DESIGN AND IMPLEMENTATION OF NUMBER CRUNCHER: AN ONLINE MATH QUIZ GAME USING A CUSTOM SYNCHRONISATION PROTOCOL

Nikeel Ramharakh (2433669)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa.

Abstract: This report details the design and implementation of a custom network protocol for an online multiplayer reaction game, Number Cruncher, developed to address challenges of network synchronisation. The core objective was to ensure fair gameplay by mitigating the impact of variable client latencies. The system features a multi-threaded server and Python clients communicating via a JSON-based protocol. The server implements a synchronisation algorithm that estimates client Round-Trip Times (RTTs) and staggers the dispatch of questions to achieve near-simultaneous arrival at all clients. Server-authoritative timing is used for scoring and measures are included to counter basic connection speed spoofing by clamping answer times. The project involved implementing user authentication, session management and the synchronous quiz game logic. System performance and synchronisation effectiveness were verified through automated tests and manual observation under simulated diverse network conditions. The findings demonstrate a functional synchronisation mechanism that enhances fairness in time-sensitive multiplayer interactions.

Key words: Network Protocol, Synchronisation, Round-Trip Time, Server-Authoritative Timing, Anti-Spoofing, JSON, Python

1. INTRODUCTION

In modern network applications, from online multiplayer games to video conferencing and collaborative document editing, synchronising information among geographically dispersed users presents a fundamental challenge. Varying network latencies, stemming from factors such as physical distance to servers, bandwidth limitations and infrastructure quality, can lead to inconsistent user experiences and create unfair advantages if not properly managed. This project addresses this challenge by designing and implementing a custom network protocol. This protocol aims to ensure that game messages are experienced by clients at approximately the same time, despite their varying network conditions. The project demonstrates this protocol through Number Cruncher, a fun, synchronous multiplayer mathematics quiz game where the server actively manages message dispatch timing and fairly determines player performance by accounting for network latencies. The developed protocol utilises structured command/response pairs to synchronise the delivery of game questions to all players, while also incorporating measures to mitigate issues such as connection speed spoofing. This report is organised as follows: Section 2 reviews existing synchronisation methods in distributed applications. Section 3 outlines the server and client architecture. Section 4 presents experimental results validating the system's synchronisation under simulated conditions. Section 5 provides a critical analysis of the system's successes and limitations, including security considerations and potential improvements. Finally, Section 6 summarises and concludes.

2. LITERATURE REVIEW

2.1 Time Synchronisation Protocols

The Network Time Protocol (NTP) [1] synchronises computer clocks over variable-latency networks, achieving millisecond accuracy by referencing hierarchical time servers traceable to atomic clocks. For higher precision, the IEEE 1588 Precision Time Protocol (PTP) [2] offers sub-microsecond accuracy through hardware timestamping, though its complexity limits gaming applications. Online games typically employ application-layer techniques focusing on game-specific events rather than absolute clock synchronisation [3]. These approaches use timestamped messages and RTT estimates to adjust event scheduling and validate client actions aligning with the game's methodology to prioritise gameplay fairness and consistency.

2.2 Synchronisation Methods in Networked Games

Networked games employ various synchronisation techniques to maintain coherence and fairness despite latency. Among these, server-authoritative timing is foundational [4]. This approach sets the server as the head for events and game states, recording when events occur using its internal clock. Number Cruncher implements this technique for scoring, measuring reaction times from the server's perspective to ensure fair competition regardless of client clock manipulation.

RTT estimation is another vital method. Clients or servers can estimate the latency to another party by measuring the time taken for a PING-PONG message exchange [5]. As demonstrated in this project, RTT estimates enable the server to stagger the transmission of game questions to different clients. By sending data earlier to higher-latency clients and later to lower-latency clients, the objective is for the information to arrive at all clients at approximately the same real-world moment, which is key for fairness in time-sensitive tasks. While not implemented in the current client for Number Cruncher, client buffering and scheduled display is a complementary technique. If a server provides a target display timestamp alongside game data, clients receiving data early could buffer it and use a local timer to render the information at the server-specified target. This can further mitigate jitter for visual synchrony.

The choice of synchronisation methods invariably involves trade-offs between accuracy, complexity, bandwidth and computational overhead, tailored to the specific game's requirements [3]. For this project, the combination of server-authoritative timing and RTT-based staggered dispatch was selected to maximise fairness in question presentation and answer evaluation.

3. SYSTEM DESIGN AND IMPLEMENTATION

3.1 System Overview

The Number Cruncher system features a central Python server managing game logic, user authentication (UserDataStore with salted password hashing) and game sessions (GameSession) [6]. Clients connect to this server, sending commands and receiving game updates. The server is multi-threaded, handling multiple clients concurrently.

Table 1: Primary Responsibilities of Server and Client.

Part	**Responsibilities**
Server	Manages user authentication; creates/manages game sessions; orchestrates game flow; sends synchronised problems; processes answers; calculates scores; broadcasts state updates.
Client	Connects to server; sends user commands (login, session, game actions); receives/displays game information, questions, results; estimates/reports RTT for synchronisation.

3.2 Synchronisation Algorithm for Fair Gameplay

To ensure players perceive new questions appearing simultaneously, a server-controlled synchronisation mechanism is employed. Clients periodically perform RTT estimation via a PING/PONG exchange and report it using an UPDATE_RTT message.

Before dispatching a new question, the server's GameSession Identifies the maximum RTT among all active players in the session. Then calculates a target_display_utc by adding half of this maximum RTT (max one-way latency) to the current server time. This timestamp is the universal target for question appearance and the scoring baseline. Computes an individual dispatch delay for each client. Finally, it uses separate threading. Timer instances to send the SESSION_PROBLEM message (containing the question and target_display_utc) to each client after their delay. This staggering aims for simultaneous network arrival of the problem data.

Upon receiving the SESSION_PROBLEM, the client currently displays the question immediately. Fairness is achieved by the server's synchronised arrival of this message and unbiased scoring: TimeTaken = ServerAnswerReceiptTime - target_display_utc.

3.3 Implemented Features

The game implements the minimum and some advanced features. Its network protocol ensures fair gameplay by measuring client connection speeds and adjusting message dispatch timing. Math questions are delivered with calculated delays, targeting near-simultaneous receipt for all participants despite varying network conditions. This synchronisation adapts to connection fluctuations and includes safeguards against speed spoofing. Advanced implemented features include: secure user authentication; management of multiple concurrent game sessions with a lobby system; host-initiated gameplay requiring at least two players and support for three or more. All participants have real-time visibility into the scores and performance of others throughout the game.

3.4 Code Structure

The Python codebase is organised into distinct server (Server.py) and client (Client.py) components. Server classes are detailed in Table 2.

Table 2: Key Server Classes and Defined Roles.

Class	**Primary Role**
Server	Main class; listens for connections, manages client threads, holds session/client references.
UserDataStore	Handles user registration/login; secure credential JSON storage.
GameSession	Manages individual game session logic: player handling, game progression, synchronisation algorithm, answer check, scoring.

The Client, encapsulated in StandardCLIClient, manages server connection, command sending, message reception (via a dedicated thread), state maintenance, CLI display and RTT estimation. Communication relies on JSON messages with defined "command" fields.

4. EXPERIMENTS AND RESULTS

4.1 System Testing and Verification

The Pytest framework facilitated the creation of automated tests to ensure the validity, stability and correct behaviour of the server and client interactions. These tests covered critical components of the application and their outcomes are summarised in Table 3.

The successful completion of these automated tests, particularly the validation of the RTT-based dispatch delays and the anti-spoofing score clamping mechanism, provided strong confidence in the robustness of the protocol commands, server state management and fairness measures. This formed a reliable baseline for the subsequent manual verification of more nuanced arrival time synchronisation behaviours.

4.1.1 Manual Verification of Synchronisation Mechanisms: A key manual test involved observing synchronised problem dispatch. To achieve this, two client instances ran. One client connected directly

Table 3: Summary of Automated Pytest Results

Test Area	Verified Aspects	Outcome
Server Lifecycle	Startup & Shutdown	Passed
User Authentication	Registration (New/Existing User Handling) & Login	Passed
Session Management	Session Creation, Client Join/Leave, Lobby State (Player Counts, Hosts)	Passed
RTT Protocol	PING-PONG Exchange, Client-side RTT Calculation, Server Reception of UPDATE_RTT	Passed
Anti-Spoofing Mechanism	Score Clamping: Server correctly clamps score to a minimal positive value (0.001s) if a client's answer arrives before the problem start time due to RTT manipulation.	Passed

to the server on the local host (127.0.0.1:4242), exhibiting minimal inherent latency of around one millisecond. The second client connected to the same server via a TCP tunnelling service, known as Zrok [7], which introduced a significant, observable latency, usually in the hundreds of milliseconds. Upon game start, server logs confirmed that distinct RTT delays were calculated and applied before sending the SESSION_PROBLEM message to each client. The message to the low-latency (local) client was sent later than the message to the high-latency (tunnelled) client. Crucially, direct visual observation of the two client terminals showed the math problems appearing nearly simultaneously. Client-side logs, timestamping the reception of the SESSION_PROBLEM message using Python's time.perf_counter(), further confirmed this near-simultaneous arrival, typically within a small margin attributable to residual network jitter and processing overheads. This manual test effectively demonstrated the server's active synchronisation efforts compensating for very different network conditions. The detailed log files, capturing server dispatch times and client reception times, are available for inspection.

The system's winner determination and latency accounting were also scrutinised. The winner is decided based on a player's TimeTaken, calculated as ServerAnswerReceiptTime minus ServerProblemStartTimeUTC. The ServerProblemStartTimeUTC represents the server's calculated target time for when all clients should ideally be able to see and react to the problem, accounting for their estimated RTTs for question delivery. This initial synchronisation of the problem's starting point is a key fairness measure. However, the calculation of TimeTaken, by using the server's answer receipt time, inherently includes the one-way network latency of the answer packet's path from the client to the server. Manual tests confirmed this: a client with a slower true human reaction but benefiting from a very low-latency path for their answer submission could achieve a better TimeTaken than a client with a faster human reaction whose answer packet traversed a higher-latency path. This behaviour is consistent with the implemented scoring logic, which measures the total duration from the synchronised problem display target until the server's receipt of the answer. While this ensures fairness in terms of when players start processing the question, it means the quality of the network path for answer submission still influences the outcome in some cases.

These manual tests, in conjunction with log analysis, validated the core synchronisation strategies implemented, highlighting their effectiveness in creating a fairer gameplay experience across network conditions.

5. CRITICAL ANALYSIS

5.1 Discussion of Implemented Features and Success

The project successfully implemented a multiplayer reaction game underpinned by a custom synchronisation protocol. A working user authentication system provides login and registration using salted password hashing (pbkdf2_hmac), as implemented in the UserDataStore class. Username must be more than three letters and password must be at least six characters. Testing confirmed its functionality in securing user accounts. Session management allows lobbied players to create, list, join and leave game sessions. A host mechanism controls the game start once a minimum number of players (configurable, default to two) have joined. Automated tests and manual verification confirmed smooth session transitions and state updates. The core objective, synchronisation, was achieved by the server calculating individual dispatch delays for game problems based on client RTTs. The server aims for problems to arrive at clients simultaneously (as per GameSession.start_new_round). Manual tests using Zrok to simulate latency differences confirmed near-simultaneous problem display on clients, fulfilling a primary project goal. Three or more users can connect at once. All users receive real-time feedback through lobby updates, round results, and final scores, facilitating an interactive experience.

5.2 Unimplemented Features

While the core objectives were met, certain advanced features or refinements were not implemented. Client display synchronisation to target_display_utc represents one such area. The current client displays the problem immediately upon receiving the SESSION_PROBLEM message. While the server synchronises the arrival of this message, a more precise synchronisation would involve the client buffering the problem and only displaying it when its local clock reaches the target_display_utc provided by the server.

This was not implemented to keep client logic simpler for this project's scope, focusing on server-driven synchronisation. Server RTT measurement was also not implemented. The server currently relies on client-reported RTTs via the UPDATE_RTT command. A more secure system might involve the server measuring RTTs (e.g. by timestamping PINGs and PONGs itself), making it harder for clients to spoof their RTT values for the dispatch delay calculation. This was omitted due to the added complexity on the server to manage ping states for all clients.

Peer-to-peer (P2P) communication or a friend system, an advanced feature from the brief, was not attempted. This was due to the significant architectural additions required, which would shift focus from the core synchronisation protocol to P2P networking challenges like NAT traversal [8], deemed beyond the project's primary scope on server-client synchronisation. Finally, enhanced UI/graphics were not a priority, as per the brief. The CLI implementation is functional but lacks the appeal of a Graphical User Interface (GUI). These decisions were made to prioritise the robustness of the synchronisation protocol and networking logic within the given timeframe.

5.3 *Implemented Anti-Foul Play Mechanisms*

Server-authoritative game logic and scoring form the foundation. The game's integrity relies on its server-authoritative architecture. The server is the sole determinant of the official start time of a question round (problem_start_time in GameSession class), which is the basis for the target_display_utc sent to clients. When a client submits an answer, the server records its receipt time. The TimeTaken is then computed exclusively on the server as the difference between server_answer_receipt_time and server_problem_start_time. This server-centric approach prevents clients from manipulating their answer times, as client-reported timings are not used for scoring. This was verified by ensuring game outcomes were solely based on server calculations.

Synchronised question arrival and anti-spoofing for slower RTT reports are also key. To address latency advantages, the server implements staggered dispatch. Based on client-reported RTTs, the server calculates the maximum one-way latency within the session. It then determines an individualised delay for sending the SESSION_PROBLEM message to each client, aiming for the data packets to arrive at clients' network interfaces at approximately the same real-world time (GameSession.start_new_round). This target instant is communicated as target_display_utc. This mechanism inherently makes it non-advantageous for a user to report a slower connection (higher RTT) than actual to receive the question data packet earlier. If they do, the server will send their packet with less delay. However, their answer's TimeTaken is still judged against the target_display_utc, which was calculated based on their (spoofed) slower RTT. If they answer very quickly using their actual fast connection, their answer might arrive "before" this target_display_utc, leading to clamping. The primary risk of early data packet arrival (not early display, as per current client logic) is potential parsing by a bot. Reporting an extremely fast RTT would cause the server to delay their problem message more, which is counterintuitive for cheating.

Clamping of anomalous answer times provides an anti-spoofing measure against early answers. The server's GameSession.handle_answer logic clamps the calculated TimeTaken to a minimal positive value (0.001s) if it is negative. This directly prevents players from achieving impossible negative scores if their answer, due to RTT misrepresentation or network effects, arrives at the server before the calculated target_display_utc.

5.4 *Consideration of Network Attacks and Errors*

The application is vulnerable to Denial of Service (DoS/DDoS) attacks. Mitigation typically requires infrastructure-level solutions like advanced TCP protection beyond application code [9]. The Python socket server can be overwhelmed by connection floods. Packet manipulation via Man-in-the-Middle (MitM) attacks is a concern because traffic is JSON over plain TCP. This could allow reading questions or injecting false answers. Transport Layer Security (TLS) encryption [10] would mitigate this, but adds complexity (certificate management).

Generic network errors such as packet loss, duplication and reordering are handled by TCP, which Number Cruncher relies upon. Severe network issues manifest as increased latency, which the RTT estimation and synchronisation attempt to account for.

5.5 *Acknowledging Advanced Cheating Vectors*

While the implemented measures address timing manipulations related to network latency, sophisticated cheating methods like automated bots or client binary modification remain theoretical vulnerabilities.

The design of the quiz questions themselves incorporated elements intended to present a greater challenge to simple automated systems. Questions involving visual counting of elements (lemon sweets and secure cookies) or requiring multi-step, non-straightforward mathematical reasoning (the burger server problem) were strategically chosen over boring text-based or direct-lookup arithmetic. Despite these efforts, it is acknowledged that a sufficiently advanced bot could still be developed to parse the question data from the network buffer upon arrival, interpret the visual or complex mathematical components and formulate an answer. Such a bot could then attempt to send its answer precisely as the target_display_utc is reached.

For this project's scope, mitigating highly advanced bots or client-side tampering was considered secondary to establishing a fair synchronisation baseline against network variations and basic RTT manipulation. More invasive anti-cheat measures are common in genres like First-Person Shooters (FPS) [11], but were not deemed necessary for this quiz game format. The current system provides a reasonable defense against direct manipulation of game timing or gaining unfair advantages through typical network latency variations.

5.6 Future Improvements

Several potential enhancements could further improve the system's robustness, fairness, and feature set. Core synchronisation could be refined by implementing client logic to buffer and display problems precisely at the server-specified target_display_utc, ensuring true visual synchrony. Additionally, transitioning to server-measured RTTs, rather than relying on client reports, would bolster resilience against RTT manipulation. To more accurately reflect player reaction time, the server could attempt to compensate for answer path latency by subtracting an estimated one-way delay (RTT/2) from the total TimeTaken.

Beyond core timing, overall security and engagement could be increased. Implementing this would protect against eavesdropping and man-in-the-middle attacks. Anti-bot measures could be strengthened by introducing more complex CAPTCHA-style challenges [12] or statistical analysis of submission patterns. User experience would benefit from a GUI and the addition of features such as private game sessions, a friend system, persistent statistics and leaderboards. These improvements, collectively, would build upon the current foundation to create an even more robust, secure and engaging online multiplayer experience.

6. CONCLUSION

This project successfully designed and implemented Number Cruncher, an online multiplayer reaction game, demonstrating a custom network protocol focused on client synchronisation. The system enables multiple users to participate in a time-sensitive mathematics quiz, with the server actively managing message dispatch to mitigate network latency disparities and ensure a fair gameplay experience. Key achievements include a robust user authentication system, dynamic session management and a synchronisation algorithm that adjusts problem delivery based on estimated client RTTs. Furthermore, mechanisms to counter basic forms of connection speed spoofing, such as clamping scores for answers arriving before the synchronised start time, were effectively implemented and verified within the Number Cruncher game.

Through automated testing and manual verification, including scenarios with simulated network latency, the core synchronisation logic proved effective in achieving near-simultaneous problem presentation and fair scoring. The project highlights the critical role of server-authoritative design in maintaining game integrity. While the primary objectives for Number Cruncher were met, potential future enhancements such as server-measured RTTs and more advanced security measures like TLS were identified. Overall, this work provides a practical implementation of network synchronisation principles within an engaging game context and serves as a solid foundation for further development.

REFERENCES

[1] D. L. Mills. "Internet time synchronization: the network time protocol." *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 2002.

[2] Z. Idrees, J. Granados, Y. Sun, S. Latif, L. Gong, Z. Zou, and L. Zheng. "IEEE 1588 for clock synchronization in industrial IoT and related applications: A review on contributing technologies, protocols and enhancement methodologies." *IEEE access*, vol. 8, pp. 155660–155678, 2020.

[3] M. C. Angelides and H. Agius. *Synchronization in Multiplayer Online Games*, pp. 175–196. 2014.

[4] G. Gambetta. "Client-Side Prediction and Server Reconciliation." https://www.gabrielgambetta.com/client-prediction-server-reconciliation.html, 2017. Accessed: 2025-05-07.

[5] NATS.io. "Ping Pong." https://docs.nats.io/using-nats/developer/connecting/pingpong, 2024. Accessed: 2025-05-07.

[6] GeeksforGeeks. "What is Salted Password Hashing?" https://www.geeksforgeeks.org/what-is-salted-password-hashing/, 2021. Accessed: 2025-05-09.

[7] OpenZiti Team. "zrok: The world's first shareable, secure, tunneling service built on OpenZiti." https://zrok.io/, 2023. Accessed: 2025-05-09.

[8] strongSwan Project. "NAT Traversal – strongSwan Documentation." https://docs.strongswan.org/docs/latest/features/natTraversal.html, 2024. Accessed: 2025-05-10.

[9] Cloudflare, Inc. "DDoS Protection Documentation – Cloudflare Developers." https://developers.cloudflare.com/ddos-protection/, 2024. Accessed: 2025-05-10.

[10] Cloudflare, Inc. "How Does SSL Work? — SSL Handshake TLS." https://www.cloudflare.com/learning/ssl/how-does-ssl-work/, 2024. Accessed: 2025-05-10.

[11] J. Zhang, C. Sun, Y. Gu, Q. Zhang, J. Lin, X. Du, and C. Qian. "Identify As A Human Does: A Pathfinder of Next-Generation Anti-Cheat Framework for First-Person Shooter Games." *arXiv preprint arXiv:2409.14830*, 2024.

[12] B. S. Saini and A. Bala. "A review of bot protection using CAPTCHA for web security." *IOSR Journal of Computer Engineering*, vol. 8, no. 6, pp. 36–42, 2013.

Appendix

A Custom Network Protocol Specification

This appendix outlines the custom network protocol designed for the online multiplayer Number Cruncher Game. It enables client-server communication for authentication, session management, game flow and synchronisation.

A1 Message Structure and Format

Table 4: Message Format Specifications

Field	**Details**
Encoding	UTF-8
Framing	Messages are sent over a TCP stream. Each JSON object is assumed to be a distinct message. Clients must buffer input and parse full JSON objects, accounting for possible fragmentation or concatenation.
Structure	Every message is a JSON object containing at least a `"command"` key. Other fields carry additional data.

Example Client Request (Login):

```
{ "command": "LOGIN", "username": "testuser", "password" "password123" }
```

Example Server Response:

```
{ "command": "LOGIN_RESPONSE", "success": true, "message": "Login successful.", "username":
"testuser" }
```

A2 Commands and Responses

Table 5: Client-to-Server Commands and Key Server Responses.

Client Command Message Structure	**Description**	**Primary Server Response to Requester**
`{"command": "REGISTER", "username": "<uname>", "password": "<passwd>"}`	Client attempts to register a new user account.	`{"command": "REGISTER_RESPONSE", "success": <bool>, "message": "<text>"}` (e.g., "Username already exists.", "Registration successful.")
`{"command": "LOGIN", "username": "<uname>", "password": "<passwd>"}`	Client attempts to log in with existing credentials.	`{"command": "LOGIN_RESPONSE", "success": <bool>, "message": "<text>", "username": "<uname>" (on success)}` (e.g., "Invalid password.", "User already logged in elsewhere.")
`{"command": "PING", "payload_timestamp": <float>}`	Client sends a PING to measure Round-Trip Time (RTT). The payload is a client-generated timestamp.	`{"command": "PONG", "original_payload_timestamp": <float>}` (Server echoes back the client's original timestamp.)
`{"command": "UPDATE_RTT", "rtt_ms": <float>}`	Client informs the server of its newly calculated RTT in milliseconds.	No direct JSON response. Server internally updates the RTT estimate for this client in its `GameSession`.

Continued on next page

Client Command Message Structure	Description	Primary Server Response to Requester
`{"command": "LIST_SESSIONS"}`	Client requests a list of currently active game sessions.	`{"command": "SESSION_LIST", "sessions": [<session_info>] }` (Array of objects, each with `session_id`, `host_username`, `num_players`, `game_in_progress`).
`{"command": "CREATE_SESSION"}`	Client requests to create a new game session. The requester becomes the host.	`{"command": "CREATE_SESSION_SUCCESS", "session_id": "<sid>", "message": "<text>"}` Or `ERROR` if already in a session.
`{"command": "JOIN_SESSION", "session_id": "<sid>"}`	Client requests to join an existing game session identified by `<sid>`.	`{"command": "JOIN_SESSION_SUCCESS", "session_id": "<sid>", "message": "<text>"}` Or `ERROR` (e.g. "Session not found.", "Game actively in progress...").
`{"command": "LEAVE_SESSION"}`	Client requests to leave their current game session.	`{"command": "LEAVE_SESSION_SUCCESS", "message": "<text>"}` Or `ERROR` if not in a session.
`{"command": "START_SESSION_GAME"}`	Session host requests to start the game in the current session.	No direct JSON response to the host. Server validates request (is host? min players?) and then initiates game start sequence, broadcasting `SESSION_GAME_STARTING` to all in session, or sends an `ERROR` message to the host if validation fails.
`{"command": "SESSION_ANSWER", "=": "<text>"}`	Client submits their answer to the current question in the game session.	No direct JSON response. Server processes the answer asynchronously. Round results are broadcast later via `SESSION_ROUND_RESULT`.

A3 Synchronisation Fields and Mechanism

The synchronisation mechanism relies on several key message fields and a server-driven process to ensure fair problem presentation. Clients first estimate their Round-Trip Time (RTT) using a PING/PONG exchange: the client sends a `PING` with a `payload_timestamp` (its current `time.time()`), and the server's `PONG` echoes this timestamp back, allowing the client to calculate the RTT. This RTT (in milliseconds) is then sent to the server via an `UPDATE_RTT` command, where it's stored as the client's `estimated_rtt`.

For problem dispatch, the server's `GameSession` executes the following:

1. It gathers the latest `estimated_rtt` from all active players.
2. It identifies the session's maximum RTT and calculates the maximum one-way latency (max RTT in ms / 2000.0, yielding seconds).
3. A common `problem_start_time` (also sent to clients as `target_display_utc`) is set to `current_server_time + max_one_way_latency`. This `target_display_utc` serves as the server's authoritative reference for scoring.
4. For each player, an individual send delay is computed: `player_delay = max_one_way_latency - (player_rtt_ms / 2000.0)`. This delay is clamped to zero if negative.
5. A `threading.Timer` dispatches the `SESSION_PROBLEM` message to each player after their respective `player_delay`.

This staggered dispatch aims for near-simultaneous arrival of the problem at all clients. Player scores (`time_taken`) are calculated against the server's `problem_start_time`.