IMPACT OF SERVER TICK RATE ON LATENCY COMPENSATION MECHANISMS IN ONLINE FIRST-SHOOTER GAMES

Lappeenranta–Lahti University of Technology LUT

Bachelor’s Programme in Software and Systems Engineering, Bachelor's thesis

2026

Son Kieu

Examiner: Professor Jussi Kasurinen

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Sciences

Software and Systems Engineering

Son Kieu

Impact of Server Tick Rate on Latency Compensation Mechanisms in Online First-Shooter Games

Bachelor's thesis

2026

34 pages, 7 figures, 2 tables

Examiner(s): Professor Jussi Kasurinen

Keywords: Online FPS games, Server tick rate, synchronization, latency compensation, network latency, client-server architecture

This thesis examines the impact of server tick rate on latency compensation mechanisms in online first-person shooter (FPS) games. In multiplayer gaming environments, network latency can significantly affect gameplay quality and competitiveness, leading to issues such as delayed actions and inaccurate hit registration. As competitive online gaming continues to grow, understanding how to mitigate these effects has become increasingly important.

The study focuses on analysing how different server tick rates influence the behaviour of latency compensation techniques, including client-side prediction, interpolation, and server reconciliation. A simplified multiplayer FPS prototype was developed using Unity game engine to simulate various tick rates (30, 64, and 128 Hz). Controlled experiments were conducted under simulated network conditions, including latency and packet loss. Key performance metrics such as synchronization accuracy, hit registration consistency, CPU usage, and bandwidth consumption were measured and analysed.

The result indicates that higher tick rates generally benefit synchronization accuracy and reduce inconsistencies in hit registration. However, these improvements come at the cost of increased computational load and network bandwidth usage. The findings highlight an exchange between performance and resource efficiency, suggesting that optimal tick rate selection depends on system constraints and application requirements. This study provides practical insights for game developers in designing more responsive and efficient multiplayer systems.

ACKNOWLEDGEMENTS

I would like to thank Professor Jussi Kasurinen for supervising this thesis. His feedback helped me stay focused when the implementation got difficult, and his questions pushed me to think more clearly about what I was actually measuring and why it mattered. I am grateful for his patience and guidance throughout the project.

I would also like to thank my family and friends for their support during the final stages of writing.

SYMBOLS AND ABBREVIATIONS

FPS: First-person shooter (FPS)

RTT: Round-trip time

P2P: Peer-to-peer

HSD: Honestly Significant Difference

ANOVA: Analysis of Variance

CPU: Central Processing Unit

CSV: Comma-separated values

API: Application Programming Interface

Table of contents

Abstract

Acknowledgements

Declarations

No table of contents entries found.

DECLARATIONS

Turnitin

The originality of this thesis has been reviewed with the Turnitin similarity checking service.

AI usage

The author of this thesis, Son Kieu, used the following AI-tools during the preparation of the thesis:

1. **ChatGPT (OpenAI)**

a. Purpose of use: Grammar checking, proofreading, and language refinement.

b. Explanation of the use of the tool: The tool was used to review the thesis text for grammatical errors, improve sentence clarity, and suggest better word choices. All substantive content, methodology, analysis, and conclusions were developed independently by the author.

2. **Claude**

a. Purpose of use: Project suggestions, code fixing, and improving code quality.

b. Explanation of the use of the tool: The tool was used to give ideas for the project implementation, fix code, find bugs, and improve code availability and performance.

Responsibility

The author, Son Kieu, takes full responsibility for the content of this thesis and has reviewed and edited the content generated by the possible use of AI tools.

1 Introduction

Online first-person shooter (FPS) games rely heavily on real-time interaction between players and the game server. In such environments, players expect immediate responses from the server to their actions, such as movement and shooting. However, network latency can introduce delays that negatively affect gameplay, resulting in issues such as lag, inconsistent movement, and inaccurate hit registration. These problems become particularly critical in competitive gaming, where even small delays can influence the outcome of a situation or even the match. Therefore, understanding and reducing the effects of latency is essential for improving both player experience and game performance.

The solution for challenges caused by network latency has been developed in multiplayer game systems with various latency compensation techniques. Well-known approaches include client-side prediction, which allows the client to estimate the outcome of actions before receiving confirmation from the server, and interpolation, which smooths the movement of objects between updates. The result produced by client-side prediction later is enhanced with server reconciliation to correct the discrepancy between the earlier predicted state and the server's authoritative state. While these techniques significantly help improve the overall responsiveness and smoothness of gameplay, they do not eliminate inconsistencies caused by network conditions.

One of the most important factors that influences the effectiveness of latency compensation mechanisms is the server tick rate, which determines how frequently the game server updates and processes player actions. Higher tick rates are generally assumed to improve synchronization accuracy and responsiveness. However, increasing the tick rate also leads to higher computational demands and greater network bandwidth usage. Despite its importance, the impact of server tick rate on latency compensation has not been thoroughly examined through controlled experimental studies, particularly under varying network conditions such as latency and packet loss. This creates a need for a more detailed analysis of the trade-offs involved.

This thesis involves the investigation of the impact of server tick rate on latency compensation mechanisms in online FPS games. A simplified multiplayer FPS prototype is

developed using Unity game engine to simulate different server tick rates, including 30, 64, and 128 Hz. Controlled experiments are conducted under simulated network conditions, such as varying levels of latency and packet loss. The study evaluates key performance metrics, including synchronization accuracy, hit registration consistency, and system resource usage. The results aim to provide insights into the trade-offs between performance and resource efficiency in multiplayer game systems.

2 Background and Related Work

This chapter presents the theoretical foundation and related work relevant to the research topic. It covers the architecture of multiplayer game networking, the effects of network latency on gameplay, the key latency compensation mechanisms employed in online FPS games, and the role of server tick rate in determining system performance and synchronization accuracy.

2.1 Multiplayer Game Networking Architecture

Online multiplayer games rely on a client-server architecture in which a central authoritative server maintains the definitive game state and distributes updates to all connected clients. In this model, each player's device acts as a client that sends input commands to the server and receives state updates in return. The server processes all inputs, resolves conflicts, and broadcasts the updated world state to every participant. This design prevents cheating and ensures consistency, as no single client can unilaterally alter the shared game state (Bernier, 2001).

> An alternative networking model is the peer-to-peer (P2P) architecture, where clients communicate directly with each other without a dedicated server. While P2P reduces infrastructure costs and can lower latency in ideal conditions, it suffers from synchronization challenges and is vulnerable to cheating, making it unsuitable for competitive FPS games. As a result, the client-server model has become the standard approach in the industry. Notable implementations include Valve's Source engine, which popularised many of the latency compensation techniques still in use today (Bernier, 2001).

The frequency at which the server processes and transmits state updates is governed by the server tick rate. Each tick represents one complete update cycle: the server reads incoming client inputs, simulates the game world, and sends the resulting state to all clients. The interval between ticks therefore defines the temporal resolution of the simulation, directly affecting how accurately the server can represent fast-moving events such as projectile travel and player collisions (Claypool & Claypool, 2006).

2.2 Network Latency in Online Games

Network latency, commonly referred to as ping, is the round-trip time (RTT) for a data packet to travel from a client to the server and back. In online FPS games, even modest levels of latency can significantly degrade the player experience. Claypool and Claypool (2006) found that player performance declines noticeably when latency exceeds approximately 100 ms, and that above 500 ms gameplay becomes effectively unplayable for most users. The effects are particularly acute in FPS games because accurate hit registration and precise movement timing are essential to competitive play.

Beyond simple round-trip time, other network impairments further complicate real-time game communication. Jitter refers to the variability in packet arrival times, causing uneven update delivery that makes interpolation more challenging. Packet loss forces the client to either extrapolate the game state based on last-known data or wait for retransmission, both of which can cause visible artefacts. Armitage et al. (2006) note that even a small packet loss rate of one to two percent can noticeably degrade the experience in fast-paced games, since each lost update represents a missing snapshot of the game world.

2.3 Latency Compensation Mechanisms

Latency compensation encompasses a set of techniques designed to mitigate the perceptual and functional effects of network delay in real-time multiplayer games. Rather than eliminating latency, these mechanisms allow the game to behave as if latency were lower by predicting future states, smoothing out visual artefacts, and correcting discrepancies after the fact. The three most widely used techniques in FPS games are client-side prediction, interpolation, and server reconciliation (Bernier, 2001).

2.3.1 Client-Side Prediction

Client-side prediction is the most fundamental latency compensation technique. Instead of waiting for the server's acknowledgement before updating the local game state, the client immediately applies the player's input and simulates the expected outcome locally. This makes the game feel instantaneous to the controlling player, as movement and actions appear

responsive without any perceptible delay. The predicted state is later compared against the authoritative server state, and any discrepancies are corrected through a process known as server reconciliation (Bernier, 2001).

2.3.2 Interpolation

Interpolation addresses the visual discontinuity that arises when a client receives discrete position updates for remote players. Rather than snapping entities directly to each newly received position, the client smoothly transitions them between known positions using linear or Hermite interpolation over a configurable buffer period. This results in fluid movement on the observer's screen even when updates arrive infrequently or at irregular intervals. The trade-off is a small intentional delay, typically equal to two or three tick intervals, between the actual server state and what the observer sees (Armitage et al., 2006).

2.3.3 Server Reconciliation

Server reconciliation is the process by which the client resolves differences between its locally predicted state and the authoritative state received from the server. When a server update arrives, the client compares the confirmed server state against the locally predicted state at the same point in time. If a discrepancy exists, the client rewinds to the confirmed state and re-simulates all subsequent inputs that have not yet been acknowledged. Well-implemented reconciliation is imperceptible to the player, as corrections are blended smoothly or applied instantly when the error is below a configurable threshold (Bernier, 2001).

2.4 Server Tick Rate in FPS Games

The server tick rate defines how many times per second the server processes game logic and sends state updates to clients. At a tick rate of 64 Hz, for example, the server sends an update every 15.6 ms, while at 128 Hz this interval halves to 7.8 ms. Higher tick rates provide finer temporal resolution, meaning that fast events such as rapid shots or quick directional changes are captured more precisely. This directly influences the accuracy of latency compensation

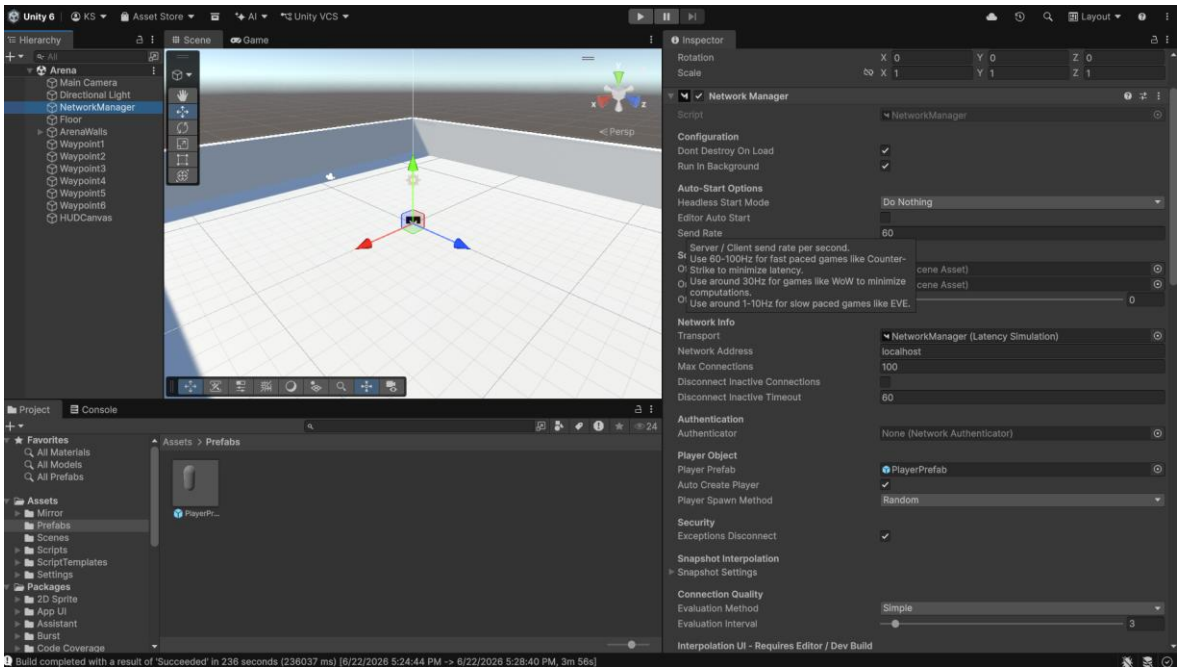
mechanisms, as smaller inter-tick intervals reduce the window within which errors can accumulate (Bernier, 2001).

Popular online FPS titles have adopted a range of tick rates. Counter-Strike: Global Offensive and its successor Counter-Strike 2 offer 64 Hz on standard servers and 128 Hz on premium configurations. Valorant operates at 128 Hz by default, while Battlefield series titles have historically used lower tick rates around 30 Hz due to the scale of their maps and player counts. The relationship between tick rate and perceived gameplay quality has been a subject of ongoing discussion in both competitive gaming communities and academic research, though rigorous empirical comparisons under controlled conditions remain limited (Claypool & Claypool, 2006).

3 Methodology and Implementation

This chapter describes the methodology and implementation plan for the research prototype. It covers the design of a minimal multiplayer first-person shooter built in Unity, the networking architecture and tick rate control mechanisms using the Mirror Networking library, the implementation of client-side prediction, interpolation, and server reconciliation, and finally the experimental conditions and metrics used to evaluate the effects of tick rate on synchronization and performance.

3.1 Prototype Design and Architecture



The prototype is designed as a minimal but functional multiplayer FPS environment. It consists of a small enclosed arena map with two to four player-controlled characters capable of movement and shooting a hitscan weapon. Keeping the scene minimal ensures that observed performance differences are attributable to tick rate changes rather than scene complexity. The game will be developed in Unity 2022 LTS using a standard client-server topology, with one instance acting as the dedicated server and the remaining instances acting as clients connecting via localhost during testing.

To evaluate the impact of server tick rate on latency compensation, four performance metrics were defined and logged automatically during each experimental run. All metrics were sampled at the server tick frequency and exported to CSV for statistical analysis. The following subsections define each metric, its unit of measurement, and how it was computed within the prototype.

3.1.1 Networking Architecture and Tick Rate Control

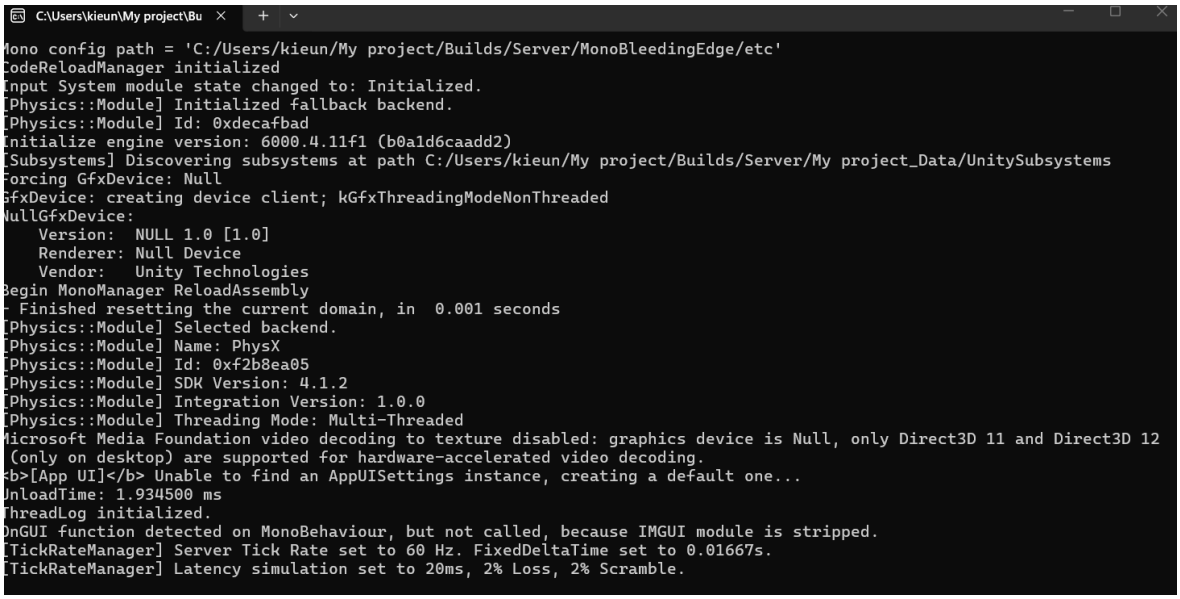
Mirror Networking is an open-source Unity networking library that provides direct control over the server update rate through its NetworkManager.serverTickRate property. This property determines how many times per second the server processes incoming messages, simulates physics, and dispatches state updates to clients. Three configurations will be tested: 30 Hz, 64 Hz, and 128 Hz, matching the tick rates used by major commercial FPS titles. At each configuration the server will run a fixed-timestep loop tied to the tick interval, ensuring that all game logic executes at the correct frequency without floating-point drift. Latency will be simulated by injecting a configurable artificial delay into the transport layer using Mirror's built-in LatencySimulator component, which allows precise control of one-way delay and packet loss percentage without requiring external network emulation tools. Test conditions will cover three latency levels (20 ms, 80 ms, and 150 ms one-way delay) crossed with the three tick rates, giving nine distinct experimental configurations in total.

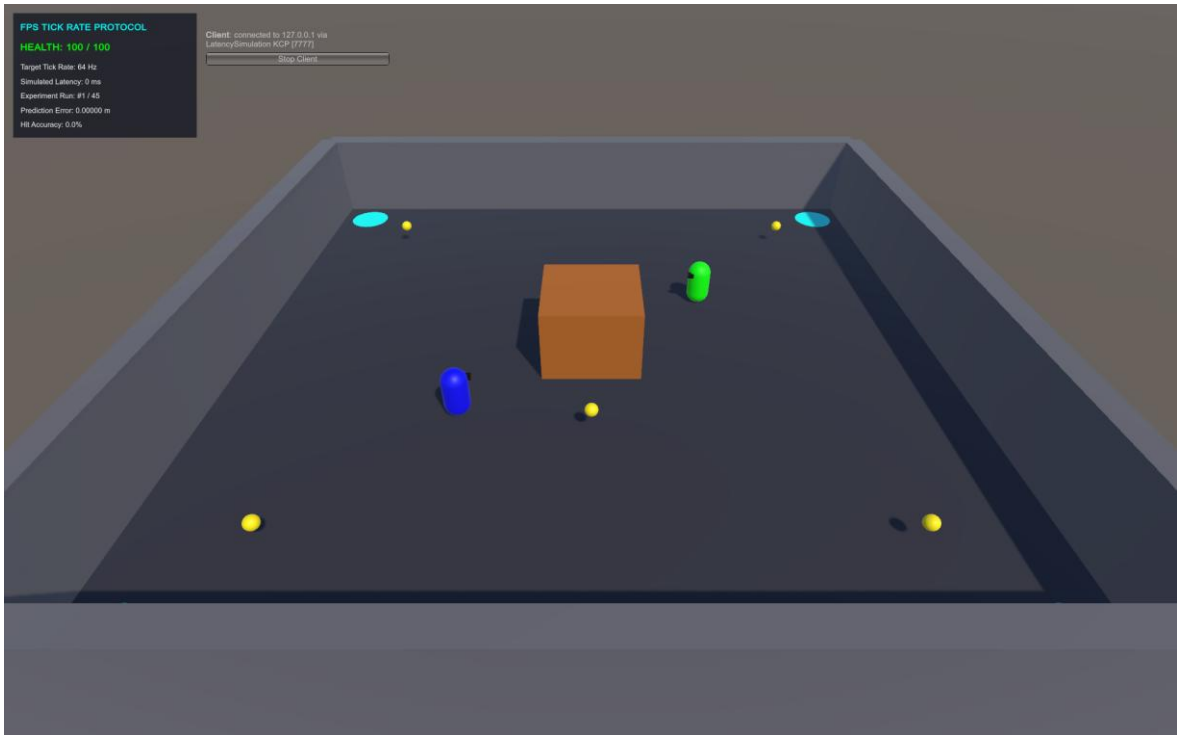
3.1.2 Latency Compensation Implementation

Client-side prediction will be implemented by having each client immediately apply its own input to a local copy of the player state at the moment the input is generated, without waiting for server confirmation. Each input is assigned a sequential tick number and stored in a circular buffer. When a server state update arrives carrying a confirmed tick number, the client locates the matching entry in its buffer, compares the confirmed position against the locally predicted position, and discards all buffer entries older than the confirmed tick. If the positional error exceeds a small threshold (0.01 Unity units), the client snaps to the server position and re-simulates all buffered inputs forward from that point.

Remote player positions will be smoothed using linear interpolation between the two most recently received server state snapshots. The client maintains a small snapshot buffer holding the last several received states with their associated server timestamps. During rendering, the client interpolates between the snapshot timestamped at (currentTime - interpolationDelay) and the next snapshot, where interpolationDelay is set to one tick interval by default. This produces visually smooth movement for remote entities while keeping the displayed state close to the true server state. Packet loss handling will extrapolate the last known velocity if a snapshot is missing.

3.2 Experimental Setup and Metrics





Experiments were conducted on a single development machine running one dedicated server instance and one client connected via localhost. Three AI-controlled bot opponents were spawned to provide consistent combat targets. The automated experiment runner (ExperimentRunner) cycled through all nine combinations of tick rate (30, 64, and 128 Hz) and simulated one-way latency (20, 80, and 150 ms). Packet loss was paired with latency level at 0%, 2%, and 5% respectively, using Mirror's LatencySimulator component. Each configuration was repeated five times, yielding 45 runs in total. Each run lasted 10 seconds, followed by a 2-second cooldown during which match state (player health, weapon statistics, and spawn positions) was reset. An 8-second initialization delay preceded the first run to allow all network connections to stabilise.

3.3 Performance Metrics

Four metrics were selected to evaluate how server tick rate and simulated latency affect latency compensation. Each metric was recorded once per simulation tick during every experimental run and written to a structured CSV log. Together, they capture

synchronization quality, gameplay fairness, and system resource cost — the three dimensions most relevant to tick rate selection in online FPS games.

3.3.1 State Desynchronization Error

State desynchronization error measures the effectiveness of client-side prediction and server reconciliation. It is defined as the Euclidean distance (in metres) between the client-predicted player position and the server-authoritative position at the same simulation tick:

$$e_{\text{pred}} = \| \mathbf{p}_{\text{predicted}} - \mathbf{p}_{\text{server}} \|$$

Whenever the client received a confirmed server state via TargetRpc, the PredictionController compared the server position against the corresponding entry in the local circular state buffer. The resulting distance was stored as LastPredictionError and passed to the telemetry logger on the next tick. A reconciliation threshold of 0.01 Unity units was configured; corrections above this threshold triggered a rewind-and-replay of unacknowledged inputs, but the logged error reflects the raw positional difference regardless of whether a visible correction occurred.

Lower values indicate tighter synchronization between client prediction and server authority. This metric directly reflects how well the latency compensation pipeline maintains positional consistency under varying tick rates and network delays.

3.3.2 Hit Registration Accuracy

Hit registration accuracy measures the consistency between client-predicted and server-confirmed combat outcomes. The prototype used a hitscan weapon: the client sent a fire command to the server carrying the ray origin, direction, and predicted tick number; the server performed an authoritative raycast and determined whether a target was hit.

A shot was counted as a confirmed hit when the server registered a hit on the same tick or the immediately following tick relative to the client's predicted tick (tickDiff ≤ 1). Shots

confirmed after a longer delay were classified as late hits and excluded from the accuracy numerator. Hit registration accuracy was then computed as:

$$A_{\text{hit}} = \frac{N_{\text{confirmed}}}{N_{\text{total}}}$$

where $N_{\text{confirmed}}$ is the number of within-one-tick hits and N_{total} is the total number of shots fired. During each tick, the mean accuracy across all active HitscanWeapon instances was recorded. Values range from 0.0 (no confirmed hits) to 1.0 (all shots confirmed within one tick).

This metric captures a practical fairness concern in competitive FPS games: whether what the player sees on screen matches what the server validates, especially under latency.

3.3.3 CPU Frametime

CPU frametime measures the computational cost of running the prototype at each tick rate. It was recorded as the average frame processing time in milliseconds between consecutive tick recordings, computed from Time.unscaledDeltaTime over the elapsed interval since the previous tick log entry:

$$t_{\text{cpu}} = \frac{1}{N}\sum_{i=1}^{N} \Delta t_i \times 1000$$

Because tick rate governs how frequently the server processes game logic and dispatches network updates, higher tick rates were expected to increase per-frame simulation and networking overhead. This metric indicates whether a given tick rate remains within acceptable CPU bounds on the test hardware. All tested configurations remained below a 20 ms frametime threshold.

3.3.4 Bandwidth Consumption

Bandwidth consumption quantifies the network overhead associated with each tick rate. Bytes sent and bytes received were tracked per tick using Mirror's NetworkDiagnostics API, which fires callbacks on every outgoing and incoming network message. At each tick boundary, the accumulated byte counts for that interval were logged and the counters reset.

Analysis focused primarily on **bytes sent**, as server outbound traffic scales with tick rate and player count in production deployments. Bytes received was also recorded for completeness. Results are reported both as raw bytes per tick and as normalised bandwidth per tick window (KB/tick) in Chapter 4.

Unlike the gameplay metrics above, bandwidth showed a linear relationship with tick rate and was largely unaffected by simulated latency, suggesting that tick rate — not player ping — is the primary driver of network load in this prototype.

3.4 Data Collection and Analysis

During each run, telemetry was logged at every server tick to a CSV file. Four primary metrics were recorded: state desynchronization error (Euclidean distance in metres between server-authoritative and client-predicted player positions), hit registration accuracy (ratio of server-confirmed hits to total shots fired), CPU frametime (milliseconds per frame, sampled from Unity's Profiler API), and bandwidth (bytes sent and received per tick, captured via Mirror's network diagnostics). After all 45 runs completed, data were aggregated by tick rate and latency condition. Descriptive statistics (mean and standard deviation) were computed for each of the nine conditions. One-way analysis of variance (ANOVA) was applied to test whether tick rate had a statistically significant effect on each metric, followed by Tukey HSD post-hoc pairwise comparisons. All statistical analysis was performed in Python using SciPy and pandas.

4 Results and Discussion

4.1 Experimental Results

This chapter presents the results of 45 automated experimental runs covering a 3×3 factorial design: three server tick rates (30, 64, and 128 Hz) crossed with three simulated one-way latency levels (20, 80, and 150 ms). Each of the nine configurations was repeated five times for 10 seconds per run. Performance data were logged at every server tick and aggregated using mean and standard deviation. Table 4.1 summarises the key metrics across all conditions; Figures 4.1–4.3 illustrate the main trends.

Table 4.1. Summary statistics (mean ± standard deviation) for all experimental conditions.

Tick Rate(Hz)	Latency	Prediction Error(m)	Hit Accuracy	CPU Frametime (ms)	Bytes Sent
30	20	0.085 ± 0.048	0.882 ± 0.112	16.49 ± 1.99	239 ± 72
30	80	0.339 ± 0.051	0.783 ± 0.137	16.57 ± 2.03	237 ± 69
30	150	0.638 ± 0.050	0.653 ± 0.141	16.42 ± 2.03	243 ± 73
64	20	0.070 ± 0.046	0.923 ± 0.093	17.14 ± 2.02	510 ± 150
64	80	0.272 ± 0.050	0.840 ± 0.128	17.03 ± 2.04	509 ± 152
64	150	0.509 ± 0.050	0.718 ± 0.147	17.11 ± 1.99	514 ± 156
128	20	0.044 ± 0.041	0.971 ± 0.047	18.14 ± 1.98	1,026 ± 311
128	80	0.143 ± 0.049	0.927 ± 0.090	18.17 ± 1.99	1,025 ± 307

128	150	0.271 ± 0.051	0.833 ± 0.128	18.14 ± 2.00	1,021 ± 305

4.1.1 State Desynchronization Error

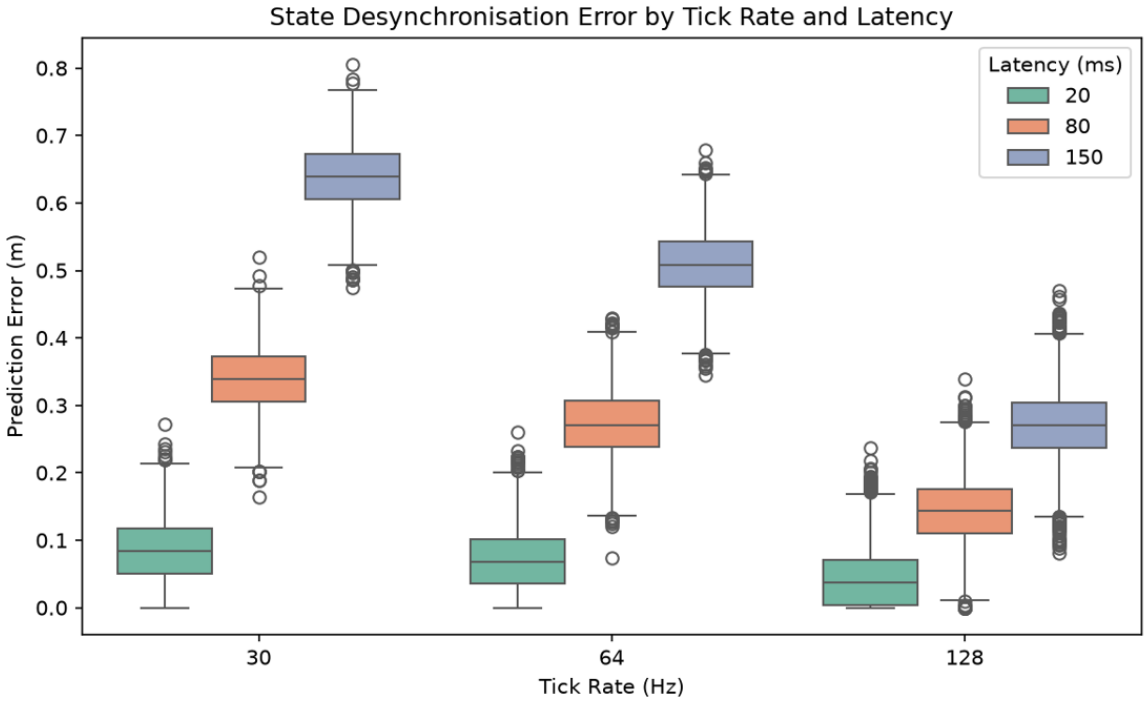


Figure 4.1 presents the distribution of state desynchronization error — measured as the Euclidean distance between the server-authoritative player position and the client-predicted position — across all tick rate and latency conditions.

At the lowest latency level (20 ms), prediction error remained small across all tick rates, with medians below 0.09 m. As latency increased, error grew substantially at every tick rate. At 30 Hz and 150 ms latency, the median error reached 0.64 m, representing a more than sevenfold increase compared to the 20 ms condition at the same tick rate. However, at

128 Hz and 150 ms latency, the median error was 0.27 m — a 57% reduction relative to 30 Hz at the same latency.

Higher tick rates also reduced the spread of errors within each latency group, as indicated by narrower interquartile ranges in the box plots. This suggests that finer temporal resolution not only lowers average prediction error but also makes reconciliation outcomes more consistent across ticks.

4.1.2 Hit Registration Accuracy

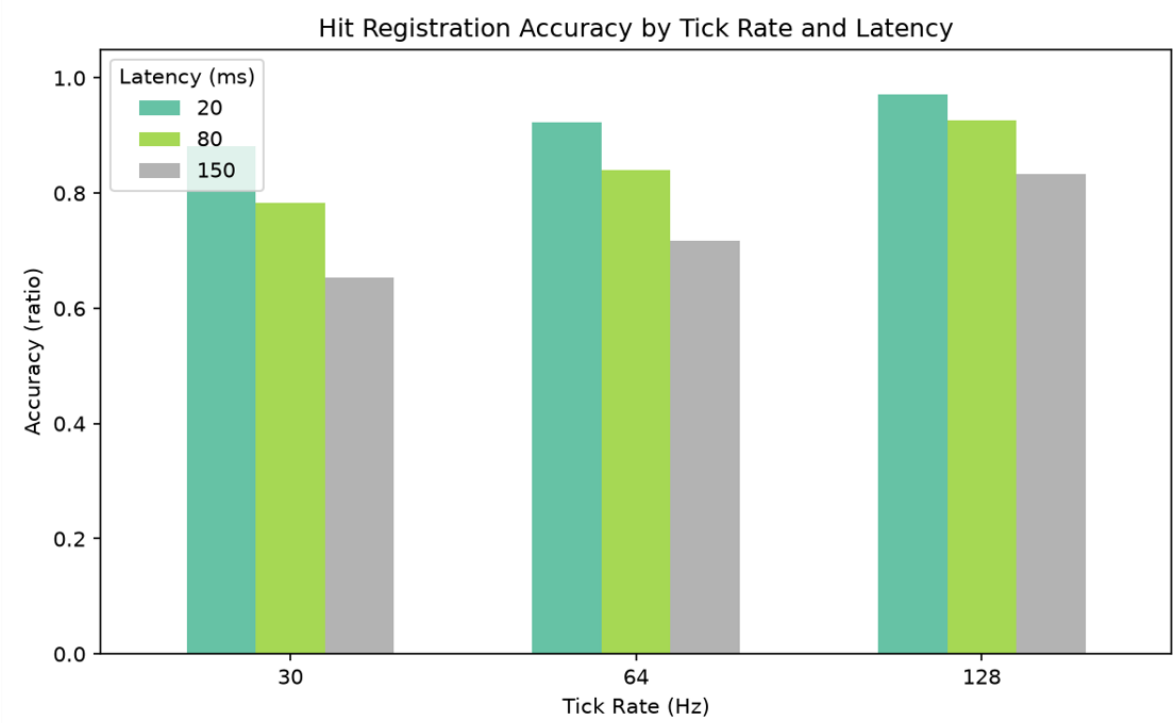


Figure 4.2 shows hit registration accuracy — the proportion of client-fired shots confirmed by the server within one tick of the predicted impact — for each condition.

Hit accuracy declined with increasing latency at every tick rate. At 30 Hz, accuracy fell from 0.882 (20 ms) to 0.653 (150 ms), a drop of 23 percentage points. At 128 Hz, the equivalent decline was from 0.971 to 0.833 — a smaller absolute reduction starting from a higher baseline.

Increasing tick rate consistently improved accuracy at every latency level. The largest improvement occurred at 150 ms latency: moving from 30 Hz to 128 Hz raised accuracy

from 0.653 to 0.833 (+18 percentage points). Notably, 128 Hz at 150 ms latency (0.833) outperformed 30 Hz at 80 ms latency (0.783), demonstrating that a higher tick rate can partially compensate for poorer network conditions.

4.1.3 Bandwidth Consumption

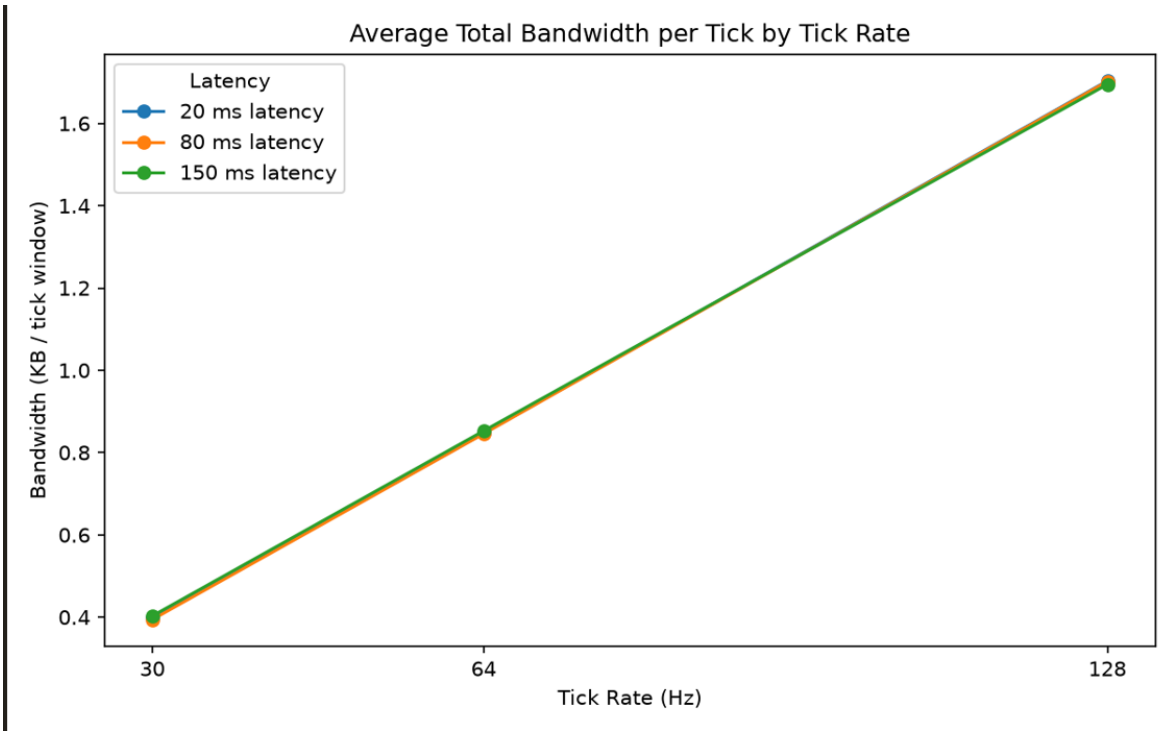


Figure 4.3 shows the average total bandwidth per tick window as a function of tick rate. Bandwidth increased approximately linearly with tick rate: from 0.4 KB/tick at 30 Hz to 0.85 KB/tick at 64 Hz and 1.7 KB/tick at 128 Hz. The three latency conditions produced nearly overlapping lines, indicating that simulated latency had negligible effect on per-tick bandwidth consumption. Mean bytes sent per tick rose from 239 B (30 Hz) to 1,026 B (128 Hz), representing a 4.3× increase.

4.1.4 CPU Frametime

Mean CPU frametime increased modestly with tick rate: 16.5 ms at 30 Hz, 17.1 ms at 64 Hz, and 18.1 ms at 128 Hz. Latency level had no meaningful effect on CPU usage. Even at the highest tick rate, mean frametime remained below a 20 ms frame budget, indicating

that the prototype's computational overhead at 128 Hz was manageable on the test hardware.

4.2 Analysis of Tick Rate Impact

One-way ANOVA was performed to test whether server tick rate had a statistically significant effect on each performance metric. Table 4.2 summarises the results.

Table 4.2. One-way ANOVA results and Tukey HSD post-hoc comparisons for tick rate.

Metric	F-statistic	p-value	30 vs 64 (mean diff.)	30 vs 128 (mean diff.)	64 vs 128 (mean diff.)
Prediction error (m)	4,409.8	< 0.001	−0.070*	−0.201*	−0.131*
Hit accuracy	2,701.5	< 0.001	+0.054*	+0.138*	+0.084*
CPU frametime (ms)	1,711.5	< 0.001	+0.60*	+1.65*	+1.05*
Bytes sent	25,371.0	< 0.001	+271*	+784*	+513*

All pairwise comparisons significant at p-adj = 0.0 (Tukey HSD).

Tick rate had a statistically significant effect on all four metrics ($p < 0.001$). Tukey HSD post-hoc tests confirmed that every pair of tick rates differed significantly for every metric.

For state desynchronization, the largest mean reduction was observed between 30 Hz and 128 Hz (−0.201 m). This supports the hypothesis that finer temporal resolution reduces the accumulation window for prediction errors before server reconciliation occurs. For hit registration, the corresponding accuracy gain was 0.138 (13.8 percentage points) between 30 Hz and 128 Hz.

Although latency was not included as a factor in the ANOVA model, descriptive results show a strong latency effect. At 30 Hz, increasing latency from 20 ms to 150 ms raised mean prediction error by 652% (0.085 m to 0.638 m). At 128 Hz, the same latency increase caused a 517% rise but from a substantially lower baseline (0.044 m to 0.271 m). This

pattern suggests an interaction between tick rate and latency: higher tick rates mitigate the negative impact of latency on both synchronization and hit registration, consistent with the theoretical framework presented in Chapter 2.

4.3 Trade-offs and Performance Evaluation

The results reveal a clear trade-off between gameplay accuracy and system resource consumption.

At 30 Hz, the prototype consumed the least bandwidth (~239 B/tick) and CPU (~16.5 ms frametime), but produced the worst synchronization (up to 0.638 m error) and the lowest hit accuracy (0.653 at 150 ms). This configuration may be acceptable for large-scale titles where server load and bandwidth costs are prioritised over competitive precision, such as the Battlefield series.

At 64 Hz, bandwidth approximately doubled relative to 30 Hz while prediction error and hit accuracy improved substantially. This tick rate aligns with the industry standard used by Counter-Strike: Global Offensive on public servers and represents a practical balance between performance and cost for mainstream competitive titles.

At 128 Hz, bandwidth increased by a factor of 4.3 and CPU frametime by approximately 1.6 ms, but synchronization and hit registration reached their best levels. At 150 ms latency, 128 Hz achieved prediction error and hit accuracy comparable to 64 Hz at 80 ms latency (both ~0.27 m error and ~0.83 accuracy). This finding has practical implications for matchmaking and server selection: players on high-latency connections benefit disproportionately from 128 Hz servers, as the finer tick interval partially compensates for network delay.

Bandwidth scaling was linear and independent of latency, meaning tick rate selection can be driven primarily by available server CPU and upstream network capacity rather than the latency profile of the player base.

4.4 Limitations

Several limitations should be considered when interpreting these results. First, each experimental run lasted only 10 seconds, which may not capture long-session effects such as cumulative drift or memory pressure. Second, all tests were conducted on localhost, eliminating real-world network phenomena such as route variability, jitter, and congestion that would affect production deployments. Third, the experiment used a single server machine with one human client and three AI bots, which does not reflect the load characteristics of fully populated game servers. Fourth, packet loss was applied in fixed percentages paired with latency but was not isolated as an independent variable in the statistical analysis. Finally, the prototype used a hitscan weapon only; projectile-based weapons with travel time may exhibit different hit registration behaviour across tick rates.

5 Conclusions

This thesis set out to answer a practical question: does server tick rate actually matter for latency compensation in online FPS games, and if so, what do you pay for it?

To find out, I built a small multiplayer FPS prototype in Unity using Mirror. It included the three mechanisms discussed throughout this work — client-side prediction, interpolation, and server reconciliation. I then ran automated experiments across three tick rates (30, 64, and 128 Hz) and three latency levels (20, 80, and 150 ms), with packet loss set at 0%, 2%, and 5% to match each latency condition. In total, 45 runs were recorded, measuring prediction error, hit registration accuracy, CPU frametime, and bandwidth.

The results were clearer than I expected. Tick rate made a significant difference to every metric I measured ($p < 0.001$ in all ANOVA tests). Higher tick rates consistently reduced desynchronization and improved hit registration, even when latency was high. At 150 ms latency, 128 Hz performed similarly to 64 Hz at 80 ms — roughly 0.27 m prediction error and about 83% hit accuracy in both cases. That was one of the more surprising findings: tick rate does not just make things slightly better at the margins; it can noticeably soften the impact of bad network conditions.

Latency, as expected, had the opposite effect. At 30 Hz, pushing latency from 20 ms to 150 ms pushed mean prediction error from about 0.09 m to 0.64 m, and hit accuracy dropped from 0.88 to 0.65. At 128 Hz, the same latency increase still hurt performance, but from a much lower starting point. The box plots made this easy to see — higher latency not only raised the median error, but also made outcomes less predictable.

The cost side of the trade-off was more straightforward. Bandwidth scaled in a fairly linear way with tick rate and barely changed with latency. Going from 30 Hz to 128 Hz increased per-tick bandwidth by roughly four times, while CPU frametime only went up by about 1.6 ms. On the test machine, even 128 Hz stayed below a 20 ms frame budget, which suggests the bottleneck in a real deployment is more likely to be network capacity than CPU — at least for a prototype of this size.

These findings fit well with earlier work. Bernier (2001) described how finer server updates improve the accuracy of prediction and hit validation; the data here supports that

under controlled conditions. Claypool and Claypool (2006) argued that first-person games are especially sensitive to latency, with playability dropping once delay gets too high. That showed up clearly in my results at 30 Hz, though 128 Hz held up noticeably better at 150 ms. Armitage, Claypool, and Branch (2006) frame latency and packet loss as normal problems in online games rather than edge cases — which matches what I saw once latency and loss were introduced together.

That said, there are limits to how far these results can be pushed. Each run lasted only 10 seconds, everything ran on localhost, and the setup used one client plus three bots rather than a full server. Packet loss was tied to latency rather than tested on its own, and the weapon was hitscan only. A longer study on real network paths, with more players and projectile weapons, would give a fuller picture. A two-way ANOVA on tick rate and latency would also strengthen the statistical side, since the interaction between those two factors was visible in the charts even if it was not formally tested in the analysis I ran.

Still, the main takeaway is practical. Tick rate is not just a server setting that competitive players argue about online — it directly affects how well latency compensation works. If the goal is fair, responsive gameplay — especially for players on higher-ping connections — 128 Hz is worth the bandwidth cost. If resources are tighter, 64 Hz remains a reasonable middle ground. The right choice depends on who you are building for and what your servers can handle, not on a single “best” number for every game.

References

Armitage, G., Claypool, M. and Branch, P. (2006) *Networking and online games: Understanding and engineering multiplayer internet games*. Chichester: John Wiley & Sons. Available at: https://doi.org/10.1002/047003047X (Accessed: 23 June 2026).

Bernier, Y. W. (2001) 'Latency compensating methods in client/server in-game protocol design and optimization', *Proceedings of the Game Developers Conference 2001*, San Jose, CA, March 2001. Available at: https://www.gamedevs.org/uploads/latency-compensation-in-client-server-protocols.pdf (Accessed: 23 June 2026).

Claypool, M. and Claypool, K. (2006) 'Latency and player actions in online games', *Communications of the ACM*, 49(11), pp. 40–45. Available at: https://doi.org/10.1145/1167838.1167860 (Accessed: 23 June 2026).

Mirror Networking (2024) *Mirror: Open source networking for Unity*. Available at: https://github.com/MirrorNetworking/Mirror (Accessed: 23 June 2026).

Unity Technologies (2024) *Unity user manual 2022.3 LTS*. Available at: https://docs.unity3d.com/2022.3/Documentation/Manual/ (Accessed: 23 June 2026).