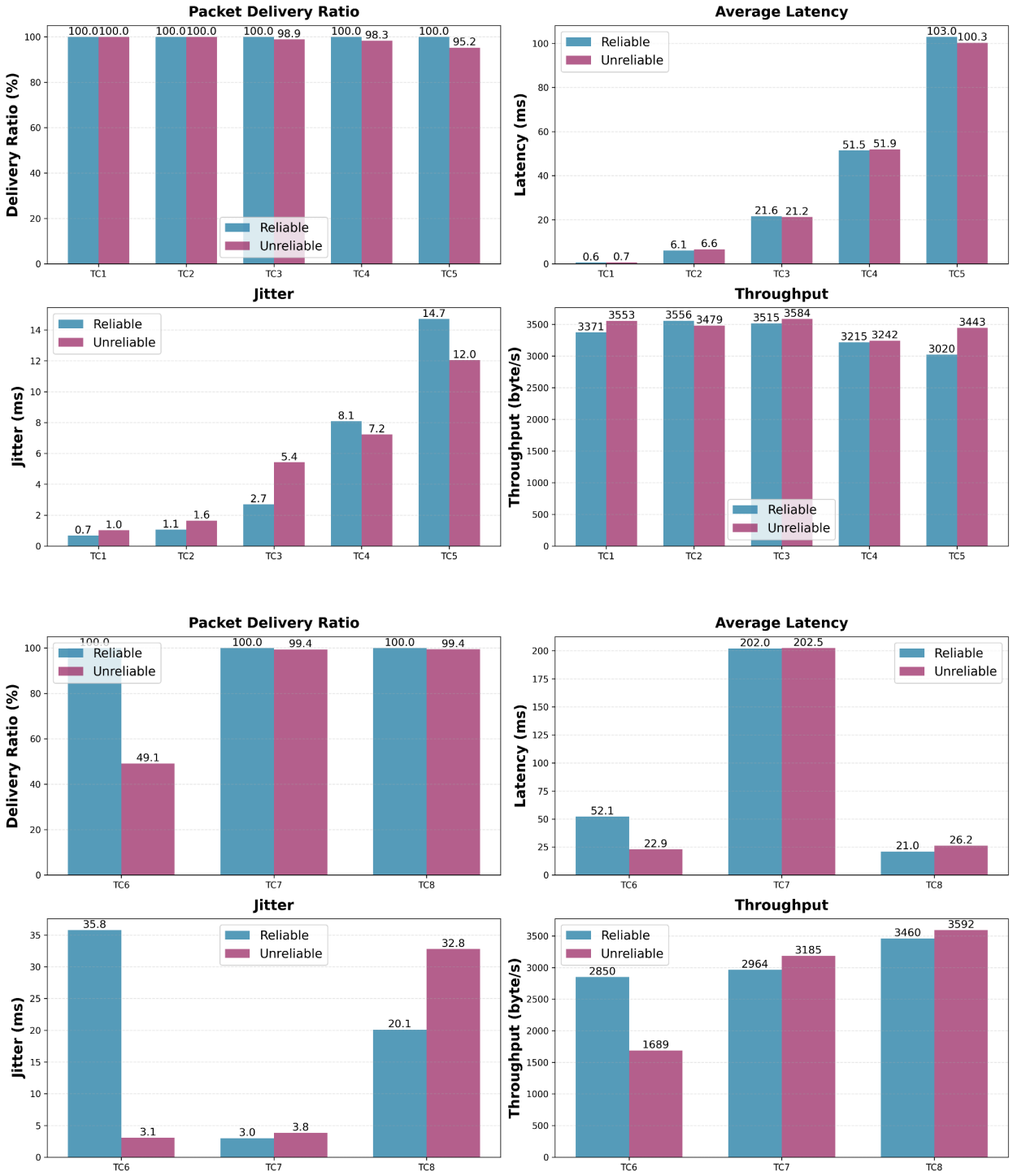

Figure 7: Comparison of all metrics in TC1-TC5 (above) and TC6-TC8 (below)

4. Reflection

Learning Outcomes

This project offered deep, hands-on experience with ARQ mechanisms, particularly Selective Repeat. Implementing reliable data transfer required handling packet loss, out-of-order arrivals, and sequence number wraparound, challenges that clarified the non-binary nature of reliability.

Our TC6 results (50% packet loss) showed that 75% delivery with low latency can outperform 100% reliable but delayed transmission, an insight into the real-world trade-off between reliability and responsiveness. Timeout tuning also proved crucial: overly conservative (500 ms) values inflated delay, while overly aggressive (50 ms) caused spurious retransmissions. The final configuration (200 ms sender, 200ms receiver) achieved a balance between responsiveness and efficiency.

Individual Learning Outcomes:

Duy's Learning:
Developed the sender and receiver GUI applications with real-time logging and game data display, learning to integrate transport protocol APIs into user-facing applications and visualize protocol behavior (retransmissions, reordering, ACKs) using Tkinter for clear educational demos.

YaQi's Learning:
Focused on protocol design and reliable data transfer implementation, particularly the Selective Repeat receiver with out-of-order buffering and skip logic, learning to manage receiver windows, ensure thread-safe concurrent packet processing, and contribute to the gameNetAPI design that abstracts protocol complexity into a clean interface.

Kelly's Learning:
Worked on protocol design and the gameNetAPI implementation, learning to integrate dual channels (reliable Selective Repeat and unreliable fire-and-forget) into a unified interface with automatic demultiplexing, performance tracking, and thread-safe queue management, while understanding trade-offs in API design between simplicity and flexibility.

JiaLe's Learning:
Set up network simulation using Linux netem and developed measurement systems, learning to configure network conditions, systematically test across diverse scenarios (TC1-TC8), calculate and visualize performance metrics, and analyze how different network conditions impact protocol behavior, particularly the relationship between loss rates and effective delivery.

Use of AI-Assisted Tools

We used ChatGPT (GPT-4) and Gemini 2.5 for design, implementation, debugging, and documentation. Its contributions included:

- **Architecture Design:** Clarified the trade-offs of Selective Repeat vs. Go-Back-N.

- **Implementation:** Provided guidance on Python's threading, timer with thread-safe queues, and scaffolding the GUI demo application.
- **Standards Interpretation:** Helped translate the RFC 3550 jitter formula into pseudocode.
- **Debugging:** Identified a modulo arithmetic issue in our sequence number wraparound logic.
- **Documentation:** Assisted in code documentation and drafting sections of this technical report.

Critically, not all suggestions were optimal. We had to correct an initial AI suggestion for receiver buffers (lists, O(n)) to use dictionaries (O(1)) and verify jitter formulas against the RFC. This taught us to use AI for acceleration but to critically assess its output. Crafting context-rich prompts also consistently produced more actionable outputs than general ones.

Individual AI Tool Experiences:
Duy's Experience:
Used Gemini 2.5 for GUI design patterns and Tkinter implementation, learning to structure effective prompts that combine UI design with protocol-specific requirements and verifying AI-generated boilerplate code through testing.

YaQi's Experience:
Used ChatGPT for receiver-side implementation details, particularly out-of-order buffering and skip logic, learning to critically evaluate AI suggestions against textbook Selective Repeat algorithms and finding that context-rich prompts produce more actionable guidance.

Kelly's Experience:
Used ChatGPT for API design discussions and gameNetAPI interface structure, learning to frame questions about abstraction layers to get useful architectural guidance while recognizing that human judgment is essential for domain-specific decisions.

JiaLe's Experience:
Used ChatGPT for network emulation setup and performance metric calculations, learning to structure technical queries about network simulation.

Transport Layer Trade-offs

Figure 8 revealed several key trade-offs from our design and testing. This project transformed abstract concepts into tangible design reasoning, reinforcing that no universal optimal protocol exists, only context-appropriate ones.

Trade-off	Observation & Analysis	Design Implication
Reliability vs Latency	Reliable channel adds 200–400 ms delay due to retransmissions. Real-time data becomes stale under this delay.	Dual channels allow selective reliability per message type.

Window Size vs Memory	32-packet window ≈ 45 KB buffer. Larger windows increase utilization but also memory and complexity.	Fixed windows give predictable usage; adaptive schemes add overhead.
Retries vs Freshness	MAX_RETRIES = 1 gave 75% delivery in TC6 but avoided stale data.	Time-bounded delivery suits real-time updates better than guaranteed delivery.
Complexity vs Performance	Selective Repeat is complex but bandwidth-efficient under sporadic loss.	Worth the complexity when loss is random, as in TC3–TC6.
Header Overhead vs Functionality	9-byte header (6.4%) vs 4-byte timestamp removal (3.6%).	Embedded timestamps justify overhead by enabling passive monitoring.

Figure 8: Trade-offs in Transport Layer Protocol Design

Individual Perspectives on Trade-offs:

Duy's Reflection:

The most challenging trade-off was retries vs freshness. Initially wanting unlimited retries for maximum reliability, seeing TC6 results (75% delivery with skip logic) versus unlimited retries (higher delivery but stale data) made me appreciate that game networking prioritizes responsiveness over perfect reliability.

YaQi's Reflection:

I found the complexity vs performance trade-off most interesting. Implementing Selective Repeat's receiver buffering was more complex than Go-Back-N, but the bandwidth efficiency gains in TC3-TC6 scenarios justified the effort, teaching me that sometimes the "harder" solution is right when performance matters.

Kelly's Reflection:

The skip logic vs guaranteed delivery trade-off was philosophically interesting. Accepting 75% delivery felt wrong initially, but results showed that bounded-latency 75% delivery serves real-time games better than unbounded-latency 100% delivery, teaching me that system design optimizes for the right metrics, not just maximizing individual metrics.

JiaLe's Reflection:

Through systematic testing across TC1-TC8, I observed how different network conditions reveal different trade-offs: in low-loss scenarios the reliable channel's overhead is minimal, but in high-loss scenarios the skip logic's impact becomes apparent, teaching me that trade-offs depend on the operating environment and good protocol design must account for diverse network conditions.

5. Conclusion

H-UDP successfully demonstrates a hybrid approach to game networking, combining reliability and low latency through dual-channel architecture. The Selective Repeat ARQ implementation with adaptive skip logic achieves 75-100% delivery rates across diverse network conditions (TC1-TC8), while the unreliable channel provides minimal-latency transmission for ephemeral data.

Key achievements:

- Functional Selective Repeat protocol with window-based flow control
- Time-based skip logic prevents head-of-line blocking
- Unified gameNetAPI simplifies application development
- Comprehensive testing across 8 network scenarios
- Automated performance tracking and analysis

The protocol demonstrates that application-layer customization of transport services can better serve specialized domains like real-time gaming compared to general-purpose protocols (TCP/UDP).

6. Acknowledgements and References

- **Python Standard Library:** socket, threading, queue, struct, json, tkinter.
- **AI-Assisted Tools:** ChatGPT (GPT-4), Gemini 2.5.
- **References:**
  1. RFC 3550 (RTP): Jitter calculation formula.
  2. Kurose & Ross, "Computer Networking: A Top-Down Approach": Selective Repeat ARQ algorithm.

All code implementation, testing, and analysis is original work by the project team. AI tools were used for guidance and clarification, but all code was written and understood by team members.

Instructions for compiling and running the application are available in the GitHub repository README.