Master thesis for the degree course Computer Science and Media

Deterministic Lockstep in Networked Games

presented by **Paul Mieschke**
Matr. number: 42621
Email: mail@pmieschke.com

at Hochschule der Medien Stuttgart
on February 29, 2024

for obtaining the academic degree of
Master of Science

First examiner: Prof. Dr. Stefan Radicke
Second examiner: Dr. Stefanie Schrader

Hochschule der Medien Stuttgart
Germany

Ehrenwörtliche Erklärung

Hiermit versichere ich, Paul Mieschke, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: “Deterministic Lockstep in Networked Games” selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 23 Abs. 2 Master-SPO (Vollzeit)) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Paul Mieschke, February 29, 2024

Abstract

Multiplayer games can increase player enjoyment through social interactions, cooperation and competition. The popularity of such games is shown by current market trends. Especially networked multiplayer games frequently achieve great success, but confront game developers with additional networking challenges in the already complex field of game production. The primary challenge is game state synchronization across all players. Based on the current research, there are three main methods for this task – deterministic lockstep, snapshot interpolation and state-sync – with their own advantages and disadvantages.

This work quantitatively evaluated and discussed the vertical (entity count) and horizontal (player count) limitations of deterministic lockstep and compared the method to snapshot interpolation. Results showed, that deterministic lockstep has no indicated vertical scaling limitation with a player count of up to 10 supporting 16,000 or more entities. A horizontal scaling limitation could not be found either and lockstep was confirmed to work with 40 or more players while handling 1024 entities. However, both scaling dimensions correlate negatively, which was indicated by the maximum scaling configurations 30 players and 4096 entities or 20 players and 8192 entities.

An unoptimized snapshot interpolation implementation achieved a vertical scaling limitation of 4096 entities with 10 players and a horizontal scaling limit of 40 or more players with 1024 entities and therefore was found to have a lower entity limit compared to deterministic lockstep.

Furthermore, results are compared to related work. Other contributions of this thesis include an overview of game networks and the three game state synchronization techniques. An architecture model for deterministic lockstep including a hybrid approach combining it with snapshot interpolation for re-synchronization and hot-joins. And finally, a network packet deconstruction of the implemented networking framework Unity Transport Package (UTP).

Abstract in German

Multiplayer-Spiele können die Spielfreude durch soziale Interaktionen, Zusammenarbeit und Wettbewerb steigern. Die Beliebtheit solcher Spiele wird durch aktuelle Markttrends gezeigt. Insbesondere vernetzte Mehrspieler-Spiele erreichen häufig großen Erfolg, stellen die Spieleentwickler jedoch vor zusätzliche Herausforderungen im bereits komplexen Feld der Spieleproduktion. Die primäre Herausforderung besteht darin, den Spielzustand zwischen allen Spielern zu synchronisieren. Basierend auf der aktuellen Forschung gibt es dafür drei Hauptmethoden – deterministic lockstep, snapshot interpolation und state-sync – mit jeweils unterschiedlichen Vor- und Nachteilen.

Diese Arbeit evaluiert und diskutiert quantitativ die vertikalen (Entitätsanzahl) und horizontalen (Spieleranzahl) Beschränkungen von deterministic lockstep und vergleicht die Methode mit snapshot interpolation. Die Ergebnisse zeigen, dass deterministic lockstep keine vertikale Skalierungsbegrenzung bei einer Spieleranzahl von bis zu 10 vorweist, während 16.000 oder mehr Entitäten definitiv unterstützt werden. Eine horizontale Skalierungsbegrenzung konnte ebenfalls nicht festgestellt werden, und es wurde bestätigt, dass Lockstep mit 40 oder mehr Spielern und 1024 Entitäten funktioniert. Beide Skalierungsdimensionen korrelieren jedoch negativ, was durch die maximalen Skalierungskonfigurationen von 30 Spielern und 4096 Entitäten oder 20 Spielern und 8192 Entitäten verdeutlicht wurde.

Eine nicht optimierte snapshot interpolation-Implementierung erreichte eine vertikale Skalierungsbegrenzung von 4096 Entitäten mit 10 Spielern und eine horizontale Skalierungsgrenze von 40 oder mehr Spielern mit 1024 Entitäten. Snapshot interpolation wurde somit eine geringere Entitätengrenze im Vergleich zu deterministic lockstep nachgewiesen.

Darüber hinaus werden die Ergebnisse mit verwandten Arbeiten verglichen. Weitere Beiträge dieser Arbeit umfassen einen Überblick über Spielnetzwerke und die drei Spielzustandssynchronisationstechniken. Ein Architekturmodell für deterministic lockstep, einschließlich eines Hybridansatzes mit snapshot interpolation zur Resynchronisation und für sogenannte Hot-Joins. Schließlich werden die Netzwerkpakete des Networking-Frameworks Unity Transport Package (UTP) analysiert.

Dedication

I would like to dedicate this work to my dear family and especially to my parents Iris and Lutz. I am very grateful for their heartfelt support during my studies and life and now I am looking forward to exploring the exciting future of our big family together.

Acknowledgments

I would also like to mention and thank my supervisors Prof. Dr. Stefan Radicke and Dr. Stefanie Schrader. Their knowledge and experience helped me along the way while writing this thesis. And even during holidays they found time to assist me, which is very much appreciated.

Contents

Chapter 1

Introduction

Since the inception of video gaming, multiplayer games have made it possible to increase player enjoyment through social interactions. “Pong”, one of the first major game releases, is a local multiplayer game. It is played on one computer with two controllers. Therefore, both players have to be at the same location in order to play together. Networked multiplayer games, however, allow players to play with each other by using multiple computers at different locations.

This introduces networking challenges both for locally networked multiplayer games in a local area network (LAN) and especially for online multiplayer games in a wide area network (WAN) like the internet. When playing a local multiplayer game on one computer the state of the game is guaranteed to be the same for all players. But for networked multiplayer games, where multiple distributed computers are involved, game state must be synchronized over networks, which is no trivial task. In order for a game state to be perfectly in-sync, every game entity must be synchronized so that the game states of each player do not diverge over time. Therefore, prior to the development of a networked multiplayer game one must carefully decide on the synchronization method to be used. Also, since game state information is communicated to other computers using network packets, latency, jitter, packet loss and other problems arise that the method must handle.

Based on the current state of research, there are three main methods for game state synchronization: deterministic lockstep, snapshot interpolation and state-sync [1], [2], [3]. With deterministic lockstep each client runs a deterministic game simulation in lockstep (synchronized timing). Only player inputs are shared between clients by sending them over the network. Therefore, deterministic lockstep yields low network traffic, but its requirement that the simulation must be deterministic complicates implementation [4], [5].

When using snapshot interpolation game state is synchronized by sending snapshots. Snapshots contain the complete or partial game state and are interpolated in order to achieve smooth transitions. This method can therefore lead to high network traffic, but, on the other hand, is easier to implement [6], [5].

State-sync combines deterministic lockstep and snapshot interpolation. Both inputs and states (snapshots) are shared. This means that the simulation must not be perfectly deterministic and there is no need for interpolation between snapshots since each client also simulates the game. This also means the snapshot interval can be reduced leading to lower network traffic. However, implementation is complex because two methods have to be implemented [7], [5].

By applying one of these three synchronization methods most singleplayer video games can be transformed into a networked multiplayer game. However, based on the requirements of a game genre the most adequate method must be selected, since they have different advantages and disadvantages. For example, strategy games with a high entity count should prefer deterministic lockstep, because snapshots may get very large in size. On the other hand, deterministic lockstep has a relatively high input latency and is therefore not well suited for fast-paced action games.

Not only games can benefit from networked synchronization methods. One example are simulations where computation must be distributed over several machines due to hardware limitations. Another example are serious games for educational purposes in schools and companies or for promoting social interactions [8].

Playing games with friends and other humans can increase player enjoyment by socializing, cooperating or competing with each other [9], [10]. Hence, it may be reasonable for game developers to devote additional effort on a multiplayer game or mode in order to prolong player attention.

The global games market is forecast to grow from a revenue of 201 billion US$ in 2021 to 312 billion US$ in 2027 [11] and is therefore twice as large as the worldwide theatrical and home/mobile entertainment market with 99.7 billion US$ revenue in 2021 [12]. Especially the multiplayer games sector is popular. This can be observed in figure 1.1. The figure depicts the ten most played games on Steam in 2023 by hourly average number of players. Eight out of ten games are primarily multiplayer games, while the other two offer multiplayer modes as well. However, although Steam is the largest games distribution platform mainly targeted at the PC market, figure 1.1 is not representative for the whole gaming market. Nevertheless, figure 1.1 is a clear indication of multiplayer popularity.

In summary, although networked multiplayer games pose new challenges to the already difficult field of game development, history and current market trends have proven that they may yield significant benefits for both players and game studios. There are three main methods for multiplayer synchronization, namely deterministic lockstep, snapshot interpolation and state-sync. Based on their advantages or disadvantages and the requirements of the game, the most suitable method can be determined.

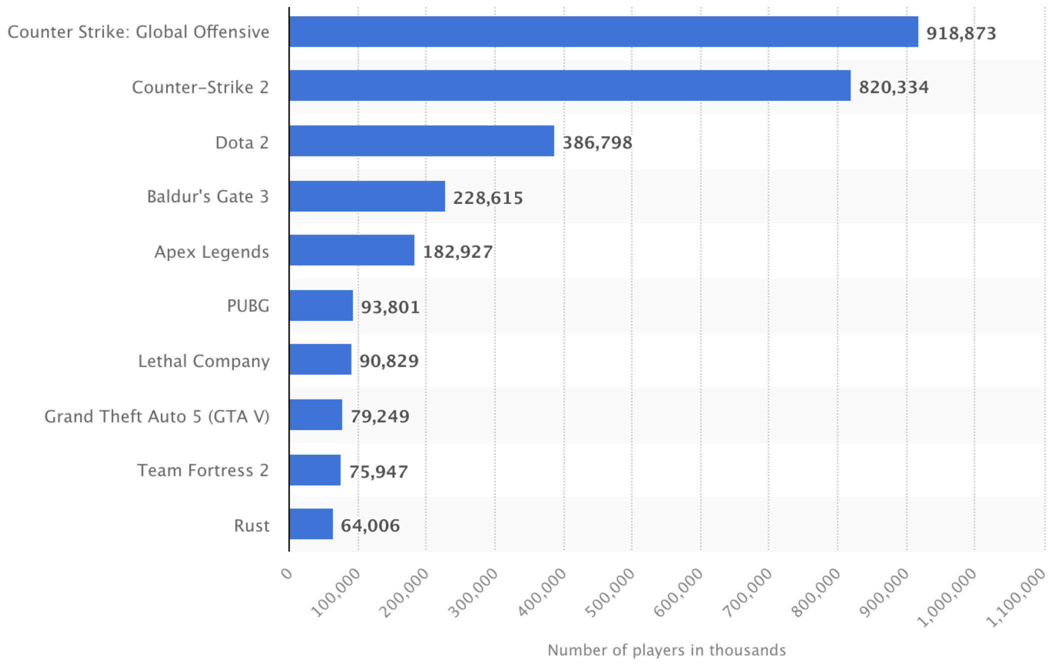


Figure 1.1: Most played games on Steam in 2023, by hourly average number of players [13].

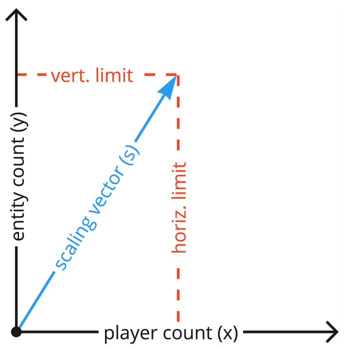
1.1 Scientific question

As mentioned in the introduction, developing networked multiplayer games comes with a new set of problems. Besides network latency, jitter and packet loss, each synchronization method has limitations. Snapshot interpolation and state-sync are limited by bandwidth, since they require players to share large game state data over the network [2]. This means that games with many entities like strategy games, where players control a high number of units, benefit from the deterministic lockstep approach of only sending player inputs. However, literature often mentions a limit of four [3] to eight [14] players for deterministic lockstep.

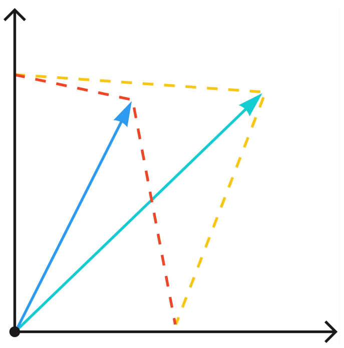
In this work these two limiting dimensions will be evaluated by scaling a sample game both vertically and horizontally. The vertical axis (y-axis) represents game entity count, while the horizontal axis (x-axis) indicates number of players. Figure 1.2 visualizes this approach. $\vec{s}$ is a scaling vector and points at a certain scaling configuration. The goal of this thesis is to find the maximum lengths of $\vec{s}$ in both dimensions: $\max|\vec{s}|$.

Scaling limitations can be divided into three categories: (a) independent limits, where the intersection of both limits yields one maximized scaling configuration;

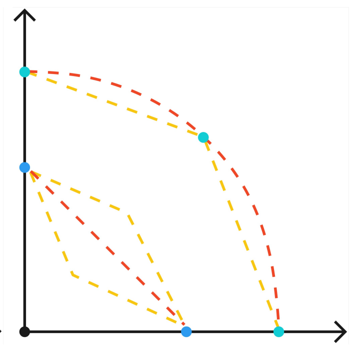
(b) dependent limits with one maximized scaling configuration; (c) dependent limits with two or more limits. In this context independence means, that the vertical and horizontal limit is not dependent on each other and therefore constant. Theoretically, there is a fourth category, where both limits are dependent and parallel. This would result in zero intersection points and a corridor of no limits. However, this case is unrealistic and thus omitted.



(a) independent limits with one max $|\vec{s}|$



(b) dependent limits with one max $|\vec{s}|$



(c) dependent limits with many max $|\vec{s}|$

Figure 1.2: Scaling graphs showing the two dimensions entity count (y-axis) and player number (x-axis), scaling vectors $\vec{s}$ (blue/cyan) and both vertical and horizontal limits (red/orange dashed lines).

In conclusion, the scientific question can be formulated as follows:

"What are the limitations in terms of vertical (entity count) and horizontal (player number) scaling of a networked multiplayer game and how does deterministic lockstep compare to a snapshot method?"

Snapshot interpolation was chosen for the comparison to deterministic lockstep. This is because snapshot interpolation is a counter-pole regarding limitations. It is also conceivable that both methods could be combined in future work, so that based on the game or network conditions and the found limitations of each method a dynamic switch between them can be performed. This way the disadvantages of snapshot interpolation (large game state data for many game entities) could be negated by the advantages of deterministic lockstep and vice-versa.

Another scenario where switching methods might be reasonable is when a game has both zoomed-out and zoomed-in phases. And, during the zoomed-in phase a lower-latency method is required, while the game's meta-state is still synchronized by deterministic lockstep.

1.2 Contributions

This thesis comprises the following contributions:

1. an overview of game networks and common game state synchronization methods
2. a comprehensive explanation of deterministic lockstep
3. an architecture and implementation model for deterministic lockstep including a hybrid approach with snapshot interpolation for re-synchronization and hot-joins
4. a deconstruction of Unity Transport Package (UTP) network packets
5. evaluation and discussion of the scientific question "What are the limitations in terms of vertical (entity count) and horizontal (player number) scaling of a networked multiplayer game and how does deterministic lockstep compare to a snapshot method?"

1.3 Thesis overview

In the next chapter "Related work" the current state of research regarding the scientific question of this thesis is investigated. After that, chapter 3 "Theoretical background" lays the knowledge base for later chapters by providing an overview of game networks and game state synchronization methods. Chapter 4 "Deterministic Lockstep" then explains the main synchronization method of this thesis in more detail, while chapter 5 "Implementation" proposes an architecture and implementation for deterministic lockstep. In chapter 6 "Evaluation" the answer to the scientific question is first evaluated and subsequently discussed using the implementation of chapter 5. The last chapter 7 concludes this thesis and topics for future work are proposed.

Chapter 2

Related work

In this chapter the current state of research regarding the scientific question of this work is investigated. Vertical and horizontal limitations are examined. Similar comparisons of synchronizations methods are presented. Additionally, related topics about qualitative research and quality of service (QoS) are explored, since they can be a limiting factor as well.

The frequently cited paper “1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond” [14] states, that the game Age of Empires is capable of synchronizing 1500 game entities and supports up to eight players using deterministic lockstep. They further mention by using a state-sync approach, where the 2D position (x, y coordinates), status, action, facing direction and damage of each unit is synchronized, the game would have been limited to a maximum of 250 moving units. Furthermore, qualitative user research concerning input delays was performed. 250 milliseconds of latency were not noticed by players, 250 to 500 milliseconds were “playable”, and beyond 500ms delay was noticeable and presumably had a negative effect on player enjoyment. These insights can be used in order to determine turn durations, a deterministic lockstep concept explained in chapter 4. The authors of the paper also found out, that in case dynamic turn durations are used, changes should be gradual and fast if turn duration must increase, but slow if the duration should decrease. However, Age of Empires is a real-time strategy game (RTS), which means due to the genre’s gameplay input lag is not as important as for other game types.

This genre difference is also shown by the two papers “The Effect of Latency on User Performance in Warcraft III” [15] and “The Effects of Loss and Latency on User Performance in Unreal Tournament 2003” [16]. Similar to the Age of Empires paper, even high latency, while noticeable, has no significant impact on game outcome in Warcraft III. This is attributed to the strategical over real-time aspect of strategy games. Also, inputs are carried out over time and there is no immediate output action like shooting in a first-person shooter game (FPS). For fast-paced games like shooters latency has negligible impact during movement but degrades user performance when shooting, since high precision and timing is needed in order to hit desired targets. 75 to 100 milliseconds showed an up to 50% accuracy and kill rate decrease. 100ms were subjectively noticeable and 200ms “annoying” to players.

Similar QoS results are shown in table 1 of [17] for RTS and FPS games. Additionally, car racing games prefer 100ms or less latency. 100-200ms felt “sluggish” and input delays of more than 500ms lead to loss of car control.

[2] and [3] both state that the vertical limitations of deterministic lockstep are not constrained by bandwidth, assumed input data is realistically small. However, in contrast to [14] a maximum player number of only four is suggested. This is due to the correlation between player count and the probability of a player lagging behind. So, if one player has a slower computer or worse network connection all players have to wait creating lags before each lockstep turn. When using snapshot interpolation game delays do not result from a player being slower, except it is the host. But, the vertical limit is constrained by maximum bandwidth, since more game entities yield more data.

[18] evaluated server performance on the game Quake using a high-performance computer of 2003. They found there is a horizontal limit of 60-100 players. This was due to a processor rather than network bandwidth bottleneck. Processor utilization increased linearly with the number of players. Quake uses the snapshot interpolation method [1].

The work presented so far already provides a base frame for multiplayer limit expectations. But, most sources and the games that were analyzed can be partially considered out-of-date by now. Technology evolves rapidly, which means updated research is beneficial. The recently released article [19] summarizes current generation networking standards. It is stated that Fortnite supports 100 players with 40-50 thousand synchronized game entities. Planetside 2 achieves an upper limit of the current generation through a set of connected servers (shards), where each of these shards is capable of handling up to 2000 players. However, in order to achieve these high player counts some fidelity must be sacrificed, e.g. projectiles are not replicated.

[19] further proposes a theoretical player limit for deterministic lockstep. For that the following assumptions are made: only player characters are synchronized, player input data includes movement with six degrees of freedom and 20 abilities resulting in 32 bits being sent at 30 Hz (960 bits per second), with a 10 gigabits/s link the server is able to receive 10,416,666 input network packets per second. Based on these conditions [19] calculates the player limit to be $\sqrt{10,416,666} = 3227$.

With respect to the scientific question the state of related work can be summarized as follows:

- vertical limitations of deterministic lockstep range from 1500 entities to practically unlimited by bandwidth
- horizontal limitations of deterministic lockstep vary between 4 to 8 and a theoretical limit of 3227 is proposed
- vertical limitations of snapshot methods range from 250 to 40-50 thousand entities
- horizontal limitations of snapshot methods vary between 60 to 100, but can be as high as 2000 in heavily optimized games
- input delay impacts vary between game genres
- delays of up to 500ms are considered “playable” for RTS games
- fast-paced shooter and car racing games prefer delays to be less than 100ms

Chapter 3

Theoretical background

This chapter conveys fundamental knowledge about game networking and game state synchronization, so that following chapters can investigate more specialized topics. However, general knowledge in the computer science fields of networking and games is assumed.

3.1 Game networks

Global networks are a cornerstone of modern technology and society. They achieved significant advances by connecting the world. Every day, a massive amount of data is sent over the internet by applications that may involve a large number of distributed computers, such as online multiplayer games. On the other hand, the internet is not perfect. Data traveling from one side of the world to the other can take up to multiple 100 milliseconds. And data may arrive late or not at all. Problems like these make networking especially challenging for real-time applications, where speed is an important factor. Many games are in real-time, e.g. RTS, FPS and car racing games. Hence, game networks must be specialized in order to combine the synchronous nature of games with the asynchronous nature of networks.

Some terms and definitions should be clarified specifically in a game networking context, since literature is not always consistent.

Multiplayer games can be categorized into local multiplayer games, played on one computer (e.g. split-screen games), and networked multiplayer games, played on multiple computers. For this thesis only networked multiplayer games are relevant. Networked multiplayer games further can be divided into locally networked multiplayer games, played in a local area network (LAN), and online multiplayer games, played in a wide area network (WAN) like the internet.

A server runs a headless version of a game (without visual output) handling multiplayer communication and synchronization. Clients are game instances of players sending inputs and receiving data from another client, server or host. A host is a combination of server and client on one computer handling both tasks. Authoritative servers have full ownership of the game state and are the only computers that can be trusted in a game network.

Network packets contain data packaged together meant to be sent, or transmitted (Tx), over a network. Network packets are then received (Rx) and can also be distributed or broadcast. Distributing a network packet means that after it has been received from a client (origin) it is transmitted to all other clients excluding the origin (Dx). On the other hand, broadcasting transmits the packet to all other connected clients and servers (Bx). Finally, a network message is serializable data transmitted between clients, servers and hosts.

Network bandwidth is the maximum data transfer capacity of a connection in a certain amount of time (usually per second). Round-trip time (RTT) is the time it takes a network packet from its origin to the destination and back.

3.1.1 Topologies

A network topology determines how computers are connected to each other. One of the most common topologies is the client-server model illustrated in figure 3.1.

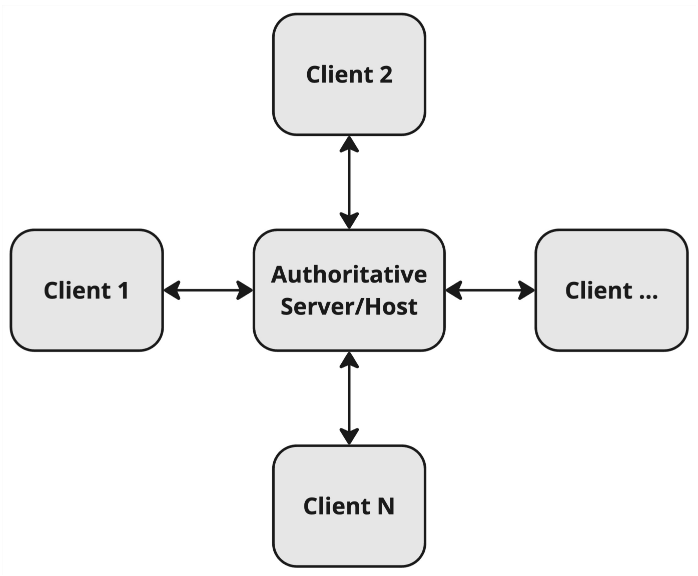


Figure 3.1: Client-server topology.

In this architecture every client is only connected to the authoritative server or host, who is responsible for distributing information across the network. N is the number of clients and each of them transmits b bytes per second. c bytes/s are broadcast by the server. This results in the following bandwidth requirements: the server must have capacity for $N \cdot b$ incoming and $N \cdot c$ outgoing bytes; clients need a capacity of b outgoing and c incoming bytes. Therefore, bandwidth requirements increase linearly with the number of clients. The server must maintain N connections ($O(N)$) and clients exactly one ($O(1)$) with a total of $O(2N)$ connections in the network [1], [20].

By using a host instead of a dedicated server operation costs are reduced, because, due to a client being also the server, no additional hardware is required. Technically, in this case the number of clients increases by one. But, bandwidth and connection count remain unchanged. However, processor demands rise and it is beneficial if the most capable computer in the network is hosting the game.

Another topology is the peer-to-peer (P2P) model shown in figure 3.2.

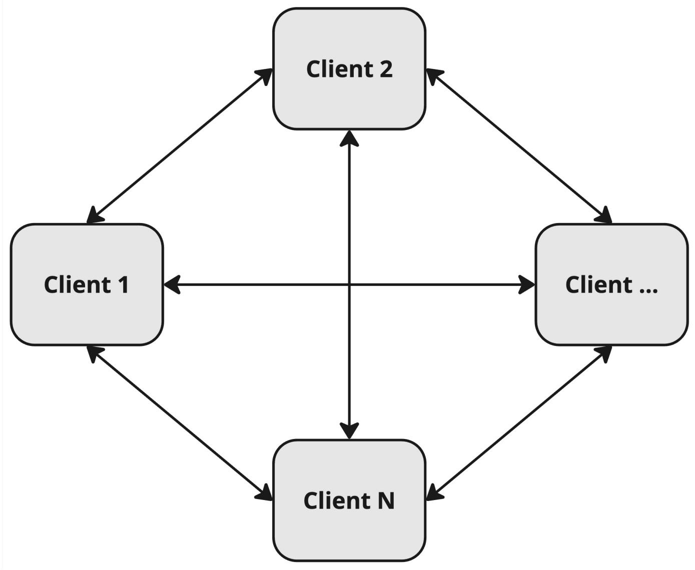


Figure 3.2: Peer-to-peer (P2P) topology.

In P2P networks every client, now called peer, is connected to every other peer with no central authority. Both the required bandwidth and connection count increase with the number of peers linearly ($O(N-1)$), while the total connection count in the network is a quadratic function ($O(N^2)$). Every peer needs the same amount of incoming and outgoing bandwidth.

One advantage of P2P is the reduced latency since information is shared directly with each other instead of first passing it to a server. P2P is also more failure resistant. If one peer disconnects the network is still functional, while a failed server renders the whole network inoperative. On the other side, cheating becomes more of a problem, since there is no authoritative node responsible for the game state [1], [14], [20].

More topologies and variations exist [21], but most games use either a client-server or P2P architecture as a base.

Finally, one problem should be mentioned. When using a P2P or client-host topology and connections should be established over the internet, firewalls and network address translation (NAT) of each clients local network become a challenge. Firewalls can discard unknown incoming packets based on their configuration. The main reason routers use NAT is to combat IP address scarcity. This is done by giving each device connected to the router a local IP. Then, only the router itself uses a scarce public IP. When a device opens up a connection to a device outside of the local network, the router assigns a public port number to this connection. This way incoming packets can be routed from the router to the local device. However, this mechanism hinders P2P connections, since public port numbers are not known by the peers.

It is possible to overcome firewalls and NAT by port-forwarding, which requires some technical know-how, or various hole-punching techniques like session traversal utilities for NAT (STUN), but there is still one particular NAT type (symmetric), that makes true P2P impossible [1]. For this case, as a fallback with guaranteed connection establishment, relay servers are used (traversal using relays around NAT (TURN)). A relay (or proxy) is a publicly available server without misconfigured firewalls and NAT, that only forwards traffic between clients. So, the operating costs are lower compared to a dedicated server. A disadvantage is that the premise of true P2P is not anymore achieved and the topology is similar to a client-server model (see figure 3.1 where the server is now a relay server). This again increases latency based on the location of the relay in relation to all clients. Figure 3.3 shows the case where a relay is used in a client-host topology.

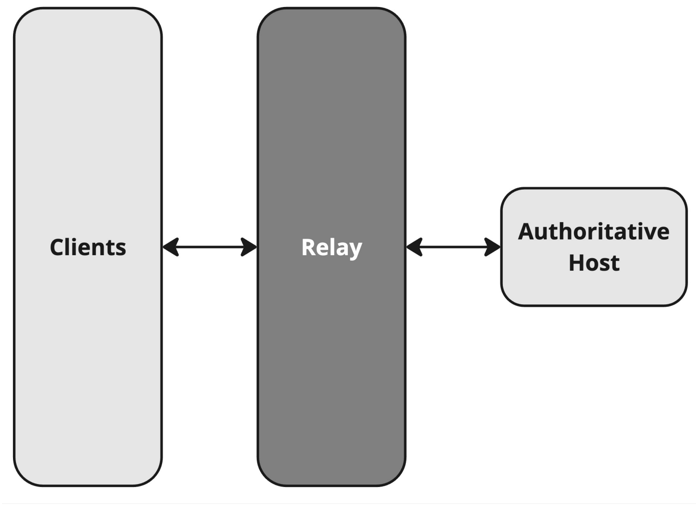


Figure 3.3: Client-host topology using a relay.

3.1.2 UDP versus TCP

When developing a networked game the fourth or “transport” layer of the open systems interconnection model (OSI) provides two options. The connection-based transmission control protocol (TCP) comes with built-in packet reliability and sequencing as well as router congestion control in order to reduce packet loss. The user datagram protocol (UDP) does not have these features and treats packets in a “fire-and-forget” way. However, UDP can be used as a base framework for transmission, that can be expanded with connection/session logic, reliability, sequencing and tailored to specific applications. This makes UDP more efficient, but at the expense of development time [1], [2], [22].

For example, TCP is suitable for turn-based games, where network messages must be reliable and speed is not an important factor. Another example are in-game chats. But, for real-time games, where speed matters or waiting for already outdated packets (head-of-line blocking) is obstructive, UDP is the better choice. Bad network conditions also seem to impact TCP more than UDP according to [3]. An RTT of 250ms and 5% packet loss with TCP rendered the game unplayable, while UDP had no problems even with 250ms and 25% loss.

Finally, it should be mentioned, that TCP’s congestion control mechanism can prevent data from being sent for up to half a second. Therefore, it can make sense to disable the Nagle algorithm. This reduces latency at the cost of an increased packet count. Also, in case of network congestion routers may prioritize TCP and drop UDP packets first [1].

3.1.3 Challenges

Networking a game comes with a range of new challenges. One set of challenges is introduced inherently through network functionality. **Latency** is the delay of information exchange due to the time network packets need to reach their destination (RTT). Realistic values range from a few milliseconds in LAN or high-speed connections to multiple 100 milliseconds for connections around the world. An RTT of below 100ms is desired [1]. **Jitter** is the variation of latency over time and usually is about half the latency [23]. **Packet loss** is an undesired event in which network packets do not reach their destination. Common values are between 1-5% [22]. **Bandwidth limitations** must be accounted for by carefully choosing which data must be transmitted and how the data can be compressed. Today’s bandwidth standards are quite high at an worldwide average of 80 MBit/s download and 35 MBit/s upload [24]. Additionally, these factors vary strongly based on the network conditions of a client and therefore a networked game should be tested also in worse-case scenarios [1], [21].

Another challenge is inherent through the nature of games: **cheating**. While usually not a concern in singleplayer games, cheating in multiplayer games is unfair for other players and can deteriorate their gaming experience or even cause player loss. One effective way to combat cheating is by using an authoritative topology, so that the source of truth is controlled by a server or host [25]. However, in the latter case the host still would be able to manipulate game state. Locally, cheating could be prevented by memory obfuscation or regular anti-cheat applications [5], which is out of scope for this thesis.

Finally, the main challenge for networked multiplayer games is **game state synchronization** leading to the next section.

3.2 Game state synchronization

Networked multiplayer games involve multiple players and therefore clients. In order to play together one definitive representation of the game, a game state, must be synchronized between all clients over networks. Being in-sync across all clients is achieved if all game entities, e.g. the player character, units, non-playable characters (NPCs) or game's meta information like player gold, share the same state at a certain point of time across all clients. State can consist of for example position, rotation, health and action.

Various synchronization methods are known. All of the presented methods in this work include sharing player commands by transmitting them to the server/host or all peers depending on the topology. Commands are player inputs that translate to a game logic relevant event. Inputs are physical player interactions (e.g. clicks, key presses). How these commands are handled on the receiving side depends on the synchronization method. In this chapter both snapshot interpolation and state-sync are explored. The next chapter then provides a comprehensive explanation of deterministic lockstep, since it is a main topic of this thesis.

3.2.1 Snapshot interpolation

Snapshot interpolation can be used in an authoritative topology, where the server or host is simulating the definitive game state. Each client only sends their commands to the server, but no client-side game simulation is performed. In literature they are also called “dumb terminals”, since clients only display a recent game state and accept inputs [1]. When receiving commands, the server executes them and the game state is manipulated. In order to share the game state the server regularly broadcasts snapshots to every client. A snapshot is the game state at a specific time and therefore contains complete or partial information on how to replicate game entities. Clients receive snapshots and can then display the new game state to the player. By interpolating between snapshots smooth transitions can be achieved, so that game entities do not jump between states. Therefore, an interpolation buffer with old and new snapshots must be maintained having the negative side effect of increased delay. Instead of using linear interpolation, which can cause subtle artifacts, Hermite interpolation yields more realistic results. Hermite interpolation makes use of an entities linear velocity, which therefore must be added to snapshot data (more bandwidth). Based on the start and end velocity Hermite interpolation creates a spline going through the start and end position of an entity [2], [6].

Snapshots are sent multiple times per second, e.g. at 10 Hz, and the more frequently they are sent the more current and accurate the game is for clients. However, broadcasting snapshots that often requires a high bandwidth especially for the server. Therefore, some optimization should be made to compress snapshot size. One approach is to quantize values like positions, rotations and velocities to an interval and range detailed enough. This includes some experimentation and careful thinking of the game's future demands. Another technique is delta compression. With this approach only changes of entity states are sent with snapshots, given that the initial state was set up equally for all clients. A disadvantage of delta compression is that clients now need to acknowledge received snapshots so that the server knows based on which snapshot new deltas can be computed [26].

3.2.2 State-sync

State-sync can be used with an authoritative topology as well and is a combination of both snapshot interpolation and deterministic lockstep (see chapter 4). The server/host receives client commands, distributes them and regularly broadcasts snapshots to all clients. Snapshot compression can be used here too. Clients again send their commands, but, in contrast to snapshot interpolation, they also simulate the game locally instead of only waiting for the next snapshot. Based on the last received snapshot and player inputs each client can extrapolate, or predict, future states [7].

Transmitted commands and snapshots contain a timestamp, so that they can be applied at the same moment. Due to latency these timestamps are usually already in the past when receiving data. Reconciliation, or rollback, is used to realign the game state based on a new received command or snapshot. On the server-side the simulation is rewind to the time of the command, the command is applied and the simulation fast-forwarded to the original time again. This also compensates for lag, e.g. when a player shoots at another character, since the server reconstructs the state to the moment of the player input [25]. On the client-side, first, the checksum of the received snapshot is compared with the own checksum at that moment.

In case they diverge the client simulation is rewind, the snapshot applied and the simulation is fast-forwarded again. Based on the accuracy of the state extrapolation only small game entity jumps are possible, that can be concealed by also applying interpolation [5], [7].

Lastly, state-sync can make use of a priority accumulator queue. Bandwidth usage may be reduced by only sending the most important entities with a snapshot. Therefore, each entity is assigned an importance value, e.g. the player character has a high value and static objects only have low priority. Each entity is added to the sorted queue based on their priority value. When sending a snapshot the queue is iterated until a maximum bandwidth is reached. The remaining entities in the queue, that have not been updated by the snapshot, increase their priority, for example by doubling the value. This way important entities are updated frequently, but not exclusively [7].

Chapter 4

Deterministic Lockstep

In the following chapter a third and the main game state synchronization method of this thesis is explained. Deterministic lockstep is based on two principles. The game progresses in **lockstep** between all clients, which is described in the first section. And, the game simulation must be **deterministic**, investigated in the second section. To conclude the chapter, all three synchronization methods, that have been covered, are compared.

4.1 Method: Lockstep

With deterministic lockstep every client runs the complete simulation of a game and only commands are either broadcast to other clients or transmitted to the server or host, who then distributes commands. Therefore, deterministic lockstep can be used with a client-server as well as a P2P topology [1], [4].

All clients must be in lockstep with each other. This means that own and received commands are not executed immediately, but at a certain point of time, that is equal for every client's simulation. Because time may be ambiguous in this context, it is beneficial to define different variants: *absolute time*, the standard world time (universal time coordinated (UTC)); *application time*, since launch; *game time*, since the game phase. Hence, more precisely, commands should be executed at the same *game time* on every client. This moment may be at different *absolute* and *application* times. Once this premise is achieved all clients are in lockstep and commands are executed synchronously.

Lockstep makes use of turns. A turn is another discrete time variant and is increased by one after computation of a previous turn has been completed. Turns have a set time interval (duration) in which the player may issue commands scheduled for a later turn. They subdivide game time and are the mechanism to keep simulations in lockstep. At the end of each turn every client must have received all commands that are scheduled for the next turn. If that is not the case the simulation pauses until missing commands arrive.

The simulation consists of all game logic relevant systems and must remain in-sync across all clients. Therefore, a fixed update interval should be used. On the other hand, the presentation layer, or view, includes systems like user interface (UI), gameplay irrelevant animations or particles, that are allowed to be non-deterministic and therefore out-of-sync. Update intervals can be variable and this layer is not affected by lockstep pauses. Hence, the presentation layer is bound to game time, whereas the simulation is dependent on the progression of turns [27].

Turns are further subdivided into ticks. Ticks are fixed interval simulation updates, or steps. When the tick number of a turn reaches the ticks count per turn, a turn is at its end. Ticks also have a set interval called *fixed delta time*, that is determined by $\frac{turnDuration}{ticksPerTurn}$. Presentation layer update intervals are called *delta time*, which is a variable value based on the hardware demands of the game and capabilities of the computer.

Due to commands in turn N being scheduled for a later turn $N + x$, there is an inherent input to action delay. When the turn duration is 250ms and a command is scheduled for turn $N + 2$ this would result in a minimum delay of 250ms (command issued at the end of a turn) and maximum of 500ms (command issued at the start of a turn). In order to mitigate this effect the presentation layer can be used for immediate feedback, e.g. with animations, particles or audio, whereas simulation logic is executed later in lockstep [21].

In case of delayed commands players experience lags. Similarly to state-sync, predictive rollback can be used to reduce lags. So, instead of waiting for the commands of the next turn, the simulation predicts subsequent game states. When these predictions turn out to be incorrect upon receiving the commands the state is rolled back and the simulation fast-forwarded based on the now correct inputs. This case can lead to entity teleportation [14], [21].

Another lockstep optimization includes dynamic turn durations based on the worst RTT in the network. This can improve player experience in two ways: when all players have a low RTT input delays can be reduced; when a player has a bad RTT the number of lags can be reduced in exchange for increased input delay.

Lockstep of the simulation between all clients is required for the game to stay deterministic, since differently timed command executions result in diverging game states. This leads to the next section: determinism.

4.2 Method: Determinism

In the context of games "determinism means that given the same initial condition and the same [sequence of] inputs [the] simulation gives exactly the same result" [4]. This requires all commands to be executed in lockstep from a common initial game state. Besides that, there are many non-deterministic systems in a game that can cause desynchronization.

The first and foremost source of non-determinism is random. Fortunately, most random implementations are pseudo random number generators (PRNGs) that, given an initial seed, output the same sequence of numbers. If the seed is synchronized before game start every client generates the same set of numbers. Nevertheless, there are some precautions to be taken. Some PRNGs yield different sequences based on the compiler, operating system and hardware used. Execution order of code can be another pitfall. For example, given the code: [20]

```
SomeFunction(GetRandom(), GetRandom());
```

The order of the two `GetRandom()` calls is not guaranteed to be the same across compilers. A solution to this uncertainty is achieved by splitting up the instruction:

```
random1 = GetRandom();
random2 = GetRandom();
SomeFunction(random1, random2);
```

Data structures like arrays and lists must be ordered in some cases [5]. For example, when elements are accessed by an index shared over the network. Or, when iterating over all elements and the logic of each iteration is influenced by order, e.g. random number generation.

Similarly, game engines can use non-deterministic update loops for entities. So, the order of execution of each entity's logic is not guaranteed [5]. An entity management system is a possible solution to this by storing references to each entity in an ordered data structure, that is iterated to call each entity's update logic deterministically. Such a system is explained in chapter 5.

Furthermore, a central data type to many game and physics engines as well as frameworks can behave non-deterministically. Floating-point number calculations possibly lead to small deviations on different compilers, operating systems and hardware, that can aggravate to problematic game state diversions [14]. To circumvent this risk, integers or fixed-point arithmetic can be utilized [5], [20], [28].

When using game engines, physics engines [4], [5] and other external frameworks, it is mandatory to make sure that every external system used in the simulation layer is deterministic by investigating the documentation or experimentation. Special care must be taken with static PRNGs, since they can be called by external systems in an uncontrolled manner and could thus manipulate the random number sequence of internal systems [29]. The presentation layer, however, is free to use any arbitrary random, since game logic is not influenced. This is also why the simulation layer should not be dependent on the presentation layer, but the other way around [5].

4.3 Desynchronization

Desynchronization is an undesired condition, that happens when game states of two or more clients diverge – they are out-of-sync. This contradicts the goal of multiplayer game synchronization. A sophisticated deterministic lockstep implementation, that takes all precautions mentioned previously into account, should not desynchronize. However, keeping a game strictly deterministic requires a considerable amount of knowledge and effort. Therefore, it is important to brief developers contributing to netcode [14]. Nevertheless, human error is always possible and thus it is recommended to implement additional systems in order to detect and handle desynchronization.

Desynchronization can be detected with checksums [14], [20], [27]. A checksum is a value generated based on data, that is used to determine data integrity. In the context of multiplayer game synchronization it is usually a number generated based on a game state. Since numbers are significantly smaller in size than game states, sharing them over the network frequently is unproblematic. Still, building checksums might be complex and place a heavy load on the hardware based on the used algorithm or large game states. Checksum generation must yield the same number for the same game state. If this condition is fulfilled clients with the same checksum of their game states are in-sync. But, when their checksums differ they are out-of-sync and a desynchronization is detected.

In case a desynchronization was detected all clients must be resynchronizied to prevent further game state divergence and possible completely different player experiences. This can be done by sharing an agreed-upon definitive game state, usually a snapshot of the host's state. Based on this reinitialized state deterministic lockstep can once again attempt to keep all clients in-sync. This process is similar to savegame loading or hot-joins.

4.4 State synchronization summary

After covering three game state synchronization methods, they can be summarized visually in figure 4.1 and by category in table 4.1:

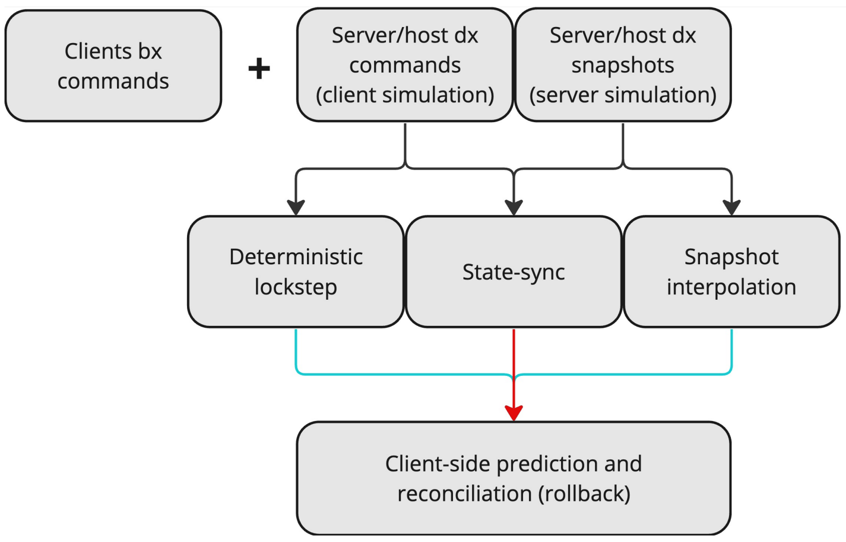

Clients bx commands
+
Server/host dx commands (client simulation)
Server/host dx snapshots (server simulation)
Deterministic lockstep
State-sync
Snapshot interpolation
Client-side prediction and reconciliation (rollback)

Figure 4.1: Game state synchronization summary. The red arrow means "uses", and the blue arrow means "could use, but is not part of a default implementation".

Category	**Snapshot interpolation**	**State-sync**	**Deterministic lockstep**
Compatible topologies	(authoritative) client-server	(authoritative) client-server	(authoritative) client-server and P2P
Game simulation	Server or host only	Server or host and clients	All clients
Networked data	Commands by clients and snapshots by server or host	Same as snapshot interpolation	Commands (snapshots only on desync)
Bandwidth usage	Relatively high	Relatively medium	Relatively low
Latency	Relatively medium	Relatively low	Relatively high
Challenges	Bandwidth limitations, snapshot size optimization	Implementation, adequate predictions	Determinism, possibility of desynchronization
Primary scaling difficulty	Vertical	Vertical	Horizontal
Cheating	The game state of other players cannot be manipulated, but concealed information can be extracted (e.g. remove FOW)		
Fairness	Host has advantage since data is there first	Same as snapshot interpolation	No advantages of any client due to lockstep
Hot-join / reconnect	Possible	Possible	Possible with snapshots
Genre recommendation	Similar to state-sync, less optimized	Fast-paced games with a conservative entity count, e.g. FPS or racing games	Games with many entities and marginal latency demands, e.g. RTS or turn-based games

Table 4.1: Game state synchronization summary based on categories.

Chapter 5

Implementation

Now that the theoretical foundation has been established, chapter 5 addresses the practical application of this knowledge. Some generalized models are presented, that provide a guideline for deterministic lockstep implementations. An example implementation has been developed based on these models in order to evaluate the models and especially the scientific question of this work. In the end, evaluation unrelated limitations of the implementation are discussed.

5.1 OSI model integration

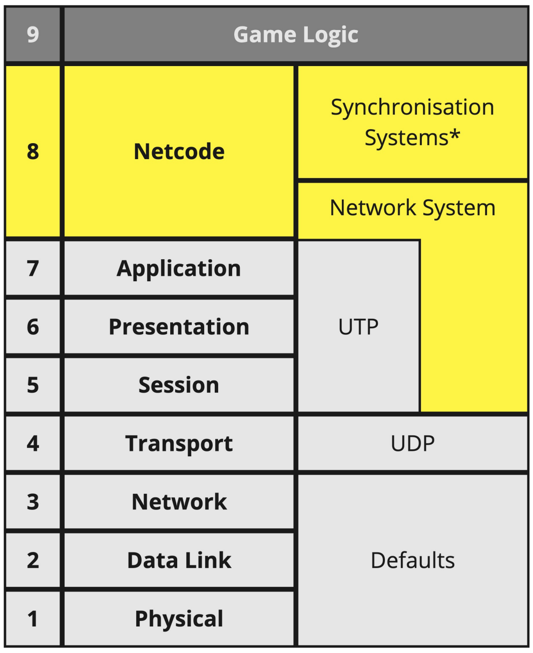


Figure 5.1: OSI model integration. Layer 1-7: OSI layers and their employed implementations. Layer 8-9: custom layers and implementations developed on top of the OSI model [30]. * e.g. snapshot system, lockstep system, ...

In order to provide a first overview and establish the implementation in the networking context, implementation details and custom layers can be added to the OSI model. This OSI model integration is depicted in figure 5.1. The figure includes all seven default layers as well as two custom layers above with their employed implementations to the right. Yellow highlighted cells indicate custom implementation code. In the following, each layer will be explained from bottom to top.

Layers one to three (physical, data link and network) build the backbone of the internet and together they allow data to be sent over networks across the world. Layer one handles transmission of data on the lowest level using signals representing bits, that are sent over a physical medium (e.g. cable or radio transmission). Layer two is responsible for sending chunks of data (frames) between computers inside a network, whereas the third layer converts frames into network packets and can handle transmission over multiple networks with the internet protocol (IP). Layer one to three use their respective default implementations for packet transfer.

In layer four (transport) UDP is chosen over TCP. Refer to section 3.1.2 for a comparison of both protocols. Another reason for UDP originates from the used low-level networking framework UTP, which builds on top of UDP. UTP resides in layers five to seven. Together with UTP, the network system handles session or connection management and network message processing. Additionally, the network system is part of the first custom layer: netcode.

Layer 8 (netcode) is a bridge between low-level networking and game logic. This layer abstracts away complex netcode by providing high-level systems with application programming interfaces (APIs), that can be used by the game logic. The network system implementation, for example, has methods for connection establishment and other session management functions. Building on top of that, one synchronization system must be used, that keeps game states in-sync. Synchronization system APIs should provide a method to transmit commands. A snapshot and deterministic lockstep synchronization system have been implemented for the evaluation, but, in general, implementations of the OSI layer integration model can be exchanged.

Layer nine contains game logic and is the last and topmost layer, that relies on all below layers. Here, gameplay is defined and high-level networking logic of the netcode layer is integrated. With this general overview of the implementation, the next section can go into more detail, particularly concerning layers five to nine.

5.2 Architecture model

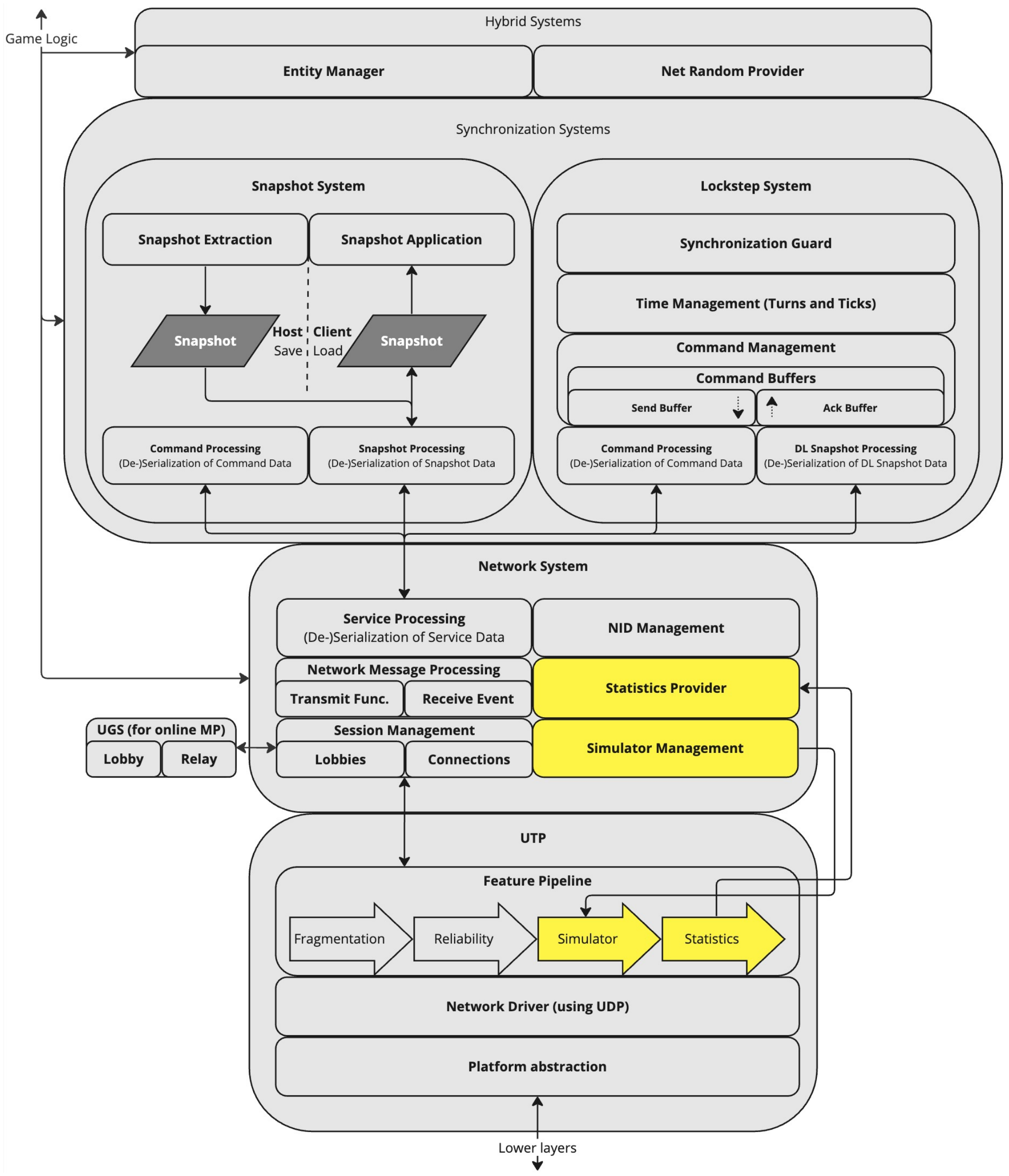


Figure 5.2: Implementation architecture model. The integrated OSI model layers five to eight and partially nine are shown in more detail from bottom to top. Each component builds on top of lower components. Yellow highlighted components may be disabled in production for a performance increase.

The implementation architecture model is illustrated in figure 5.2. It shows more details about the previously introduced OSI model integration layers five to eight and partially nine from bottom to top. Each system and component builds on top of lower ones. Yellow highlighted components may be disabled in production for a performance increase, since they only serve experimental or analytical but no functional purposes. To explain the figure further, each system and component is now described from bottom up.

5.2.1 UTP – Layer 5-7

The Unity Transport Package (UTP) is a low-level networking framework supporting client-server/host topologies and is developed for the Unity game engine. It provides convenient platform abstraction, since the game engine also has cross-platform support. Based on the platform either a UDP, for conventional devices, or web-socket, for web-based applications, network driver is used. The driver is responsible for the transport layer handling (layer four). Since, plain UDP does not support advanced features like TCP, UTP's feature pipeline can be tailored to a game's needs with additional functionality on top of UDP. There are four included pipeline stages [30].

The *fragmentation* stage splits data up into smaller chunks and multiple packets, that are below the maximum transmission unit (MTU) of approximately 1400 bytes. The *reliability* stage guarantees successful transmission of data, granted there is a possible route from origin to destination device. This is done similar to TCP with packet acknowledgments (acks). In case of packet loss the corresponding packet is resent until it arrives. Additionally, this pipeline stage sequences packets, so that the order of packet transmission is equal to the receive order. UTP also offers an *unreliable sequenced* stage if only order matters. Lastly, the *simulator* stage can be used to simulate certain network conditions like latency, jitter or packet loss and is practical for experimentation and limit testing, rather than for production [31].

There is also the possibility of custom pipeline stages, e.g. a *statistics* stage, that provides various insights after pipeline processing was completed [31]. This also implies that ordering of stages is important. For example, it makes sense to put the reliability stage after fragmentation, so that only fragments need to be resent after packet loss instead of the whole data, that was fragmented [23].

5.2.2 Network System – Layer 5-8

The network system is a custom high-level networking framework providing LAN and WAN networking functionality to both the synchronization systems as well as the game logic. At its core are the session management and network message processing components.

Although UDP is not connection-based, UTP implements protocols on-top for a connection-based communication. Therefore, the session management is responsible for connection management and lobbies. Connection management includes creating a listening host, creating a connecting client, connection establishment and disconnection by communicating with the UTP network driver. Lobby management includes hosting a lobby (host), joining a lobby (clients), lobby configuration, lobby player management and starting the game phase. There are two different procedures for connection and lobby management based on the targeted network (LAN or WAN). In a WAN the procedure is using an external service called Unity Gaming Services (UGS), that provides a UTP-compatible lobby and relay service in order to allow online multiplayer games over the internet. So, in this case a relayed client-server/host topology is used. The logic separation of these two procedures is only required up until the session management component and does not affect the following systems and components.

The network message processing component handles transmitting and receiving network messages. Transmitting is action-based and serializes a message before adding it to the transmit queue of the network driver. Receiving is event-based and at first only deserializes the network message header. "**Serialization** is the process of converting the state of an [entity] into a form that can be persisted or transported" [32]. In practice, when serializing an entity the resulting bytes can be saved to a hard-drive (savegames) or sent over networks. **Deserialization** is the opposite process, that converts a stream of bytes read from the hard-drive or received over the network into an entity state.

Only the network message header is deserialized by the network message processing. This is because further deserialization and processing is delegated to other specialized processing components based on the information extracted from the header, which contains a network message type.

A first specialized processor is the service processing component. Here, service related network messages are handled (header type byte is SVC). For example, the following service functionalities are implemented: connect, disconnect, heartbeat, lobby updates (LAN procedure only), start game, checksum, resync request, resynced. Sometimes network messages only serve a notification purpose with no additional data, but, for instance, the checksum message carries a data body containing the checksum value. Other network message processors and an overview of message types is addressed later in section 5.2.3.

Both the simulator management and statistics provider components are not recommended for production builds. They serve analytical purposes and provide crucial insights during development on the network performance of a game. With the simulator non-optimal conditions can be tested where latency, jitter or packet loss is high by setting the parameters of the simulator pipeline stage. Various statistics are collected from the statistics pipeline stage and are further explained in chapter 6. They can be displayed as graphs in real-time while running a game, or it is possible to write them to a file for in-depth analysis.

The final component of the network system handles NID management. A network ID (NID) is a unique, synchronized ID across all clients and servers. Game entities and connections, respectively players, all have a NID. There are various kinds of IDs, which suggests a summary:

- *Local IDs* are assigned to all objects and handled by the game engine, in this case Unity. They are not synchronized across clients and therefore same entities have different local IDs in other game instances.
- *Static NIDs* are assigned to static entities. These NIDs are fixed by game data and therefore every client inherently has the same static NIDs, granted the same build version is used. They are handled by the NID management component and examples of static entities are fixed buildings, factions and types in general (e.g. unit types).
- *Dynamic NIDs* are assigned to dynamic entities. Dynamic entities are spawned during runtime and therefore their NIDs must be synchronized, so that commands controlling a spawned entity by NID work on all clients. They are also handled by the NID management component and examples of dynamic entities are units or bullets.
- *Connection NIDs* are assigned to network connections and are based on the transmitted NID in a connect service network message (SVC). Hence, they are synonymous with client or player IDs. However, in case a client-server topology is used player NIDs must be distributed by the server or host. Connections and therefore connection NIDs are handled by the session management component.

When using a snapshot synchronization system dynamic NID generation and synchronization is straightforward. The server/host assigns a new NID if an entity is created and distributes the NID along with the entity's state via snapshots. However, it is a more intricate process for deterministic lockstep:

Client sends a command, that will spawn a new entity $\Longrightarrow$ Host receives the command and registers a new NID without an entity object association (prevents the unlikely case of generating duplicate NIDs while the entity is not yet spawned) $\Longrightarrow$ Host appends the NID to the command and adds the command to the ack buffer for a later execution (ack buffer is explained in section 5.2.3) $\Longrightarrow$ Host broadcasts the command (command is acked) $\Longrightarrow$ Clients receive the acked command and add it to their ack buffer $\Longrightarrow$ When the command's execution turn is reached all clients spawn the entity and associate it with the NID.

5.2.3 Synchronization Systems – Layer 8

Snapshot System

The snapshot system itself is responsible for a regular snapshot distribution on the server-side. In theory, it should also handle the interpolation of snapshots with buffers. However, in this implementation no interpolation is performed, since the focus is on deterministic lockstep.

Snapshot extraction is the process of creating a snapshot at a certain game time based on all entity states retrieved from the entity manager and other arbitrary serializable data that can be provided by the game logic (opt-in non-entity data, e.g. abstract information like player gold or experience). This is only performed on the server or host and also can be used as a savegame system.

The clients, on the other hand, only receive snapshots. Snapshot application is then used to deserialize the snapshot and set the player's game state according to it – similar to loading a savegame.

Similar to the service processing component, snapshot processing handles snapshot network messages (SNP), while the commands processing component manages command network messages (CMD). Since all primary network message types have been covered now, they can be summarized:

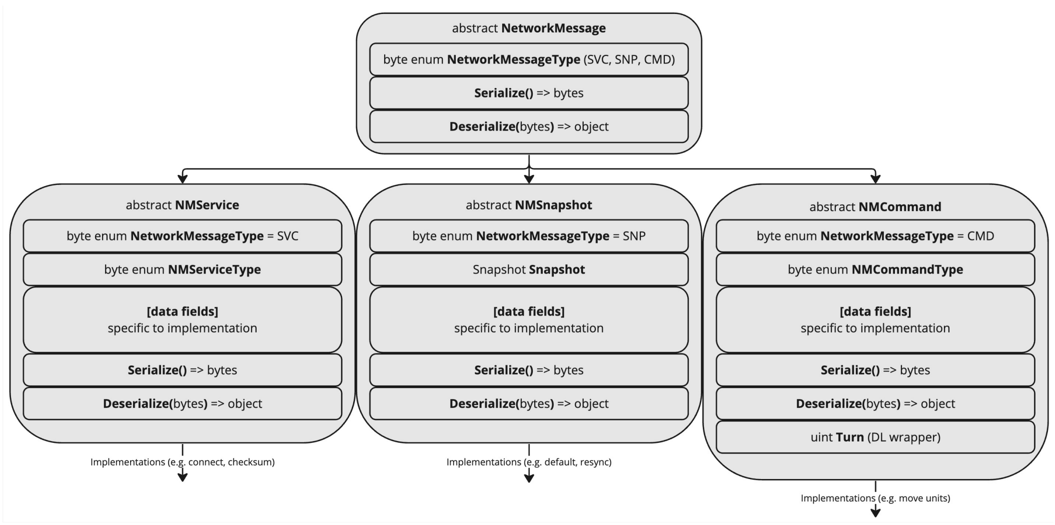

abstract NetworkMessage
byte enum NetworkMessageType (SVC, SNP, CMD)
Serialize() => bytes
Deserialize(bytes) => object
abstract NMService
byte enum NetworkMessageType = SVC
byte enum NMServiceType
[data fields] specific to implementation
Serialize() => bytes
Deserialize(bytes) => object
Implementations (e.g. connect, checksum)
abstract NMSnapshot
byte enum NetworkMessageType = SNP
Snapshot Snapshot
[data fields] specific to implementation
Serialize() => bytes
Deserialize(bytes) => object
Implementations (e.g. default, resync)
abstract NMCommand
byte enum NetworkMessageType = CMD
byte enum NMCommandType
[data fields] specific to implementation
Serialize() => bytes
Deserialize(bytes) => object
uint Turn (DL wrapper)
Implementations (e.g. move units)

Figure 5.3: Network message classes structure. Base classes are abstract. Arrows represent inheritance.

`NetworkMessage` is an abstract base class and must be extended by implementations so that they qualify for network transfer. `NetworkMessageType` is the header read by the network message processing in order to delegate the message to a suitable specialized processor (SVC, SNP, CMD). All messages must be (de)serializeable. The second layer in the inheritance tree of figure 5.3 are other specialized abstract base classes for implementations. They define the `NetworkMessageType` and have itself another type for the different implementations, that are based on them (e.g. `NMServiceType`). The data body or fields depend on the implementation, for example the checksum service message contains a checksum value. If deterministic lockstep (DL) is the primary synchronization system a wrapper around `NMCommand` is used, that adds a `Turn` field to define the command's execution turn.

Lockstep System

The lockstep system has own command and snapshot processing components. Usually, only the commands processor is required. But, because the presented model is capable of resynchronization after desyncs and supports joining a game after it has been started (hot-join), a snapshot processor tailored to the lockstep system is needed. This makes the system a lockstep-snapshot hybrid.

Command management is more intricate for the lockstep system compared to the snapshot system. Instead of immediately executing a received command, a synchronized execution is mandatory. Therefore, two command buffers are used. While the send buffer handles the timed transmission of commands, the ack buffer stores and executes received commands in the correct turn. In this implementation, the send buffer immediately transmits the command. So, no buffer would be required. However, alternative methods could store commands until the end of the turn and only then send them out, which would require a buffer (original implementation, that was changed to the current state).

The ack buffer is a ring buffer and illustrated in figure 5.4.

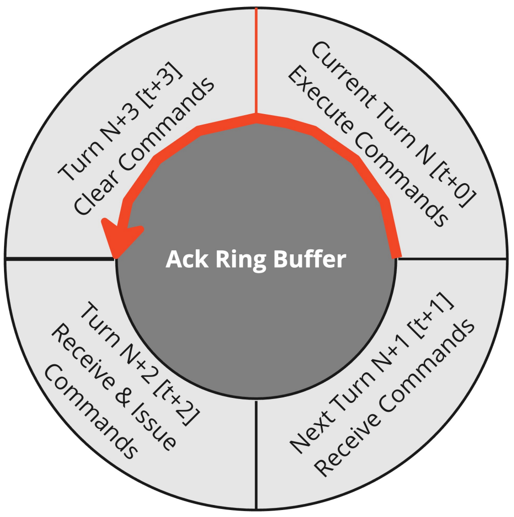


Figure 5.4: Ack ring buffer. N is the turn number and t is the ring index. Illustrated in red is the rotation direction of the ring (counter-clockwise).

The ring buffer contains four segments. In practice, segments are represented by a list with dynamic length (add and clear functions are required). Each segment is designated a turn N and collects all received commands scheduled for that turn by adding it to their list (turn buffers). Since there are only four segments that can be mapped to four turns at the same time, a ring index t is introduced. t is equal to the $\mathit{currentTurn} \mod \mathit{segmentCount}$, where $\mathit{segmentCount}$ is 4, and is used to access segments stored in an array data structure up to 3 turns ahead with the formula: $[t + (\mathit{turn} - \mathit{currentTurn}) \mod \mathit{segmentCount}]$. The ring is rotated counter-clockwise after every turn, which is done using the formula: $t = t + 1 \mod \mathit{segmentCount}$. When receiving a command it is added to the segment $[t + (\mathit{commandTurn} - \mathit{currentTurn}) \mod \mathit{segmentCount}]$. Every segment has a specific functionality based on a t offset. The segment at $[t + 0]$ (current turn) executes its commands at the start of the turn. Segment $[t + 1]$ and $[t + 2]$ both collect received commands, that will be executed in the next two turns. The commands issued in the current turn N are scheduled for turn $N + 2$, which means that commands issued by the host immediately go into the segment $[t + 2]$. Finally, segment $[t + 3]$ clears all collected commands on turn start for its next ring iteration.

At the core of the lockstep system is the time management component. This component enforces a timed execution of commands in lockstep. Additionally, it governs simulation progression with turns and ticks. So, if not all commands have been received and the next turn is not ready, the simulation, or tick progression, is paused. The lockstep timing of network messages is clarified in figure 5.5 in a typical game flow scenario. For clarity, the turn duration in the figure is 200ms compared to the used duration of 250ms in the actual implementation.

Figure 5.5 depicts an exemplary network message exchange between three game clients over time subdivided into turns, where the middle client is also the host. Client 1 has an RTT of 50ms, client 2 100ms. Host commands are omitted to declutter the figure, but, in practice, when the host issues a command it is broadcast to all other clients (acked), while the host immediately keeps the command in his ack buffer. In case a server is used, no commands are issued. There are three phases: the *pre-lobby phase*, e.g. menu navigation before joining a lobby; the *lobby phase*, where players are connected and lobby information is exchanged; the *game phase*, which is the actual game (game time). Arrows represent network messages and are categorized by color: cyan, message from client 1; blue, acked message of client 1 from the host; light purple, message from client 2; dark purple, acked message of client 2 from the host; black, message from the host (not complete). Each message is either a SVC (service message) or CMD (command message) – SNP (snapshot messages) are not included in this scenario.

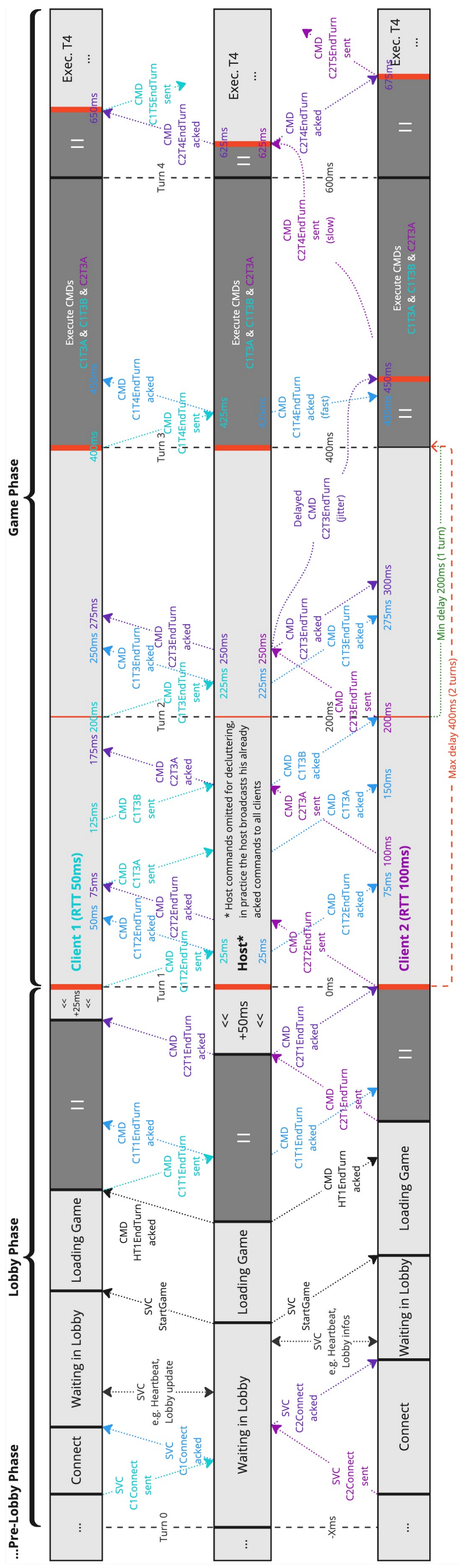


Figure 5.5: Lockstep network message timing.

Messages are named in the following scheme:
`[origin?: H(ost)|C(lient)1/2][turn?: T(urn)1..N][message function]`.
A "sent" message is transmitted from a client to the host, whereas "acked" messages are acknowledged and broadcasted by the host to clients.

The general commands distribution flow is: client sends a CMD to the host $\implies$ host adds the command to his ack buffer and acks the CMD (broadcasts the acknowledged command to all clients) $\implies$ each client receives the acked CMD and adds the command to their ack buffer. In the following, the figure is explained from left to right, respectively in the direction of time.

Once the host has created a lobby, the lobby is visible to other players and he waits for them to join. For WAN the UGS lobby service is used, while for LAN a custom implementation is needed. Both clients join the lobby with a SVC connect handshake, which is a connect request sent by a client and acknowledged by the host. Client 2 needs twice the time for connecting, since his RTT is 100ms compared to client 1's 50ms. After a client has successfully connected to the host, heartbeats and lobby updates are exchanged, e.g. player joined or lobby configuration changed. When all players are ready, the host initiates the game with another SVC message.

As soon as a client has loaded the game, the lockstep system becomes active and pauses the simulation until all commands have been received for the next turn – in this case turn 1. Central to this mechanism are end turn commands. They are sent at the end of a turn, but only if the next turn is ready, and contain the number of commands, that have been issued in the current turn. The next turn $N + 1$ is ready if:

1. the client received [client count] end turn commands scheduled for $N + 1$
2. every end turn's number of commands summed is equal to the number of received commands contained in the turn buffer for $N + 1$

With this mechanism clients can only get ahead one turn until they are forced to wait. This also prevents an ack buffer overflow. Deadlocks, where all clients wait for end turn commands and do not send them, since the next turn is not ready, are not possible as long as commands are scheduled for a turn later than the next one.

In order for the game to start, turn 1 must be ready. Therefore, to notify all other clients, that the game has loaded, an initial end turn CMD scheduled for turn 1 is sent (number of commands = 1). The end turn CMD is acked by the host according to the general distribution flow. The simulation is paused by the lockstep system until all end turn, or "game loaded", commands have been received.

At a certain absolute time, clients can be at slightly shifted game, turn and tick times due to network latency. As already mentioned, the turn time can only differ by one. However, since clients need to wait for slower ones, lockstep timing should balance itself inherently. In order to help the game begin synchronously, the start of turn 1 can be delayed artificially to account for RTTs. This can be done by sharing the maximum RTT of the slowest client in the network in the start game SVC message. Each client can then calculate the initial delay with $\frac{maxRTT-ownRTT}{2}$ (host has an $ownRTT$ of 0, minimum is 0).

The game phase starts with turn 1 and each client sends another end turn 0 CMD scheduled for turn 2. From now on, the lockstep system is working as intended. Real player commands are issued, sent and distributeed based on the general distribution flow. In the end of a turn an end turn command with the number of issued commands completes the turn. Both initial end turn CMD messages sent in turn 0 are required to synchronize game start and kick-off the lockstep system.

Given commands are scheduled for turn $N + 2$, a turn duration of 200ms has a minimum command issue to execution delay of 200ms (issued in the end of a turn), and a maximum delay of 400ms (issued in the beginning of a turn), while not taking into account possible simulation pauses.

In turn 3 and 4 the first player issued commands are executed. Also here, the effect of a delayed CMD, possibly due to jitter, is shown. The lockstep system handles the delay by first pausing simulation of client 3, only sending the end turn CMD of client 3 after the delayed CMD has been received and pausing the simulation of the other clients until client 3 is able to catch up.

The last component of the lockstep system is the synchronization guard. Since simulation determinism is challenging and game state desynchronization is possible, this component detects synchronization anomalies and manages the resynchronization process. In order to monitor synchronization each client generates a checksum of his game state every 20 turns, or approximately 5 seconds. The game state checksum (hash) algorithm in a real-time game context should be fast and must detect data permutation, e.g. wrong order in data structures that lead to diverging states. A reasonable collision safety must be given, whereas security is of low importance. Algorithm 1 was used and fulfills these criteria:

Algorithm 1: Game state checksum generation

```
Input: $gameState$: a serialized game state (byte array)
Output: the checksum hash of the game state (unsigned integer)

$checksum = 0$ ;
$i = 1$ ;
for $byte$ in $gameState$ do
    $checksum = checksum + byte \cdot i$ ; $i = i + 1$ ;
end
return $checksum$ ;
```

The host broadcasts his checksum after generation. When a client receives that checksum, both values are compared and in case they do not match a resync request SVC message is sent back to the host. Once the resync request has been received by the host, the resynchronization process is initiated:
Host clears the ack buffer, sets the turn to $N - 1$ and max. ticks (end of the last turn), generates a new random seed and broadcasts a specialized snapshot with additional data (turn $N - 1$ and the new random seed) $\Longrightarrow$ Clients receive the snapshot, apply it to their game state, clear their ack buffer, set their turn to $N - 1$ and max. ticks, initialize their random generator with the seed and send a resynced SVC message back to the host $\Longrightarrow$ Once, the host has received a resync SVC from every client, he broadcasts end turn CMD messages for each client, that are scheduled for the next turn, so that the simulation can continue.

During the whole resynchronization process, both own and incoming commands are ignored by the host. There can be a maximum command loss of two turns, but usually only commands issued in the current turn are lost. Also, the game will have a delay/lag based on the time the resynchronization process takes.

5.2.4 Hybrid Systems – Layer 8/9

Hybrid systems are in between layer eight and nine, since they are essential to networking, but are used by game logic even when the game is not networked. So, systems in the hybrid layer must work both in singleplayer as well as multiplayer while providing a unified interface.

This implementation uses a cross-platform deterministic random provider (PRNG), that is synchronized with seeds over the network in case of multiplayer, but can be used in singleplayer as well. The seed is set with the start game SVC message or resync snapshots.

The entity manager handles both static and dynamic game entities and interoperates with the NID management component to (de-)register entities and their IDs in multiplayer. The entity manager also executes the entities update loop. Entities are ordered by NID and updated in the same order to achieve a deterministic execution order, which is among others important for a deterministic random. Also, a part of a snapshot can be retrieved from the manager by iterating and serializing each entity, that is registered. Likewise, a snapshot can be applied by deserializing the entity data, which will set the state of existing entities, spawn new entities and destroy missing entities.

5.3 Additional topics

Dynamic turn duration

The turn duration can be dynamic and adapted to changing network conditions. This improves game flow and therefore player experience. Since the host knows the maximum RTT of the slowest client in the network, a suitable turn duration can be determined. A turn duration should be selected, that yields no end turn lags while input to action delay is minimized, e.g. twice the maximum RTT to account for jitter and occasional packet loss. According to [14], turn duration changes should be gradual and increase faster than they decrease. A turn duration change is communicated to clients with a SVC message, that contains the new duration and a turn $N + x$, where $x >= 2$, when the duration becomes active.

Savegames

A savegame is a snapshot, that contains a non-initial game state. They are created in order to continue the game at a later point. Savegames can be loaded and distributeed by the host in the lobby phase and are applied on every client at the beginning of the game phase.

Hot-join / Reconnect

Contrary to some literature [27], [33], with the implementation model of this work it is possible to support hot-join and reconnecting to a running game, for example, after a player lost connection. This is possible, because hot-joining is the same process as resynchronizing game states after a desync has been detected by the synchronization guard component. The only additional process is the connection establishment before the snapshot can be exchanged.

Host migration

In a client-host topology the game can not continue if the host lost connection. Host migration can be used to be able to continue the game without creating a savegame and then hosting a new lobby with that savegame. However, migrating the host to another client requires either an external rendezvous server or other precautionary measures, because the remaining clients are not connected to each other.

A rendezvous server can be contacted by the clients passing, for example, a lobby ID, so that the server knows which clients are associated. When all clients contacted the server the most suitable host can be determined based on communicated client hardware and network capabilities. After a new host was designated the remaining clients connect to the host and the game can continue.

An alternative precautionary measure could be, that the connection information to every client is shared by the host. This way all clients can connect to the new host immediately. The selection of the host must work deterministically without communication between the clients, e.g. based on the client NIDs.

Deterministic lockstep and snapshot interpolation hybrid

Based on the presented implementation architecture model (figure 5.2), it is conceivable that both synchronization methods could be combined, so that, based on the game or network conditions, a dynamic switch between them can be performed. This way the disadvantages of snapshot interpolation could be negated by the advantages of deterministic lockstep and vice-versa. For example, snapshot interpolation is used when many players participate in the current game scene, but deterministic lockstep for scenes with many game entities or when bandwidth is limited. Another scenario where switching methods might be reasonable is when a game has both zoomed-out and zoomed-in phases. And, during the zoomed-in phase a lower-latency method is required, while the game's meta-state is still synchronized by deterministic lockstep.

Switching methods from deterministic lockstep to snapshot interpolation is straight-forward. The lockstep system is disabled and stops governing simulation with turns, while the snapshot system is enabled and from now on only snapshots are distributeed by the host. This is a fluid transition with no delay. Switching from snapshot interpolation to deterministic lockstep, however, comes with a delay, since, after disabling the synchronization and enabling the lockstep system, the same resynchronization process of the synchronization guard component is used.

Limitations

There are general and implementation related limitations, that may influence evaluation results:

- network data transmission speed (close to speed of light)
- UGS lobby service (max. players 100)
- UGS relay service (max. players 100)
- bandwidth (e.g. 250 Mbit/s)
- router hardware
- client hardware
- NID uniqueness (alphanumeric 28 byte long IDs, $36^{28} \approx 1.55 \cdot 10^{44}$ IDs)

- UTP
  - send & receive network packet queue length per update schedule (1024)
  - reliable pipeline stage window size per connection (max. 32 in-flight network packets) [**A**]
  - payload size per network packet (max. 44kB) [**B**]

Due to limitation [**A**] a custom system was implemented on top of UTP to package multiple smaller network messages, that are being transmitted in the same update schedule, into one larger network packet. This reduces the number of in-flight packets and with that the frequency of packets getting requeued for transmission (delayed). The maximum package size is approximately 32kB.

Limitation [**B**] is a problem for snapshot transmission and large packaged network messages. Therefore, an additional fragmentation system was built on top of the existing UTP fragmentation pipeline stage, that supports a network message payload size of up to approximately 16MB.

Chapter 6

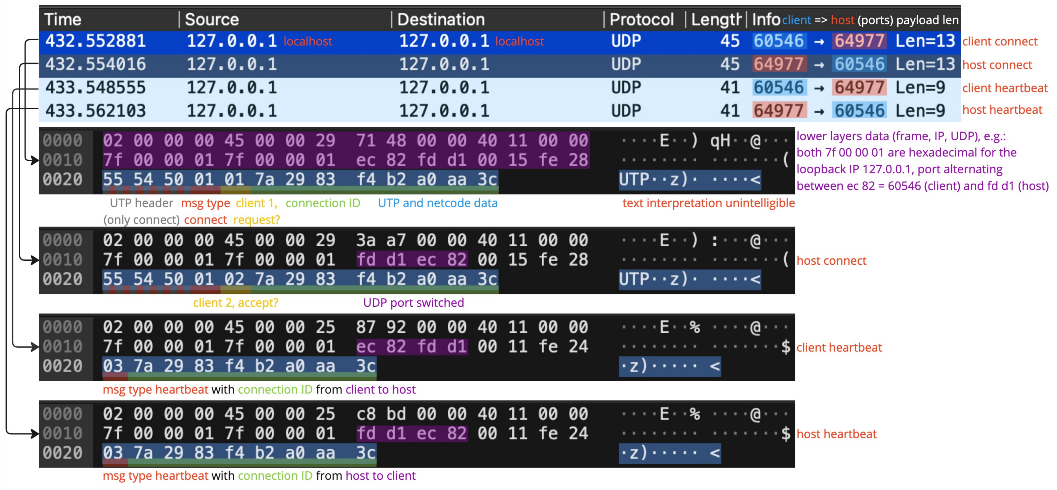
Evaluation

Based on the implementation of chapter 5 the scientific question of this thesis is now evaluated and discussed. Before that, the underlying UTP networking framework used by the implementation is briefly analyzed with regards to its network packet structure. Then, the evaluation setup is described. Results are evaluated and discussed and finally a conclusion to the scientific question is formulated. Additionally, the results are compared to related work.

6.1 UTP packet evaluation

To this date, the documentation of UTP does not include detailed information about the exact data down to the byte level, that is transmitted. Therefore, Wireshark [34] was used to deconstruct and analyze UTP packets of a localhost loopback connection. This means, that the packets never leave the computer, because origin and destination are the same (IP is `127.0.0.1`). Since only the packet data is of interest and not the transmission to another computer, this setup was chosen. The following observations were made.

Figure 6.1 shows screenshots of the packet analysis tool Wireshark. The first screenshot displays four captured packets with a timestamp, source/origin and destination IP, protocol, length/packet size, and further information, which in this case is the UDP source to destination port and payload size (len). Source and destination IP is always `127.0.0.1` for localhost and the protocol is UDP. The length varies based on the payload size. Two ports can be identified: the client port with `60546` and the host port with `64977`. Each packet's data is inspected in more detail in the following four screenshots from top to bottom.



```
Time        | Source              | Destination         | Protocol | Length | Info client => host (ports) payload len
432.552881   127.0.0.1 localhost   127.0.0.1 localhost   UDP        45      60546 → 64977 Len=13
432.554016   127.0.0.1             127.0.0.1             UDP        45      64977 → 60546 Len=13
433.548555   127.0.0.1             127.0.0.1             UDP        41      60546 → 64977 Len=9
433.562103   127.0.0.1             127.0.0.1             UDP        41      64977 → 60546 Len=9

0000  02 00 00 00 45 00 00 29  71 48 00 00 40 11 00 00   ····E··) qH··@···
0010  7f 00 00 01 7f 00 00 01  ec 82 fd d1 00 15 fe 28   ········ ·······(
0020  55 54 50 01 01 7a 29 83  f4 b2 a0 aa 3c            UTP··z)· ····<

0000  02 00 00 00 45 00 00 29  3a a7 00 00 40 11 00 00   ····E··) :···@···
0010  7f 00 00 01 7f 00 00 01  fd d1 ec 82 00 15 fe 28   ········ ·······(
0020  55 54 50 01 02 7a 29 83  f4 b2 a0 aa 3c            UTP··z)· ····<

0000  02 00 00 00 45 00 00 25  87 92 00 00 40 11 00 00   ····E··% ····@···
0010  7f 00 00 01 7f 00 00 01  ec 82 fd d1 00 11 fe 24   ········ ·······$
0020  03 7a 29 83 f4 b2 a0 aa  3c                        ·z)····· <

0000  02 00 00 00 45 00 00 25  c8 bd 00 00 40 11 00 00   ····E··% ····@···
0010  7f 00 00 01 7f 00 00 01  fd d1 ec 82 00 11 fe 24   ········ ·······$
0020  03 7a 29 83 f4 b2 a0 aa  3c                        ·z)····· <
```

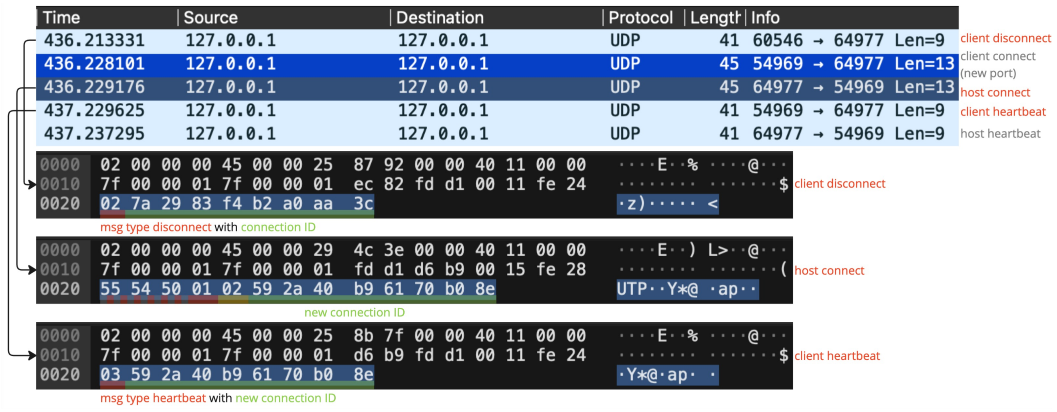
client connect
host connect
client heartbeat
host heartbeat
lower layers data (frame, IP, UDP), e.g.: both 7f 00 00 01 are hexadecimal for the loopback IP 127.0.0.1, port alternating between ec 82 = 60546 (client) and fd d1 (host)
UTP header (only connect)
msg type connect
client 1, request?
connection ID
UTP and netcode data
text interpretation unintelligible
host connect
client 2, accept?
UDP port switched
client heartbeat
msg type heartbeat with connection ID from client to host
host heartbeat
msg type heartbeat with connection ID from host to client

Figure 6.1: UTP packet deconstruction: connection process and heartbeats.

The first packet is a client connect network message. The packet data is shown both in hexadecimal (left) and textual representation (right). However, the textual interpretation is mostly unintelligible and therefore ignored. The purple highlighted bytes can be attributed to lower layers data (frame, packet, datagram). For instance, both `7f 00 00 01` bytes are hexadecimal for the localhost loopback IP `127.0.0.1`. Or, it can be identified, that the UDP port alternates between the client's `ec 82` (60546) and host's `fd d1` (64977). The blue highlighted bytes contain the UTP and netcode application data (payload). The first three bytes are a "UTP" header in American Standard Code for Information Interchange (ASCII) text encoding, followed by the UTP message type byte. `01` in combination with the "UTP" header prefix is a connection establishment message. Byte five cannot be clearly identified, but is probably the differentiation between client connection request and host connection accept. The remaining eight bytes represent a connection ID of UTP.

The second packet data shows a UDP port switch in purple. Now the host sends a message to the client, which is the accept answer to the client's connection request. Every payload byte is the same, except for byte five, which is now `02` probably meaning "connection accpeted".

The last two screenshots show a connection heartbeat, that verifies a connection is still alive. The heartbeat interval can be configured during the UTP setup. First, the client's and then the host's heartbeat are shown. The heartbeat payload only includes the message type `03` and the connection ID.

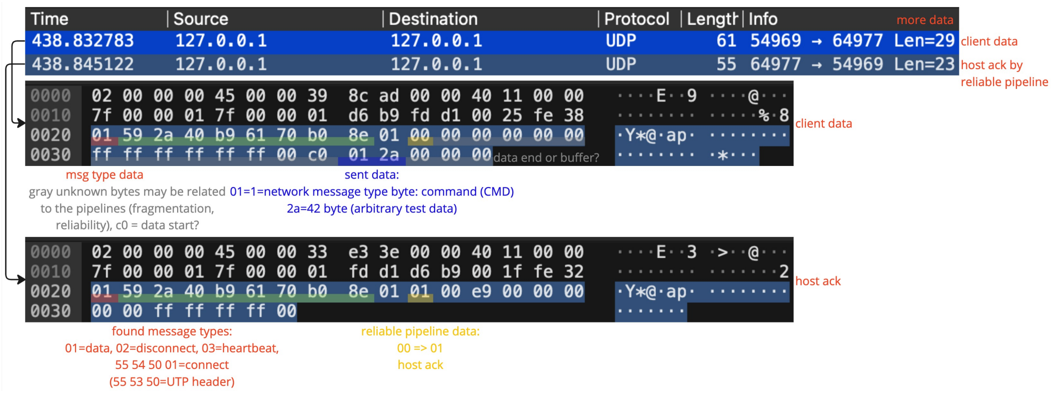


```
Time          Source      Destination   Protocol  Length  Info
436.213331    127.0.0.1   127.0.0.1     UDP       41  60546 → 64977 Len=9
436.228101    127.0.0.1   127.0.0.1     UDP       45  54969 → 64977 Len=13
436.229176    127.0.0.1   127.0.0.1     UDP       45  64977 → 54969 Len=13
437.229625    127.0.0.1   127.0.0.1     UDP       41  54969 → 64977 Len=9
437.237295    127.0.0.1   127.0.0.1     UDP       41  64977 → 54969 Len=9

0000  02 00 00 00 45 00 00 25  87 92 00 00 40 11 00 00   ····E··%  ····@···
0010  7f 00 00 01 7f 00 00 01  ec 82 fd d1 00 11 fe 24   ········  ·······$
0020  02 7a 29 83 f4 b2 a0 aa  3c                        ·z)·····  <

0000  02 00 00 00 45 00 00 29  4c 3e 00 00 40 11 00 00   ····E··)  L>··@···
0010  7f 00 00 01 7f 00 00 01  fd d1 d6 b9 00 15 fe 28   ········  ·······(
0020  55 54 50 01 02 59 2a 40  b9 61 70 b0 8e            UTP··Y*@  ·ap··

0000  02 00 00 00 45 00 00 25  8b 7f 00 00 40 11 00 00   ····E··%  ····@···
0010  7f 00 00 01 7f 00 00 01  d6 b9 fd d1 00 11 fe 24   ········  ·······$
0020  03 59 2a 40 b9 61 70 b0  8e                        ·Y*@·ap·  ·
```

Figure 6.2: UTP packet deconstruction: reconnection process.

Figure 6.2 depicts the UTP reconnection process. First, the client sends a disconnect message packet, followed by a new connect message. The client also changes the port in the process from `60546` to `54969`. After that, the host again accepts the connection request and two heartbeats are exchanged. The client disconnect, host connect and client heartbeat packets are now analyzed.

Similar to a heartbeat message, the disconnect message contains the message type byte `02` and the old connection ID. The host connect packet now uses a new connection ID, that is also used in the client's heartbeat and all following messages.



```
Time          Source      Destination   Protocol  Length  Info
438.832783    127.0.0.1   127.0.0.1     UDP       61  54969 → 64977 Len=29
438.845122    127.0.0.1   127.0.0.1     UDP       55  64977 → 54969 Len=23

0000  02 00 00 00 45 00 00 39  8c ad 00 00 40 11 00 00   ····E··9  ····@···
0010  7f 00 00 01 7f 00 00 01  d6 b9 fd d1 00 25 fe 38   ········  ·····%·8
0020  01 59 2a 40 b9 61 70 b0  8e 01 00 00 00 00 00 00   ·Y*@·ap·  ········
0030  ff ff ff ff ff ff 00 c0  01 2a 00 00 00            ········  ·*···

0000  02 00 00 00 45 00 00 33  e3 3e 00 00 40 11 00 00   ····E··3  ·>··@···
0010  7f 00 00 01 7f 00 00 01  fd d1 d6 b9 00 1f fe 32   ········  ·······2
0020  01 59 2a 40 b9 61 70 b0  8e 01 01 00 e9 00 00 00   ·Y*@·ap·  ········
0030  00 00 ff ff ff ff 00                               ·······
```

more data
client data
host ack by reliable pipeline
client data
data end or buffer?
msg type data
gray unknown bytes may be related to the pipelines (fragmentation, reliability), c0 = data start?
sent data:
01=1=network message type byte: command (CMD)
2a=42 byte (arbitrary test data)
host ack
found message types:
01=data, 02=disconnect, 03=heartbeat,
55 54 50 01=connect
(55 53 50=UTP header)
reliable pipeline data:
00 => 01
host ack

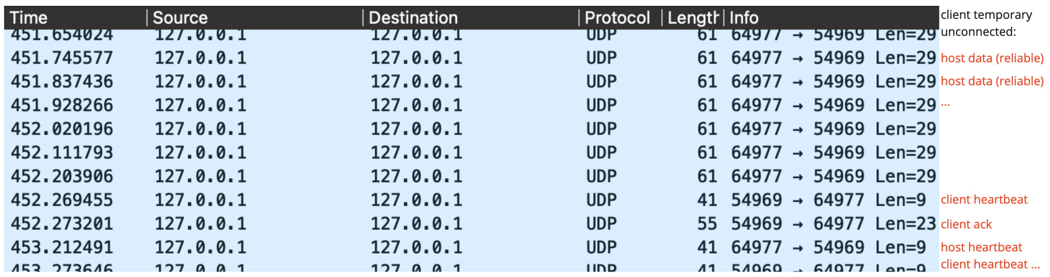
Figure 6.3: UTP packet deconstruction: sent data.

Figure 6.3 shows data exchange using the reliable pipeline. Additional payload is now transmitted, hence the len information is larger compared to the previous packets. After the client has sent its data, the host also acknowledges the packet to the client due to the reliable pipeline.

Data uses the message type byte `01` without the "UTP" header, which is the differentiation to the connect message. It is assumed that pipeline related data follows highlighted in gray (fragmentation, reliability). `c0` could indicate the start of the actual data sent. The actual netcode data is highlighted in dark blue. This part is under full control of the netcode implementation on top of UTP and therefore known. `01` is the netcode's network message type, for example a command (CMD) and `2a` is arbitrary test data, in this case a byte for the number 42. The remaining three bytes `00 00 00` again are unknown and could be a buffer or mark the end of data.

The last screenshot of figure 6.3 is the host's acknowledgment of the client's data packet. This only happens if the reliable pipeline is used. The same message type and connection ID is used, followed by again unknown presumably pipeline data. One byte that can be identified is highlighted in yellow. While this byte was `00` in the client's packet, the host changes this byte to `01`. This could be the acknowledgment byte change by the reliable pipeline.

In summary, there are four found UTP message types: `01` for data, `02` for disconnection, `03` for heartbeats, and `01` in combination with the "UTP" header prefix for connection (`55 53 50 01`).



```
Time          Source      Destination   Protocol  Length  Info                              client temporary
451.654024    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29              unconnected:
451.745577    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29              host data (reliable)
451.837436    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29              host data (reliable)
451.928266    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29              ...
452.020196    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29
452.111793    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29
452.203906    127.0.0.1   127.0.0.1     UDP       61      64977 → 54969 Len=29
452.269455    127.0.0.1   127.0.0.1     UDP       41      54969 → 64977 Len=9               client heartbeat
452.273201    127.0.0.1   127.0.0.1     UDP       55      54969 → 64977 Len=23              client ack
453.212491    127.0.0.1   127.0.0.1     UDP       41      64977 → 54969 Len=9               host heartbeat
453.273646    127.0.0.1   127.0.0.1     UDP       41      54969 → 64977 Len=9               client heartbeat ...
```

Figure 6.4: UTP packet deconstruction: reliable pipeline.

The last inspected figure 6.4 clarifies the functionality of the reliable pipeline. While the client is temporary unconnected, the host sends multiple data packets until finally the client is connected again, shown by the heartbeat, and acks the host's data packet.

One final note about packets in a relay-based WAN scenario. Packets were analyzed in this scenario as well, but, except for the finding, that packet size generally increases, deconstruction turned out to be difficult, presumably due to encryption.

6.2 Evaluation setup

As a reminder, the scientific question is "What are the limitations in terms of vertical (entity count) and horizontal (player number) scaling of a networked multiplayer game and how does deterministic lockstep compare to a snapshot method?". In order to quantitatively evaluate this question, the implementation provides a command to initiate a 20 second evaluation of a certain scaling configuration. To determine vertical limits this command distributes the desired entity count across all clients evenly. The command also instructs the statistics provider component of each client to start writing statistics to a file for later analysis.

To scale horizontally a total of eight computers were available, where each of them could run five instances of the evaluation build. This results in a maximum of 40 simulated clients.

For comparison of deterministic lockstep to a snapshot method, snapshot interpolation was selected due to the reasons mentioned in chapter "Introduction", section 1.1, and chapter "Implementation", section 5.3.

6.2.1 Synchronized data

The game state is made up of a set number of entities. An entity has the following synchronized data: `ushort` *type* (2 bytes), `NID` *id* (32 byte string), `Vector2D` *position* (two 4 byte floats = 8 bytes), `NID` *playerId* (32 bytes), `Vector2DInt` *coordinates* (8 bytes), `byte` *color* (1 byte). So, in total one entity uses 83 bytes. A UTP network packet has a 27 byte overhead. To synchronize one entity the packet would therefore have 110 bytes. If multiple entities are synchronized in a snapshot they are packaged together, so that the overhead is only applied once (fragmentation disregarded). Sending a 1024 entities snapshot results in a packet size of about 85kB, 4096 entities use approximately 340kB, and 16,000 entities require 1.3MB.

6.2.2 Hardware and environment

Inhomogeneous computer hardware was used in order to replicate the diverse cross-platform environment of the real world as best as possible. Seven of the eight computers were Windows 10-based (version 22H2) Acer Aspire V Nitro Black Edition Gaming Notebooks. They are equipped with an Intel Core i7-6700HQ CPU @ 2.6 GHz, an NVIDIA GeForce GTX 960M GPU, 16GB RAM, an SSD hard-drive, a full HD 60 Hz display, and a Qualcomm Atheros QCA61x4A wireless network adapter, that supports IEEE 802.11ac with a maximum speed of 867 MBit/s [35], [36].

A macOS-based (version Sonoma 14.0) 2021 16" Apple MacBook Pro was used as the host and for additional 4 clients. It is equipped with an M1 Max CPU/GPU, 32GB RAM, an SSD hard-drive, a 3456x2234 120 Hz display, and a wireless network adapter supporting IEEE 802.11ax and 802.11ac with a maximum speed of 866 MBit/s [37], [38], [39], [36].

The computers were placed next to each other and connected to the router wirelessly. Each computer had a distance of about 5 meters to the router. The router is a Vodafone Station supporting IEEE 802.11ax and 802.11ac at a maximum data rate of 1 GBit/s [40]. The internet provider was Vodafone as well with a Vodafone Cable 250 MBit/s downlink and 40 MBit/s uplink contract.

The evaluation took place in Karlsruhe, Germany on the 17th February 2024 from about 8 to 11 pm. Prior to the evaluation the actual internet speeds were tested. On the Windows computers an average download speed of 255.5 MBit/s and upload speed of 47.2 MBit/s were recorded. On Mac a download speed of 253.9 MBit/s and upload speed of 35.4 MBit/s was reached.

The UGS relay server was reported to be "europe-west4", which should be located in the Netherlands according to [41].

6.2.3 Metrics

The statistics provider component of each client collects and writes certain metrics into a comma-separated values (CSV) file every second for 20 seconds, which results in about 20 samples per evaluation configuration per client. The following metrics are considered and analyzed later:

Acronym	**Metric**	**Unit**	**Description**
RelTime/Time	Relative time	Seconds	The time a sample was taken relative to the starting time of the evaluation configuration.
FpsSmoothed	Smoothed FPS	FPS/Hz	The display refresh rate of the game with gradual changes (smoothed).
MaxAvgRtt	Maximum average RTT	Seconds	The maximum average RTT. For clients there is only one RTT to the host, which is the maximum; for the host the highest RTT to a client is used. Average means the RTT is averaged over many measurement, but, because this metric is collected from UTP and not documented, it is not defined how many or how long the time interval is.

Acronym	**Metric**	**Unit**	**Description**
DeltaPacketsTx	ΔPackets transmitted	Packets	The number of network packets, that have been transmitted since the last sampling.
DeltaPacketsRx	ΔPackets received	Packets	The number of network packets, that have been received since the last sampling.
DeltaPacketsTxRe	ΔPackets re-transmitted	Packets	The number of network packets, that have been re-transmitted by the reliable pipeline since the last sampling.
DeltaPacketsRxRe	Δduplicate packets received	Packets	The number of network packets, that have been received again by the reliable pipeline since the last sampling.
DeltaPacketsLost	ΔPackets lost	Packets	The number of network packets, that have been lost since the last sampling.
DeltaBytesTx	ΔBytes transmitted	Bytes	The number of bytes, that have been transmitted since the last sampling.
DeltaBytesRx	ΔBytes received	Bytes	The number of bytes, that have been received since the last sampling.
DeltaBandwidth	ΔBandwidth	Bytes	The number of bytes, that have been transmitted and received since the last sampling.
DeltaTicksLagged	ΔTicks lagged	Ticks	The number of ticks the simulation could not progress since the last sampling.
DeltaDesyncs	ΔDesyncs	Number	The number of desynchronizations, that have been detected since the last sampling.

Table 6.1: A tabular summary of the evaluation metrics.

6.2.4 Process

After setting up the evaluation environment, first, the hardware capabilities of both computer types were tested by determining the maximum entity count and reasonable instances per computer for a playable game. A playable game should have at least 30 frames per second (FPS). It turned out, that the Windows machines running five instances could handle up to 16,000 entities until the average FPS dropped below 30. On Mac, entity counts of up to 32,000 were possible. This test was performed without the multiplayer synchronization functionality in order to only find the CPU processing and GPU rendering based entity and instance count limits.

For the method evaluation the following parameters were used:

- 8 automated player commands per second per client (more than realistic)
- deterministic lockstep:
  - fixed turn duration of 250ms, no dynamic turn durations
  - 15 ticks per turn (one tick is 16.666ms or 60 FPS)
  - synchronization guard checks every 5 seconds (4 per 20 second evaluation)
- snapshot interpolation:
  - 1 second snapshot distribution intervals

The 1 second snapshot distribution interval is not realistic and should be 100ms or less for a regular game, because the input to action delay would be too high for clients. However, since the snapshot system is not optimized in this implementation, e.g. no snapshot compression is performed, a 1 second interval is reasonable. The DeltaBandwidth metric should be about the same for a large snapshot sent every second compared to 10 small in size snapshots sent every 100ms based on the optimizations evaluated in [26].

The following scaling configurations each were evaluated for 20 seconds:

- With deterministic lockstep, snapshot interpolation:
  - In a LAN, WAN:
    - For 10, 20, 40 players:
      - Evaluate 1024, 4096, 8192, 16,000 entities.

Other configurations were performed irregularly, e.g. 8 players or 2048 entities. A certain configuration is named in the following scheme: `[dl|sl]-[lan|wan]-p[player count]-e[entity count]`. `dl` stands for deterministic lockstep and `sl` for snapshot interpolation.

Additionally, the deterministic lockstep resynchronization process, and the effects of high latency (200ms), jitter (100ms) and packet loss (10%) of incoming and outgoing packets on both methods were tested. In the naming scheme these tests can be identified by a `-resync`, `-delay200`, `-jitter100` or `-loss10` suffix.

6.3 Discussion of results

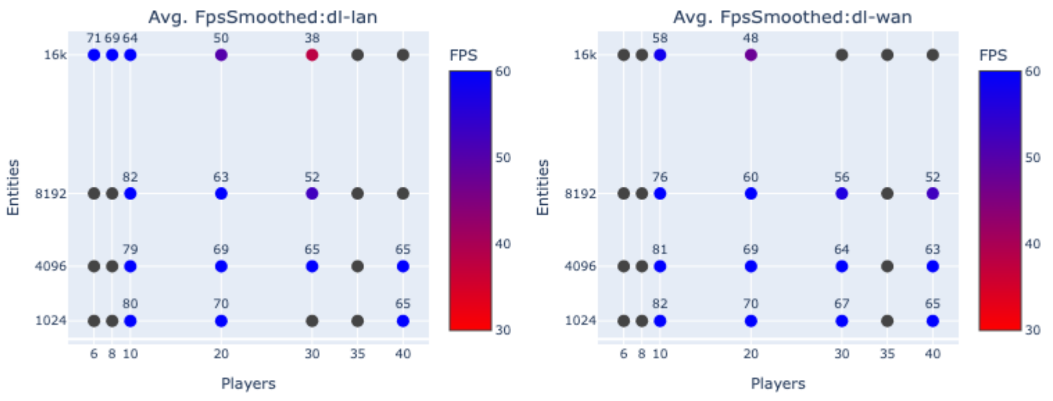
Every metric is evaluated to discuss vertical and horizontal limits in LAN and WAN, first for deterministic lockstep and then compared to snapshot interpolation. After that, resyncs (deterministic lockstep only) and special network condition limits are evaluated. The aggregated limitations are discussed and compared to related work in the last section of this chapter, section 6.4.

6.3.1 Metrics

For the evaluation of the metrics a scatter plot is used, that should convey a general overview of limitations (overview graph). Each data point of this plot represents the average value of the metric over all sampling points of every client (about 20 samples per client). Besides the value shown in text above the point, the data points are also colored based on the legend to the right (z value). The x-axis shows the horizontal scaling configurations (players) and the y-axis represents the vertical scaling steps (entities). This figure type also shows possible entity-player limit correlations, if there are any. Gray data points indicate no data for that scaling configuration. This can happen in two cases: the configuration was not tested since more complex (higher entity and/or player counts) configurations were possible and evaluated; or the configuration was not possible due to a game crash, for example, and no data could be recorded. Therefore gray points at outer edges already show possible limitations. The title of such figures is in the scheme `[Avg.|Med.] [metric]:[dl|sl]-[lan|wan]`, where `Avg.` indicates that the average over all sample points is used, whereas with `Med.` the median is used.

Some overview graphs for certain metrics might not yield new insights, but nevertheless are analyzed for completeness. Since these graphs only show general tendencies, rather than detailed information, some findings are further investigated in the 6.3.4 “Further investigations” section. For structural reasons this section is after all metric overview graphs, but to stay in context it might be advantageous to directly read a further investigation when it is first mentioned in the following text (mentions are clickable). A first mention is written as [**investigation name**], whereas subsequent mentions are surrounded by parenthesis ([investigation name]).

FPS



(a) lan (b) wan

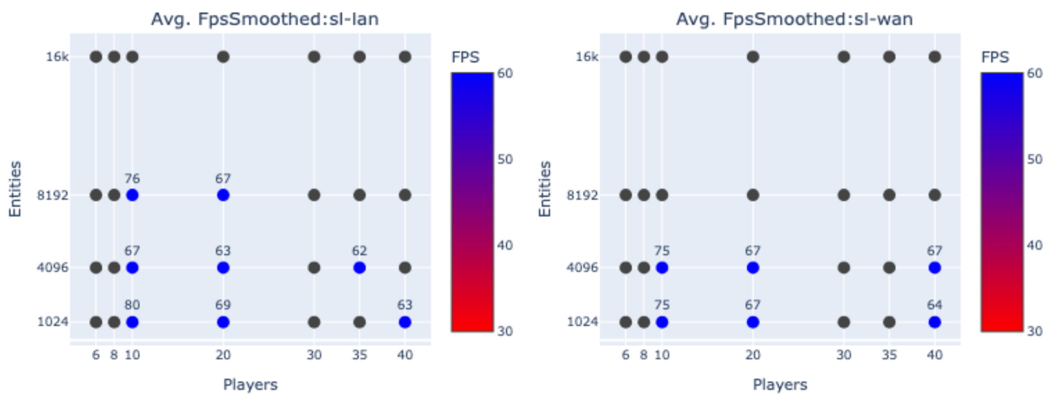
Figure 6.5: Overview graph: FpsSmoothed:dl

Figure 6.5: In LAN FPS decrease when scaling vertically and horizontally. This makes sense vertically, since there are more entities to compute and render for CPU and GPU. Horizontally, however, it is not as clear and therefore investigated further [**FPS1**]. Configs `p6/8/10-e16000` achieve an average FPS of 60+. Potential higher limits are investigated in [**FPS2**]. `p30-e16000` still has acceptable FPS, while `p40-e8192+` is not possible anymore [**FPS3**]. One can see, that entity and player count limitations correlate: a high entity count is possible with few players, and a high player count is only possible with few entities (negative correlation, or dependent limitations). The vertical limit is 16k+, and horizontal limit is 40(+) (more clients were not available, but it seems to be a limit ([FPS3])).

The figure for WAN shows similar results to LAN. There are slightly less FPS in general [**FPS4**]. However, config `p30-e16000` is not possible anymore as it was already barely possible in LAN. WAN is more demanding for the CPU ([FPS4]). On the other hand, `p40-4096` is possible compared to lan with an acceptable average FPS [**FPS5**]. But in general, similar limits to lan are found.

Comparison to snapshot interpolation

Figure 6.6: In lan FPS in general are similar to deterministic lockstep (dl) and again they decrease when scaling vertically and horizontally. But, it seems like snapshot interpolation (sl) achieves either good performance over 60 FPS or does not work at all (gray dots). This might be due to technical problems or the necessity of



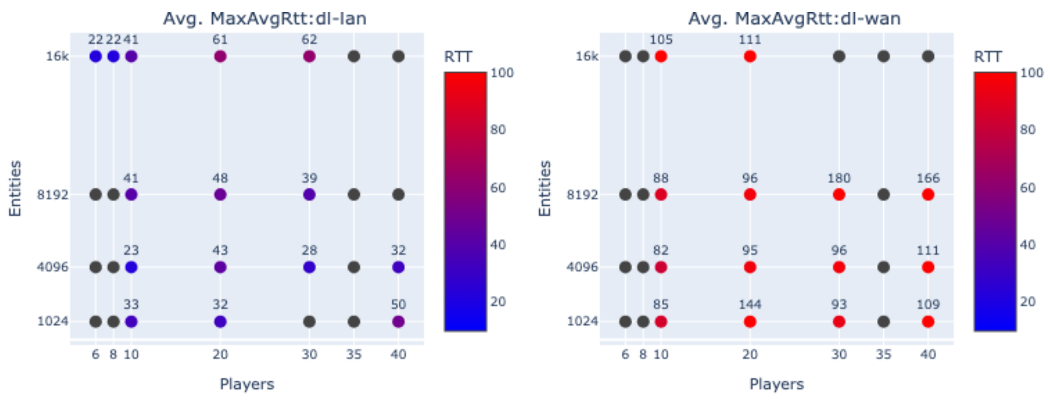
(a) lan (b) wan

Figure 6.6: Overview graph: FpsSmoothed:sl

snapshot optimization, because it seemed the transmit queue was full when the snapshot got too large and it was heavily fragmented by the custom fragmentation implementation. Thus, a vertical limit of 8192 (less compared to dl) and a horizontal limit of 40(+) (same) can be identified.

In wan there is an even lower vertical limit of 4096 and the same player limit, which looks like an independent limitation of max. 4096 entities and 40 players, regardless of the respective other limitation.

RTT



(a) lan (b) wan

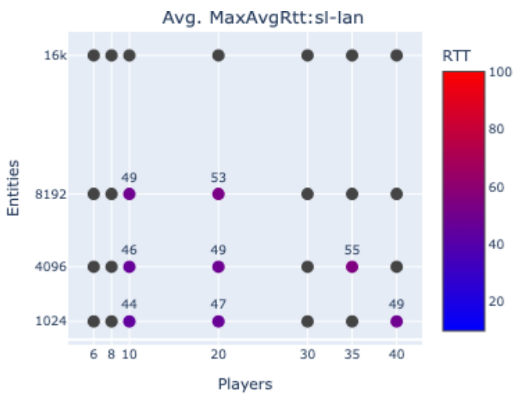
Figure 6.7: Overview graph: MaxAvgRtt:dl

Figure 6.7: In lan there seems to be a tendency of increasing RTTs with entity and player count (not as clear as with FPS), see [**RTT1**]. Acceptable RTTs for every config can be observed. `p20/30-e16000` seem to show an approaching limit [**RTT2**]. Again, there is no data available for `p40-e8192+` ([FPS3])). In summary,

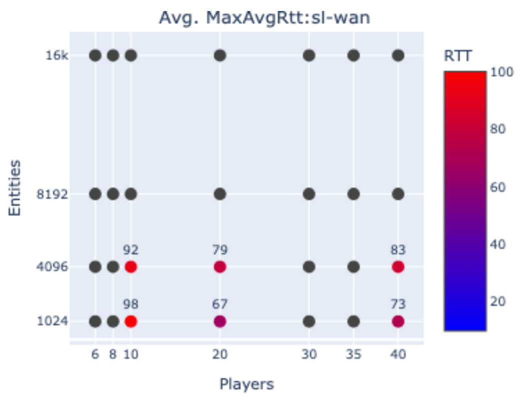
no limits are displayed, but indicated starting with `p30-e16000`.

In wan there is the same increasing tendency like in lan ([RTT1])). Reasonably, a strong RTT increase compared to lan can be seen, since the packets now have to travel to a relay server and back. Therefore, the RTT is in most configs above 100ms, which is not ideal for fast-paced games. Here, again data for `p40-e8192` can be found, but not for `p30-e16000` similar to the FPS evaluation. Similar limits to lan are recognized.

Comparison to snapshot interpolation



(a) lan

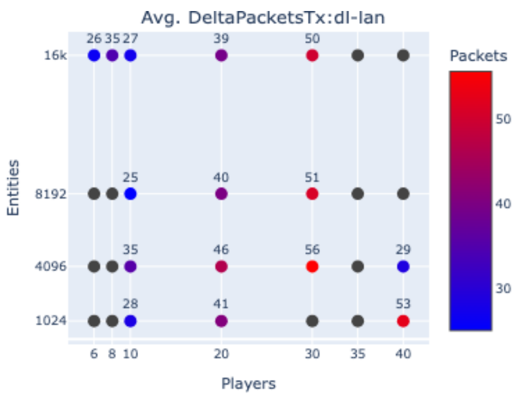


(b) wan

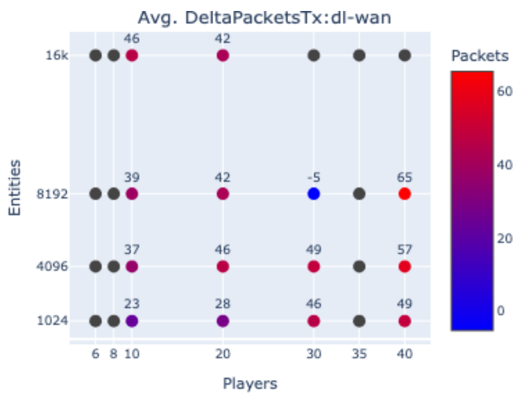
Figure 6.8: Overview graph: MaxAvgRtt:sl

Figure 6.8: No new insights for lan can be found (similar to dl). In wan the RTT seems to be lower compared to dl wan, but this could be a temporary effect.

Packets



(a) lan

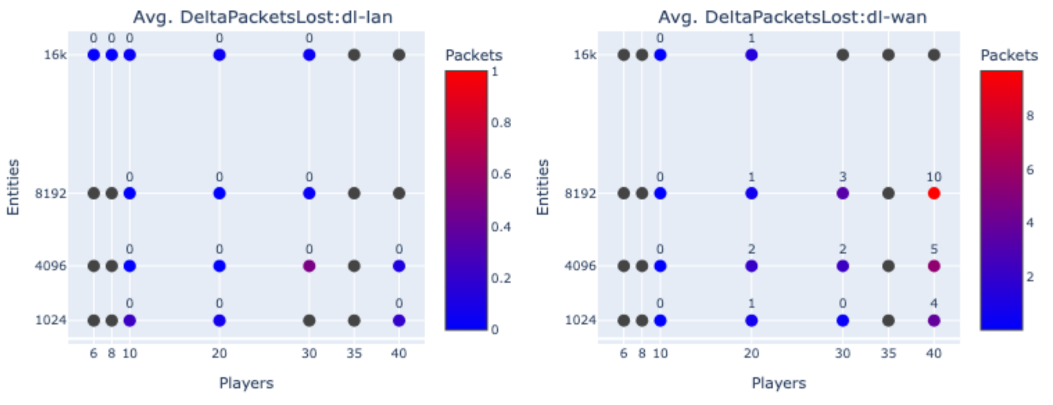


(b) wan

Figure 6.9: Overview graph: DeltaPacketsTx:dl

Figure 6.9: In lan the number of packets sent scales well vertically, since it is not directly dependent on the entity count, but only player commands, which is constant. With increasing players the packet count increases as well due to additional packets being transmitted from and to new players. However, the number of packets should scale linearly based on the player count, which is not the case and investigated in [**Packets1**]. Another investigation is about the seemingly outlying config `p40-e4096` [**Packets2**]. In conclusion, no new limits are indicated.

When looking at the corresponding wan figure, config `p30-e8192` catches attention with a packet count of -5 [**Packets3**]. Generally, more packets are sent due to packet loss being more frequent in wan, which leads to packet re-transmissions ([FPS4])).



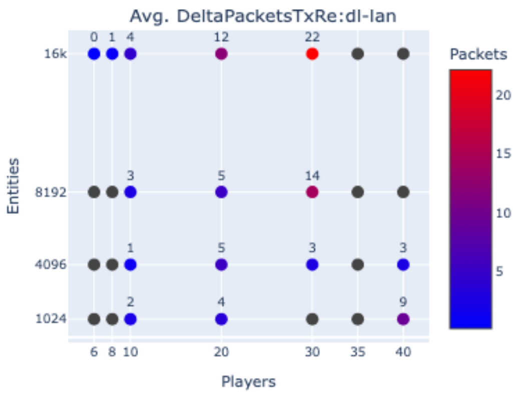
(a) lan (b) wan

Figure 6.10: Overview graph: DeltaPacketsLost:dl

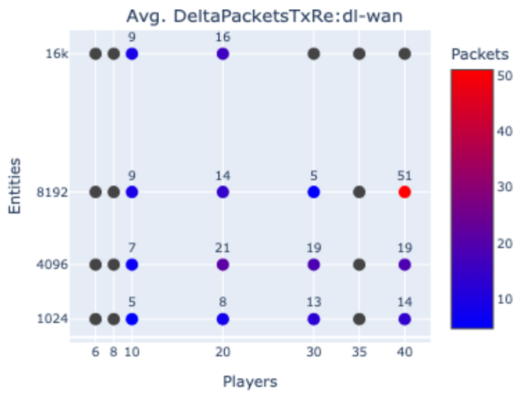
Figure 6.10: There is almost no packet loss in lan (only `p-30-e4096` has a rounded packet loss of 0, but based on the color the loss was probably about 0.4). More frequent packet loss can be observed in wan: up to an average of ten packets per second with `p40-e8192`. This fact indicates problems with that particular config ([FPS5])). The limit at `p-40-e4096` is similar to lan. No vertical limits are indicated.

Figure 6.11: Generally, more packets are being re-transmitted in wan, which comes down to the already identified higher packet loss, and presumably also a higher RTT for acks. But, there are some packets re-transmitted in lan as well. Especially `p30-e16000` could indicate a lan limit [**Packets4**]. In wan `p40-e8192` again is problematic.

Figure 6.12: Compared to dl, sl has a much higher packet transmission count due to snapshot data being larger than commands data. Also, the packet count increases not only with players but also with entities using snapshot interpolation, because the snapshot gets larger with more entities. Again, the packet count does not

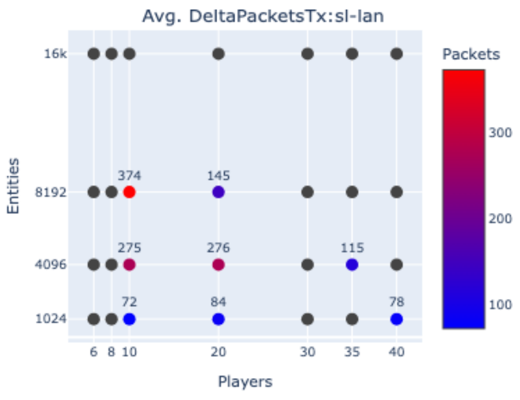


(a) lan

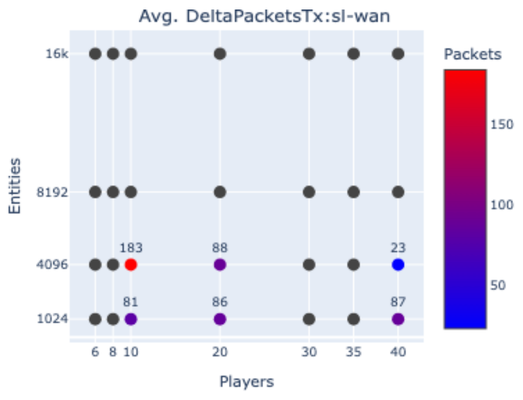


(b) wan

Figure 6.11: Overview graph: DeltaPacketsLost:dl



(a) lan



(b) wan

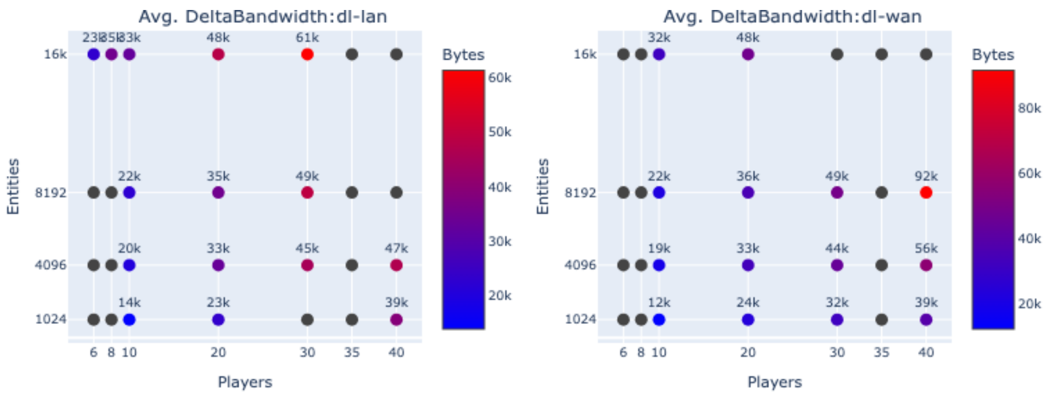
Figure 6.12: Overview graph: DeltaPacketsLost:sl

increase with player count, which is counter-intuitive and was already investigated in ([Packets1])).

In wan `p20+-e4096` must be investigated, since it has too few packets [**Packets5**]. Config `p30-e4096` will result in many lags for clients, because the host has problems with the snapshot distribution due to frequent packet re-transmissions ([Packets5])). `p40-e4096` is not possible; the game crashed after a few seconds ([Packets5])). In summary, new limits can be identified for sl: vertical 4096 (only for 10 players), horizontal 40 (only for 1024 entities). Furthermore, it seems like snapshot interpolation has problems scaling horizontally, contrary to its theoretical advantage over deterministic lockstep. However, it is assumed that this comes down to implementation flaws. The logs during the evaluation revealed, that the reliable pipeline's send queue was full due to snapshot size and fragmentation. Generally, snapshots should not be sent reliably, but rather discarded if old ones are received. A better snapshot system implementation should improve this performance.

Bytes

The packets to bytes correlation is investigated in [**Bytes1**].



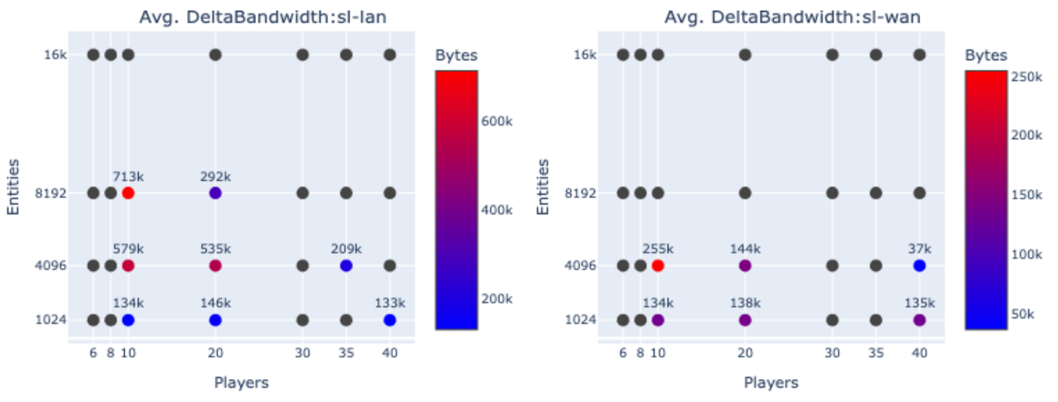
(a) lan (b) wan

Figure 6.13: Overview graph: DeltaBandwidth:dl

Figure 6.13: In lan the bandwidth should not increase with the entity count, investigated in [**Bytes2**]. Bandwidth increases with player count, because more bytes are transmitted from and to new clients. An especially high bandwidth can be observed with `p-30-e16000` ([Packets4]). Ultimately, no new limits can be identified.

In wan there is a similar bandwidth compared to lan, which is reasonable, since packet loss is low. `p-40-e8192` stands out and was already investigated in ([Packets2]). Same limits.

Comparison to snapshot interpolation



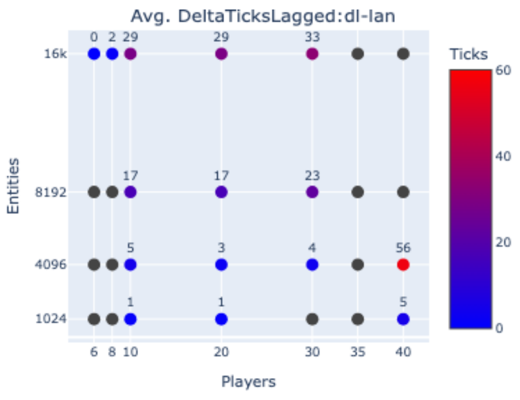
(a) lan (b) wan

Figure 6.14: Overview graph: DeltaBandwidth:sl

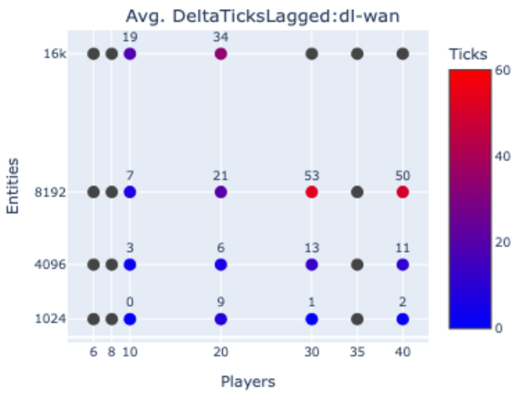
Figure 6.14: Similar results to the packets evaluation are shown. Again, the bandwidth is higher for sl. Again, the bandwidth should scale with entities and players. But, it does not horizontally, because the DeltaBandwidth metric extracted by the statistics provider does not include re-transmitted packets [**Bytes3**]. And, it does not scale horizontally due to the known average over clients problem, already identified in ([FPS1]), ([RTT1]), ([Packets1]): first, the high bandwidth of the host raises the average, and later, the slightly increased bandwidth of the now higher number of clients keeps the average in a similar range for all configurations, although the bandwidth indeed increases with players. This effect would not be visible if the player count would increase above 40. The outliers `lan-p20-e8192` and `lan-p35-e4096` can be explained by [**Bytes4**] and `wan-p40-e4096` by ([Packets5]). On average not a megabyte is sent, but ([Bytes4]) shows, that the host needs a relatively high uplink bandwidth compared to clients. In conclusion, the bandwidth confirms the lower limits for sl, that were already found during the packets evaluation.

Lags and desyncs (dl only)

Generally, an average of 60 lagged ticks per second means that the game does barely progress and is therefore unplayable. There are 15 ticks per 250ms turn, which results in a tick interval of 16.666ms or 60 Hz. Therefore 60 lagged ticks are equivalent to 1 second of lag (16.666ms * 60 = 1000ms = 1 second). Because the sampling rate of the statistics provider is also 1 second, then all ticks since the last sample were lagged.



(a) lan



(b) wan

Figure 6.15: Overview graph: DeltaTicksLagged:dl

Figure 6.15: In lan lagged ticks increase with entity count [**Lags1**]. There is also an increasing lag with player count, even though the increase is only subtle with 1024 entities. This comes down to a disadvantage of dl: with a rising number of clients the probability of worse computer or network conditions and therefore lags

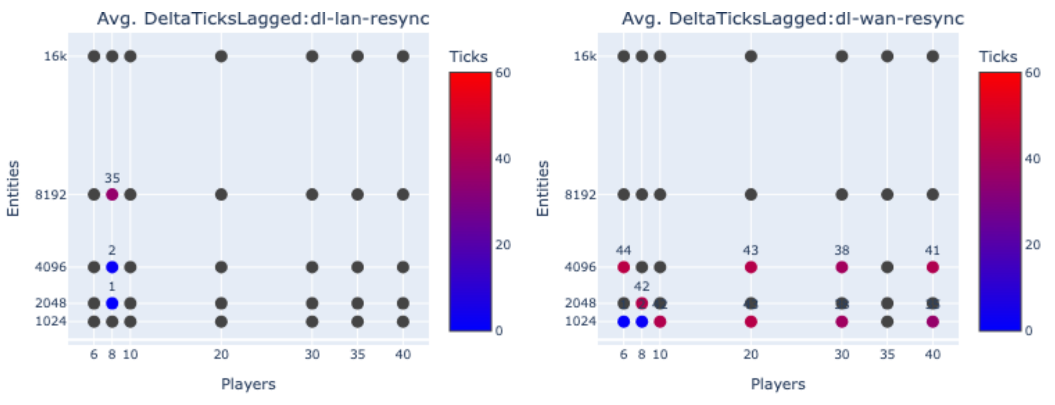
increases. In summary, this figure provides detailed limits: `p6` has no indicated entity limit; `p10` to `p30` have every second tick lagged with 16,000 entities, which is not ideal; almost every third tick is lagged with 8192 entities ([Lags1]); `p40` works with 1024 entities, but 56 lagged ticks per second with 4096 entities renders the game unplayable. Therefore, new limits for dl are found: while the vertical limit is still 16,000 entities, the horizontal limit supports 40 players with only 1024 entities and is hardware related ([Lags1]).

In wan the results are similar, but `p30-e8192` and `p40-e8192` are unplayable with about 50 lags per second. Compared to lan, configuration `p40-e4096` works better in wan and is indeed playable [**Lags2**].

Not one desync was recorded during the whole evaluation, which shows that the simulation runs deterministically.

In general, after evaluating all metrics, the DeltaTicksLagged and FPS metrics provide good insights for deterministic lockstep limitations, whereas the DeltaBandwidth and FPS metric as well is valuable for snapshot interpolation limits.

6.3.2 Resynchronization (dl only)



(a) lan (b) wan

Figure 6.16: Overview graph: DeltaTicksLagged:dl-resync

Figure 6.16: [**Resync1**] dives deeper in order to understand the bad resynchronization performance shown. The more entities the longer the waiting time (lagged ticks), since more data must be synchronized (larger snapshots). In lan a maximum of 4096 entities and 8 players is possible and in wan only maximum 1024 entities and 8 players. This means that resynchronization has a limitation of 1024 to 4096 entities vertically and 8 players horizontally.

6.3.3 Special network conditions

Only LAN and 10 players are evaluated, since the worse conditions of WAN and the probability of one lagging client are simulated.

Latency 200ms

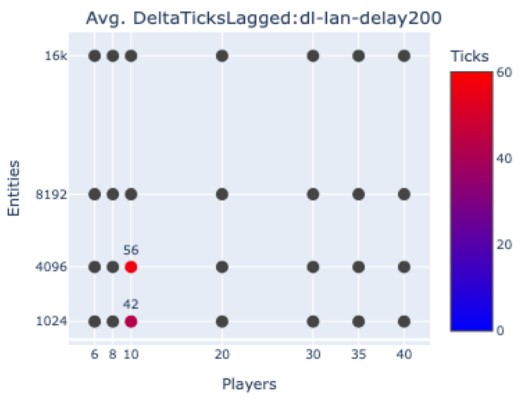
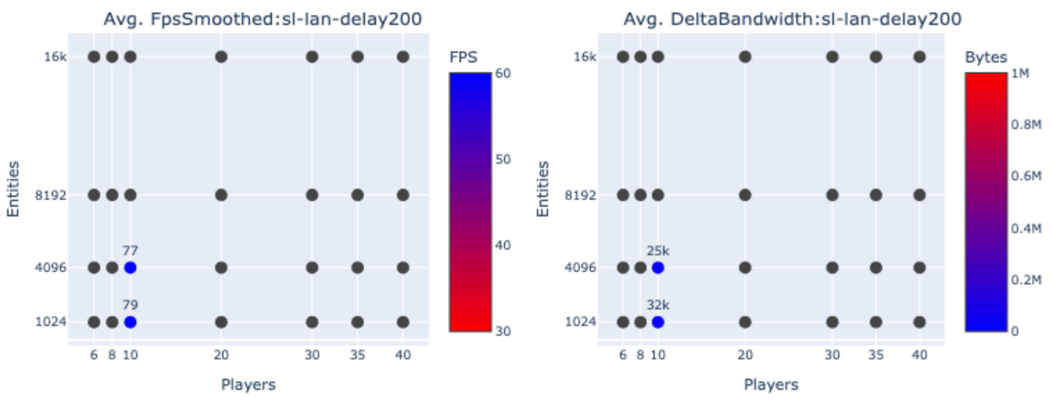


Figure 6.17: Overview graph: DeltaTicksLagged:dl-delay200

Figure 6.17: Only barely playable conditions are achieved with 1024 entities, but 56 lagged ticks per second with 4096 entities is unplayable. The investigation [**Latency1**] reveals, that latency induced a desynchronization. Since resynchronization is only possible for up to 8 players, the system failed to continue with 4096 entities explaining the high average lag. So, latency has an effect on deterministic lockstep limitations, reducing them to the resynchronization limits of 1024 to 4096 entities and 8 players. A better resynchronization process would improve this limit.

Comparison to snapshot interpolation



(a) FpsSmoothed (b) DeltaBandwidth

Figure 6.18: Overview graph: sl-delay200

Figure 6.18: The same limits as without the latency in wan can be observed. Latency has no significant impact on snapshot interpolation except for increased input to action delay for players.

Jitter 100ms

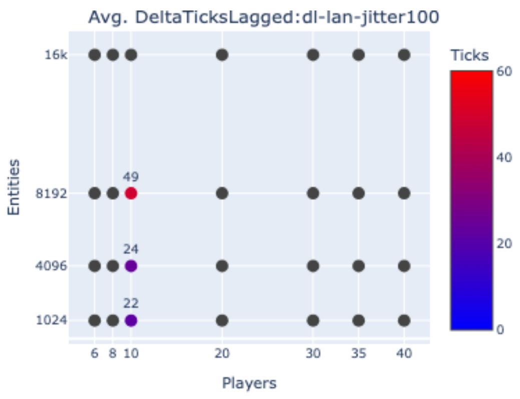


Figure 6.19: Overview graph: DeltaTicksLagged:dl-jitter100

Figure 6.19: Compared to a latency of 200ms, 8192 entities are barely possible again. Lower entity counts work with every third tick lagging. Still, jitter has a large impact on dl performance, since it is similar to latency and has a lower value [**Jitter1**].

Comparison to snapshot interpolation

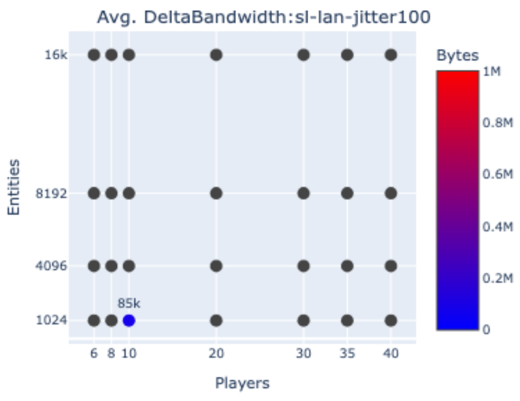


Figure 6.20: Overview graph: DeltaBandwidth:sl-jitter100

Figure 6.20: Only one config data point is available. Hence, the limit is 1024 entities. Jitter has a stronger impact on sl, although it is a similar effect to latency as it is the variation of latency. This is investigated in [**Jitter2**].

Packet loss 10%

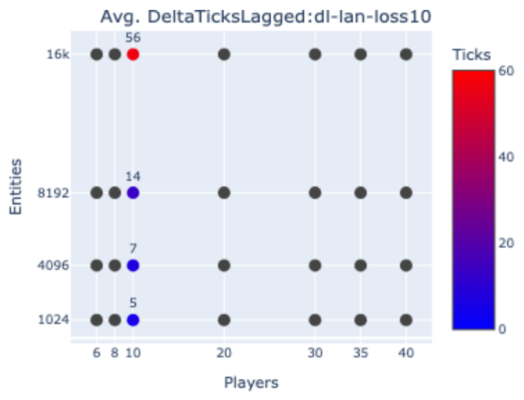


Figure 6.21: Overview graph: DeltaTicksLagged:dl-loss10

Figure 6.21: Similar limits as without the packet loss are shown. The indicated lags are mostly playable, but not with `e16000` [**Loss1**]. Generally, loss has a small impact on dl performance. Only with 16,000 entities loss induced a desynchronization rendering the game unplayable.

Comparison to snapshot interpolation

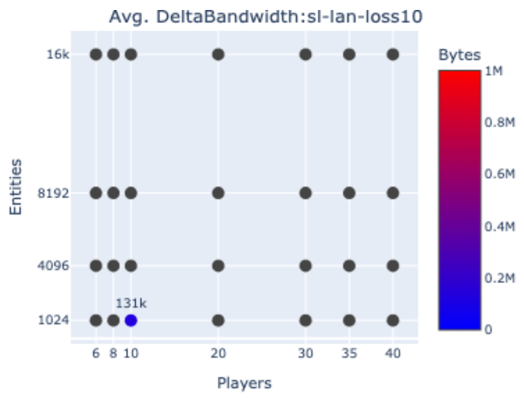


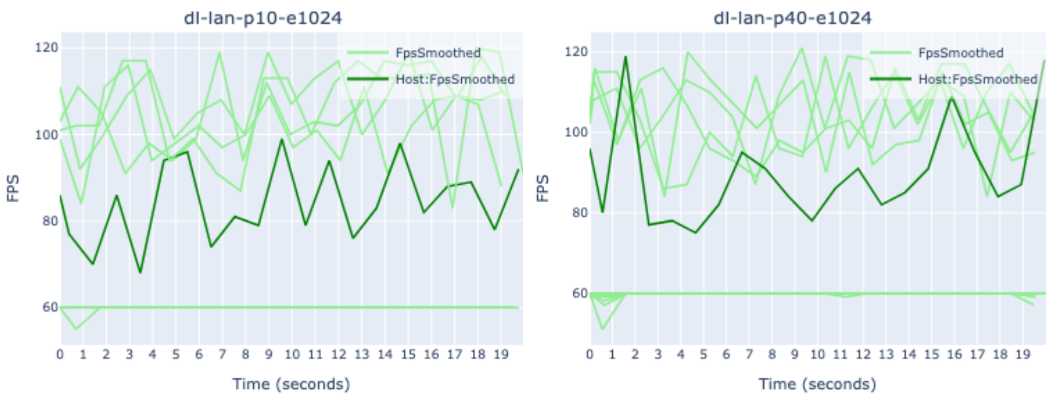
Figure 6.22: Overview graph: DeltaBandwidth:sl-loss10

Figure 6.22: Again, only one data point is available. So, packet loss also has a negative impact on sl performance.

6.3.4 Further investigations

For the further investigations and discussion a line plot is used, that enables a more in-depth analysis of metrics and limitations (detail graph). The various metric values are shown on the y-axis over time (x-axis). Client and host data is separated, where host data usually uses a darker tone of the client color to keep an association, since multiple different metrics can be shown in one graph. Sometimes two different y-axis are used in case the units of multiple metrics do not match. The title is based on the configuration naming scheme.

[FPS1] Scaling horizontally decreases FPS (dl-lan)



(a) (b)

Figure 6.23: Detail graph: [FPS1].1

Figure 6.23: Generally, the FPS of Windows (win) clients is stable at 60 FPS, which is the maximum possible due to 60 Hz monitors and vertical synchronization (V-Sync). macOS (mac) clients have a higher FPS due to 120 Hz monitors (more unstable). The host (on mac) has a lower FPS compared to the mac clients on the same machine, which comes down to the increased processing workload. A small detail can be observerd: FPS seem to be higher for the host when FPS for clients are lower, e.g. at second 5. The same computer has to distribute its resources.

Comparing `p-10-e1024` to `p40-e1024` (same entities for four times the players), win computers are in both configuations stable at 60 FPS and macs have an average of about 100 FPS. Therefore, contrary to the indications of the corresponding overview graph, player count has no significant effect on FPS. But, more win computers are used, which means the average moves towards 60 FPS.

Figure 6.24: When using the median instead of average in the overview graph, the finding is confirmed, since FPS are stable scaling horizontally at 60 FPS, except for the limiting config of `p30-e16000`. The median also reveals that scaling vertically

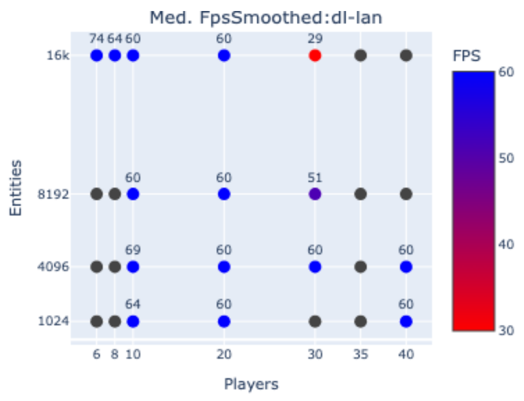


Figure 6.24: Overview graph: Median FpsSmoothed:dl-lan

achieves stable FPS as well, but this is due to the fact, that only some win computers start to drop FPS, while some other wins and especially macs keep the median high. For this case average is more suited.

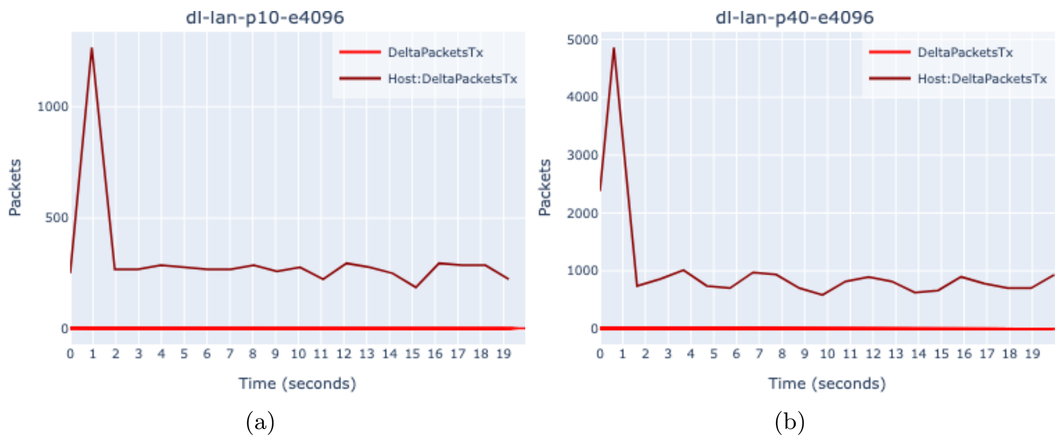
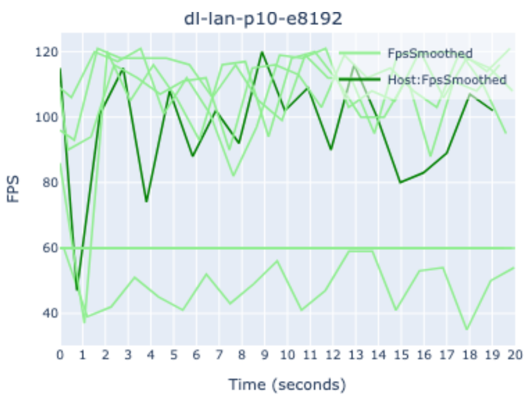


Figure 6.25: Detail graph: [FPS1].3

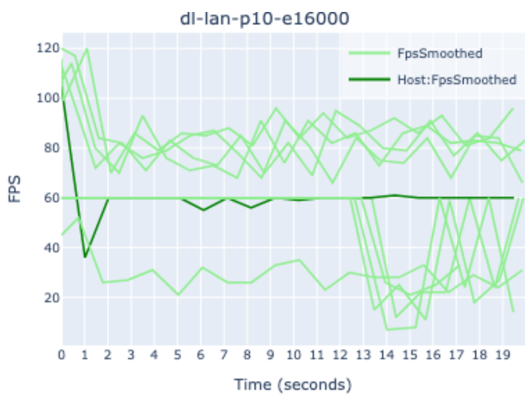
Figure 6.25: The lower FPS of the host can be explained with a higher workload and bandwidth, since he has to send more packets. The figure shows, that indeed the host must send more packets (dark red line, light red line is not important). The initial spikes are due to synchronizing the entity number across all clients (IDs are exchanged, entities are spawned). About four times more packets are sent for four times the player count. In general, the player count is not limited by FPS (CPU/GPU), at least with up to 40 players, since there is a small impact shown for the host.

[go back to first mention]

[FPS2] Potential entity limit for low player counts (dl)



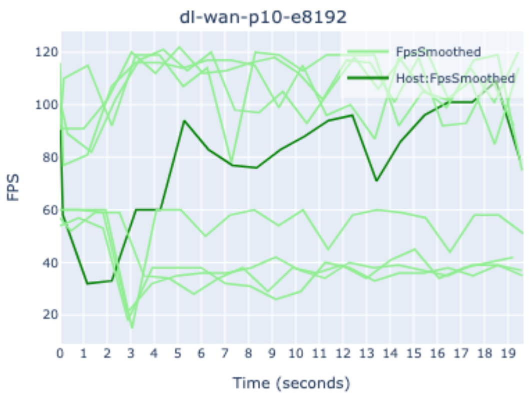
(a)



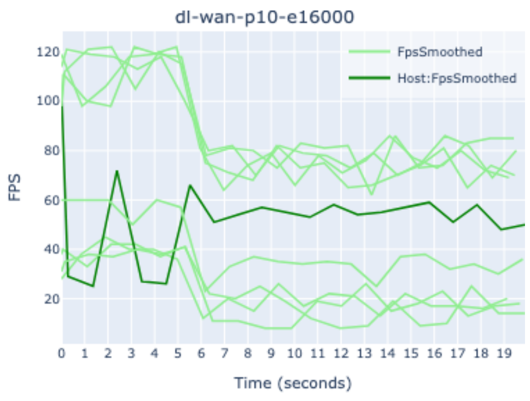
(b)

Figure 6.26: Detail graph: [FPS2].1

Figure 6.26: With 8192 entities one win client starts falling behind, and even stronger with 16,000 entities (client "c66..."). The FPS drop in the beginning, which is again due to synchronizing the initial entity number. 16,000 entities is still possible, but win clients start to fall below 30 FPS and the host drops to 60 FPS. Other mac clients have an average of 80 FPS, which is still very playable. So in summary, more entities are possible (32,000 on mac, only max. 16,000 on win), but CPU/GPU limits start to show through FPS at 16,000 entities based on the hardware.



(a)



(b)

Figure 6.27: Detail graph: [FPS2].2

Figure 6.27: In wan and with 8192 entities win clients start to have lower FPS (still 40). 16,000 entities have a significant impact on mac and an even higher impact on win (mostly below 20 FPS). Also, the initial entity synchronization takes 5 seconds,

compared to only 2 in lan. This results in slightly lower CPU/GPU limits in wan. But why does wan degrade performance compared to lan, since the same amount of data is sent? ([FPS4])) also investigates this question, but the following figure 6.28 gives additional insights.

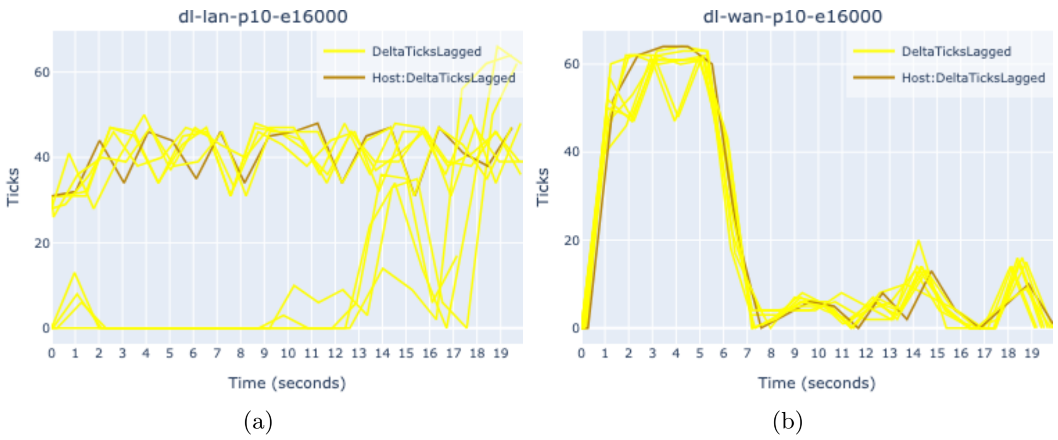


Figure 6.28: Detail graph: [FPS2].3

Figure 6.28: Less lags can be observed in wan. The fewer lags lead to more simulation steps and less waiting. This means that in wan the FPS are lower, because more computation is performed. FPS is therefore influenced by the deterministic lockstep performance. In summary, more waiting time leads to more FPS, less waiting time leads to less FPS (positive correlation).

The fact, that there are less lags in wan, is unfortunately not explainable through the available data. Host and clients have similar lag behavior: if one player waits, all have to wait.

[*go back to first mention*]

[FPS3] 40 players entity limit (dl-lan)

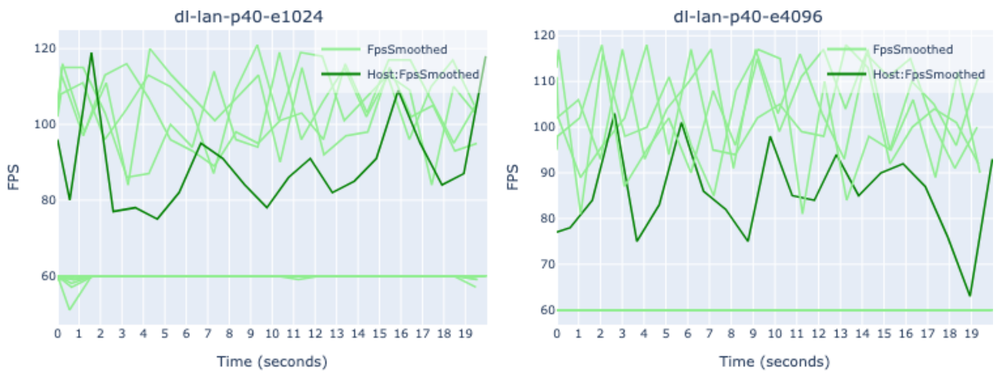


Figure 6.29: Detail graph: [FPS3].1

Figure 6.29 shows, that the limit is not induced by FPS.

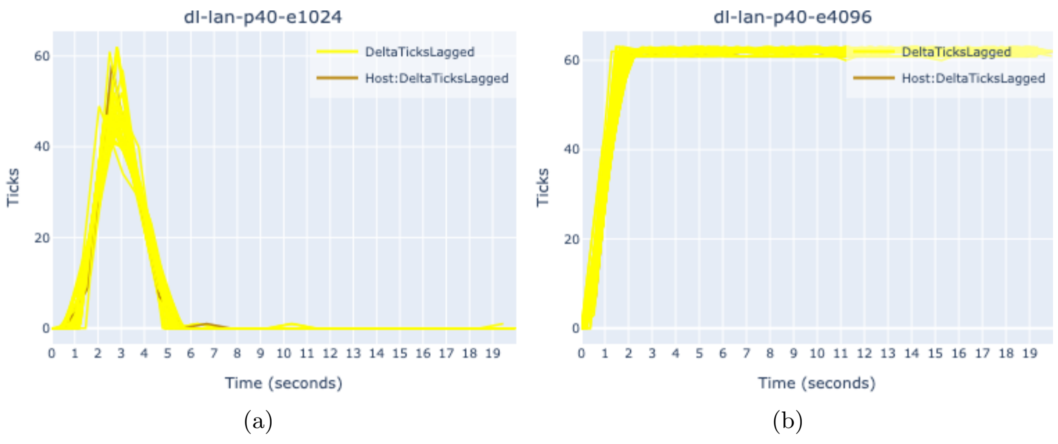


Figure 6.30: Detail graph: [FPS3].2

Figure 6.30: Again, an initial entity synchronization spike can be seen. Thereafter, smooth gameplay with 1024 entities is possible (few lags). However, with 4096 entities the game is totally unplayable, since there is no game progress. This shows a clear limit and is an implementation problem, because too much data for the reliable send queue must be shared in the beginning, which leads to the game never starting.

[*go back to first mention*]

[FPS4] FPS are less in WAN (dl)

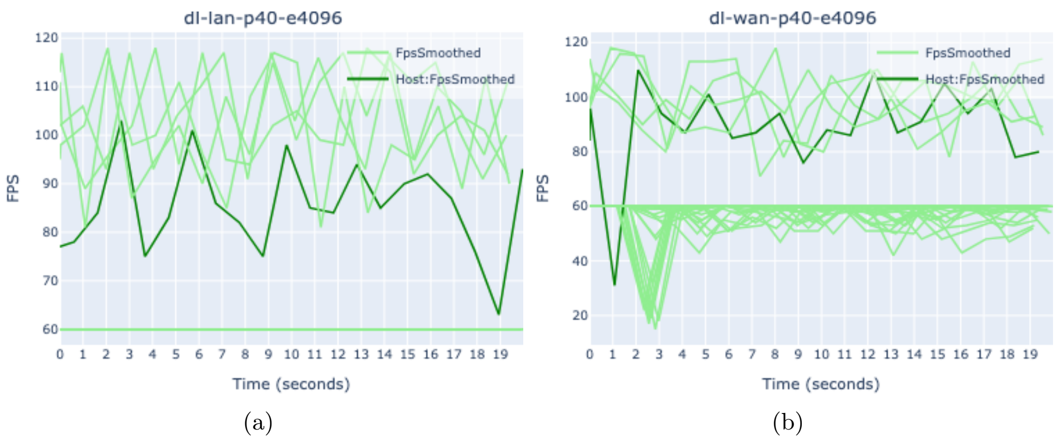


Figure 6.31: Detail graph: [FPS4].1

Figure 6.31: Especially the win clients have slightly lower FPS in wan. This should be due to more re-transmissions and therefore more more workload.

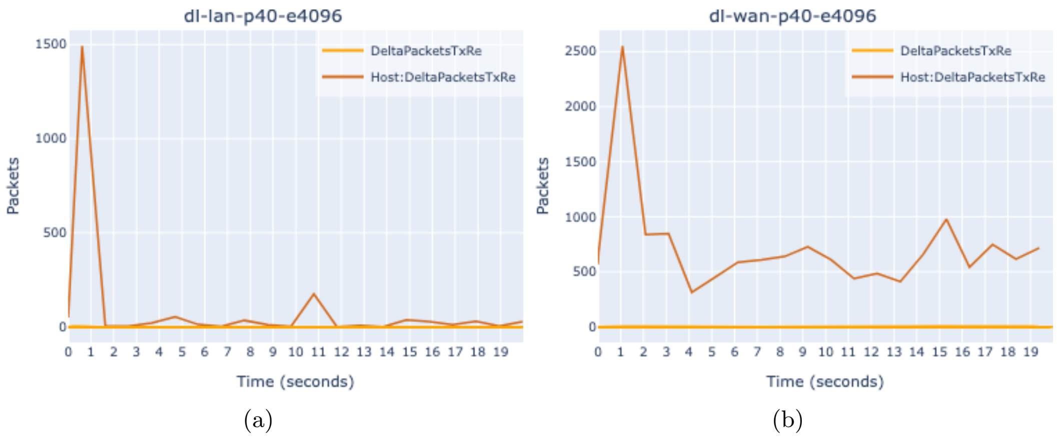


Figure 6.32: Detail graph: [FPS4].2

Figure 6.32 confirms this assumption. The host has to re-transmit approximately 500 packets per second, compared to only a few in lan.

[go back to first mention]

[FPS5] Explaining the entity limit increase for 40 players in dl-wan

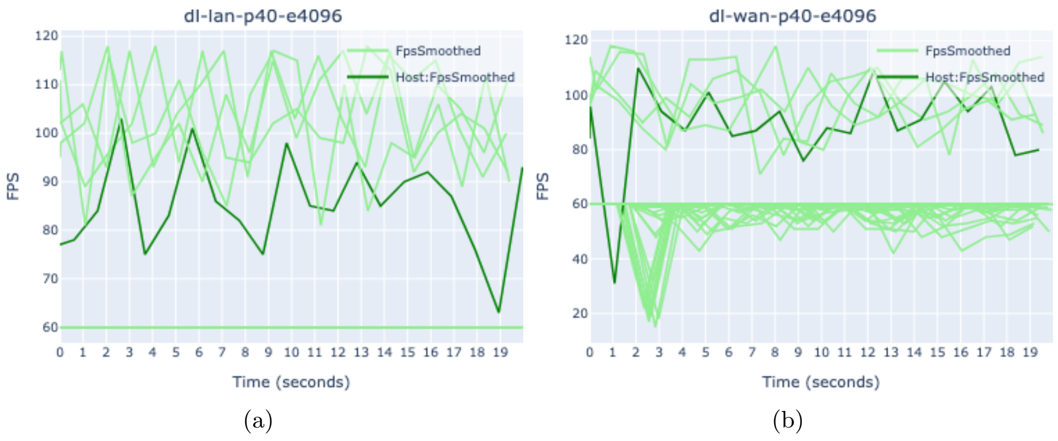
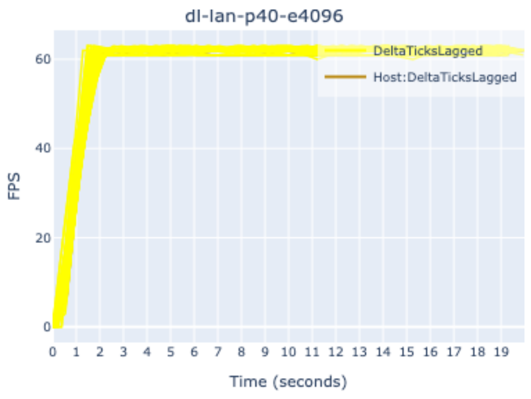


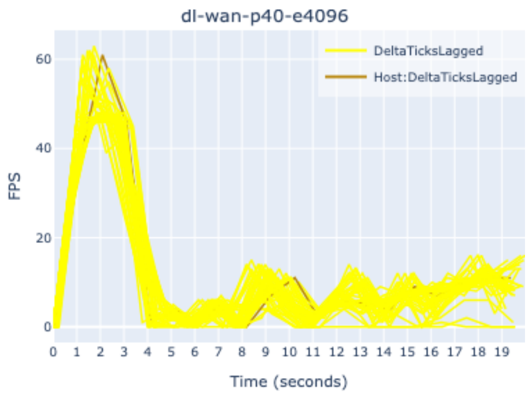
Figure 6.33: Detail graph: [FPS5].1

Figure 6.33: LAN is more stable with 4096 entities; so why does wan manage to handle 8192 entities as well compared to lan? Though, some clients have bad performance making 8192 entities unplayable.
Figure 6.34: In lan the 4096 entities config is completely lagged, while in wan the configuration is still playable. With 8192 entities in wan there is also a complete simulation pause in the end, but still, wan performs better (related to [FPS2]).

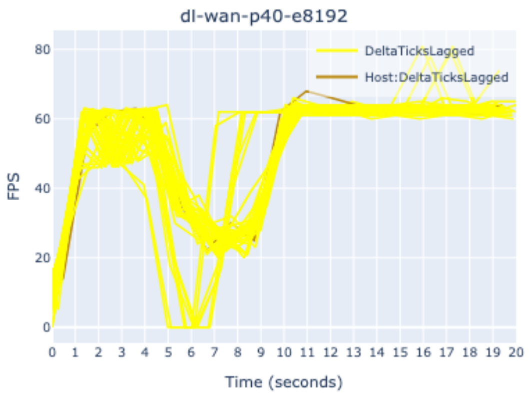
[go back to first mention]



(a)



(b)

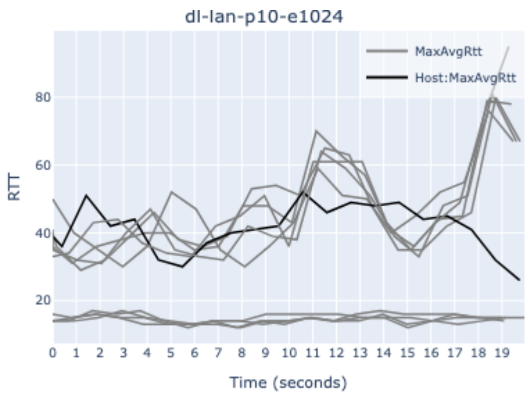


(c)

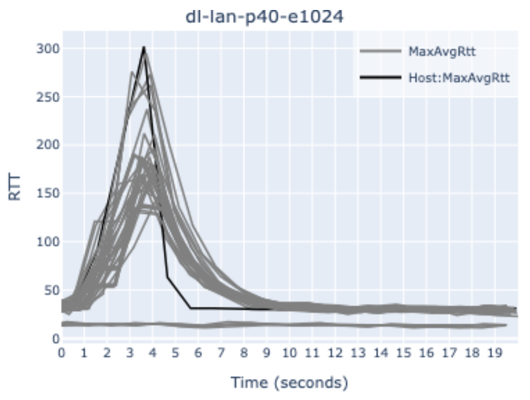
Figure 6.34: Detail graph: [FPS5].2

[RTT1] Tendency of increasing RTTs with entity and player count (dl)

No answer was found based on data for the vertical RTT increase.

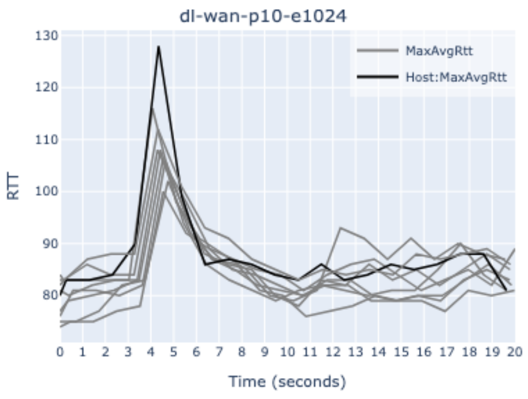


(a)

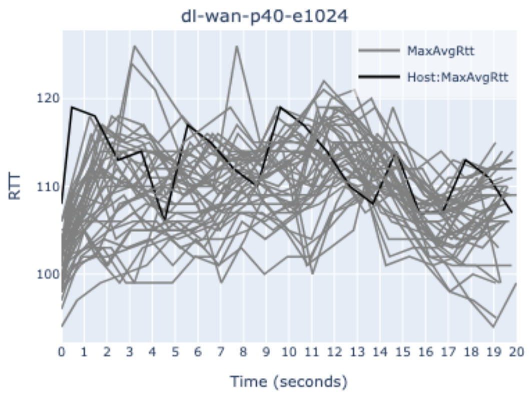


(b)

Figure 6.35: Detail graph: [RTT1].1



(a)



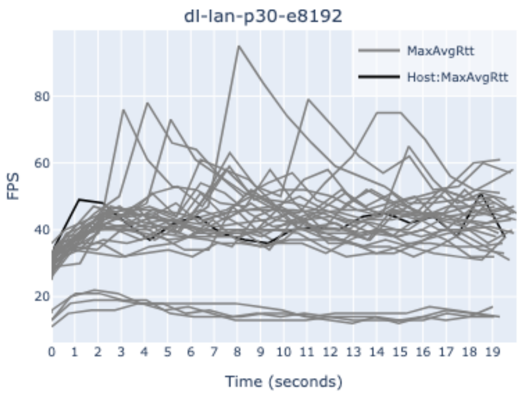
(b)

Figure 6.36: Detail graph: [RTT1].2

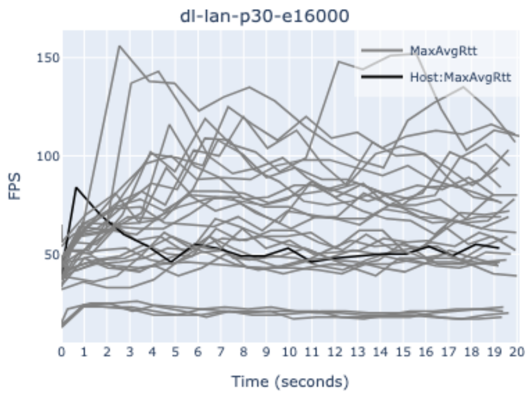
Figure 6.35 and 6.36: Similar to ([FPS1]), win clients generally have a higher RTT compared to mac. They increase the average RTT, since only five mac clients are used and win clients are scaled horizontally. But, there is still a tendency in wan. RTTs are generally higher in wan and mac and win RTTs are approaching each other. However, RTTs increase nevertheless, which might be due to network congestion or a temporary effect.

[*go back to first mention*]

[RTT2] Investigation of dl-lan-p30-e16000



(a)

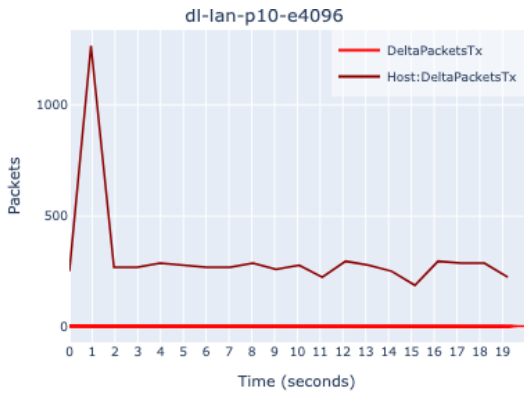


(b)

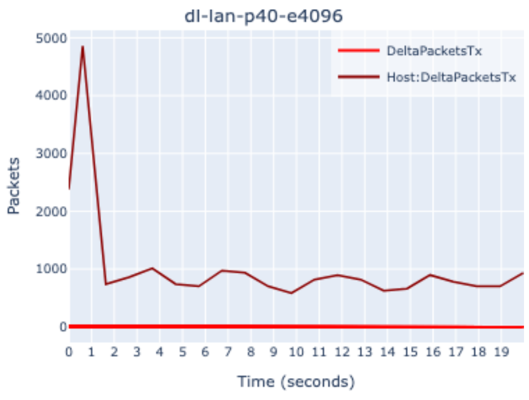
Figure 6.37: Detail graph: [RTT2].1

Figure 6.37: Similar to ([RTT1]), the increase in RTT for higher entity counts cannot be explained by data. It is clear, that RTT can be a limiting factor. For dl the RTT should be smaller than the turn duration. So, `e16000` shows a limit approaching. [*go back to first mention*]

[Packets1] Packet count should scale linearly with player count (dl)



(a)



(b)

Figure 6.38: Detail graph: [Packets1].1

Figure 6.38: Contrary to the overview graph, the packet tx count increases by a factor 4 for the host when the player count is scaled by the same factor ([FPS1], same graphic). But, similar to both ([FPS1]) and ([RTT1]), the increasing client count lowers the average, because clients send less packets than the one host (about 5 per second). The median, again, will show that.

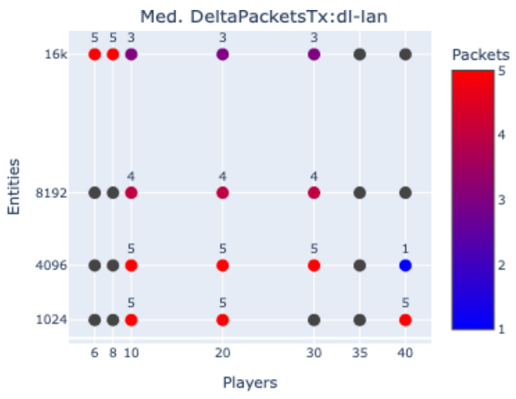


Figure 6.39: Overview graph: Median DeltaPacketsTx:dl-lan

Figure 6.39: The median also shows that the packets tx decrease in more complex configs. This could be due to smaller packets being packaged together, since the game starts to slow down, and packet transmissions are delayed, so that more packets are in the send queue, that will be packaged.

[*go back to first mention*]

[Packets2] Investigation of dl-lan-p40-e4096

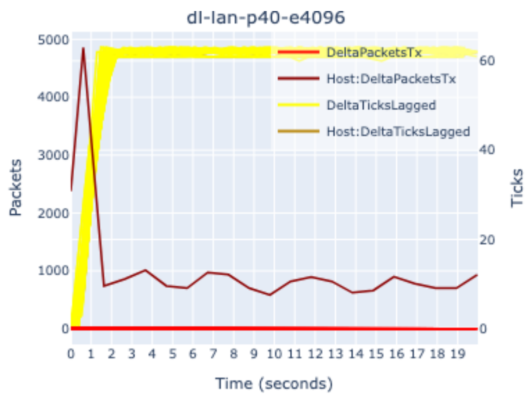


Figure 6.40: Detail graph: [Packets2].1

Figure 6.40: The game seems to never really start, since from second 2 on, after the initial game state has been synchronized, the game is completely lagged. The initial and further host packets, that were transmitted, increase the average of the overview graph, but the clients only tx one packet, that could be the heartbeat, but no commands. It is unknown why the host keeps transmitting that many packets.

[*go back to first mention*]

[Packets3] Investigation of dl-wan-p30-e8192 -5 PacketsTx

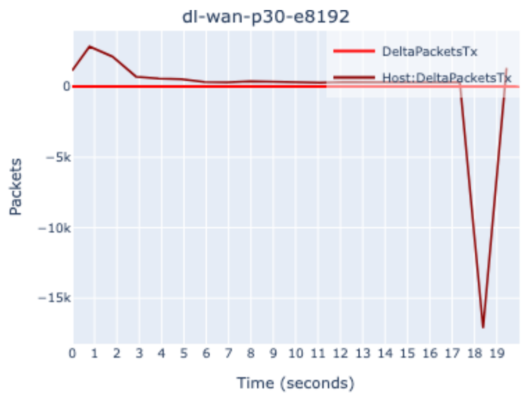


Figure 6.41: Detail graph: [Packets3].1

Figure 6.41: `dl-wan-p30-e8192` must be a bad data point at second 18., e.g. an implementation bug, since -5 packets is not possible. Maybe there was a data type overflow due to a high packet count while stopping the evaluation.

[*go back to first mention*]

[Packets4] Does a high packet re-transmission count indicate a limit? (dl-lan-p30-e16000)

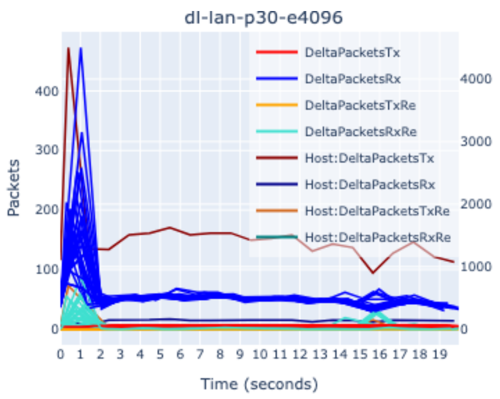
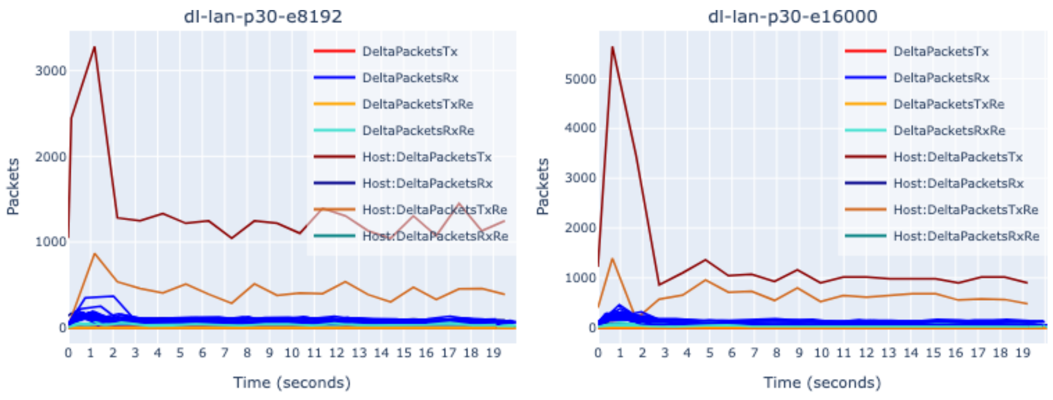


Figure 6.42: Detail graph: [Packets4].1

Figure 6.42: The host transmits about 1500 packets, while each client receives about 50 packets. This sums up reasonably: 50 packets * 30 clients (only 29 without the host) = 1500 packets sent by host. Each client transmits about 5 packets (difficult to see in the figure, but is known), whereas the host receives about 150 packets: 5 packets * 29 clients = 150 packets received by the host. There are few re-transmissions with 30 players and 4096 entities.



(a) (b)

Figure 6.43: Detail graph: [Packets4].2

Figure 6.43: One finding of this figure is, that re-transmitted and re-received (duplicate) packets do not count as regular transmitted and received packets, but are a separate value, because the host's transmitted and re-transmitted packets summed results in the target 1500 packets, that are assumed (see also [Bytes3]).

The ratio between transmitted and re-transmitted packets $\frac{DeltaPacketsTxRe}{DeltaPacketsTx}$ starts to rise from 0 and approaches 1 with 16,000 entities. When the ratio rises more packets are resent, while less new packets can be sent, since the reliable send queue is full, which delays game synchronization. So yes, a high packet re-transmission count indicates a limit, since the game cannot progress that fast, due to reliable send queue congestion.

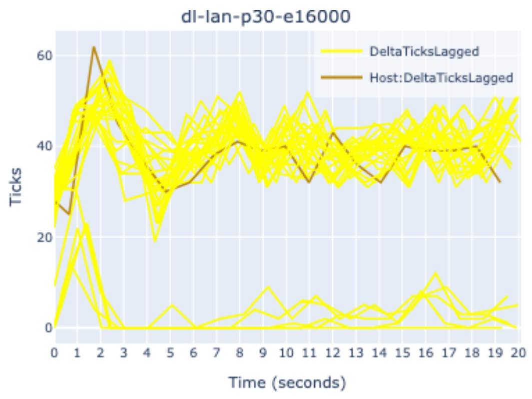


Figure 6.44: Detail graph: [Packets4].3

Figure 6.44: Game progression problems are indicated by lags as well. Some clients also lag behind due to hardware limitations (only few lags), and others additionally have to wait for them.

[go back to first mention]

[Packets5] Investigation of sl-wan-p20+-e4096

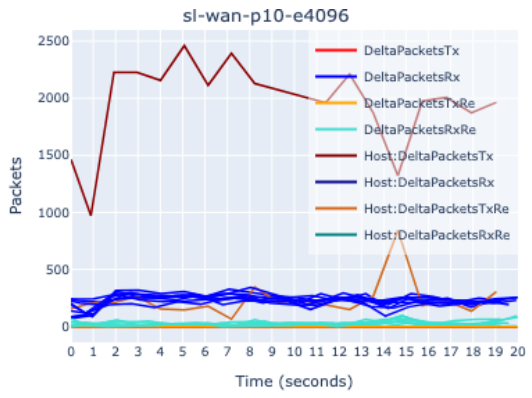


Figure 6.45: Detail graph: [Packets5].1

Figure 6.45: `sl-wan-p10-e4096` indicates no problems.
Figure 6.46: `sl-wan-p20-e4096` shows a rising packets re-transmission count, which is a problem according to ([Packets4]). The default packet transmission count, on the other hand, is falling, which explains the lower average packet tx count in the overview graph.

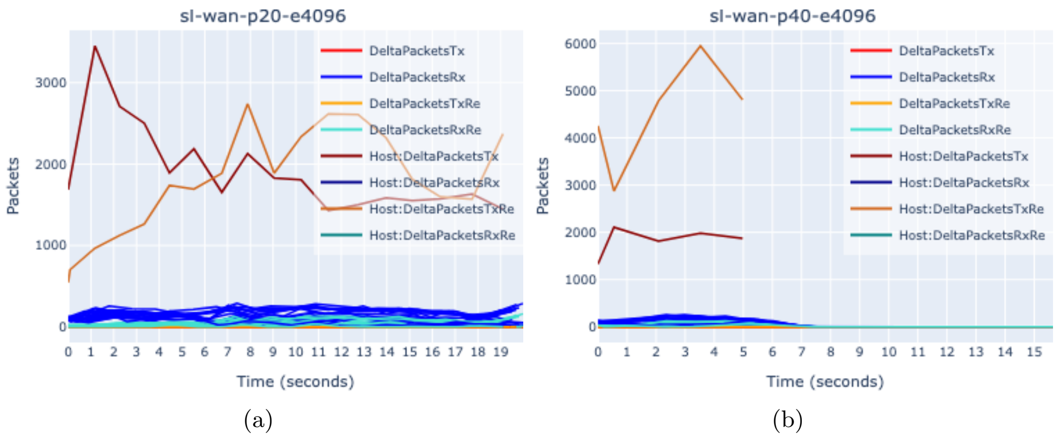


Figure 6.46: Detail graph: [Packets5].2

The `sl-wan-p40-e4096` configuration reveals a game crash after 5 seconds on the host and therefore a new limit. This is why the packets tx count is low in the overview graph. Also, the packet txre count is higher than the default tx count, which is again a problem, since snapshots are congested in the reliable send queue ([Packets4]). Ultimately, `sl-wan-p40-e4096` is not possible.

[go back to first mention]

[Bytes1] Bytes to packets correlation

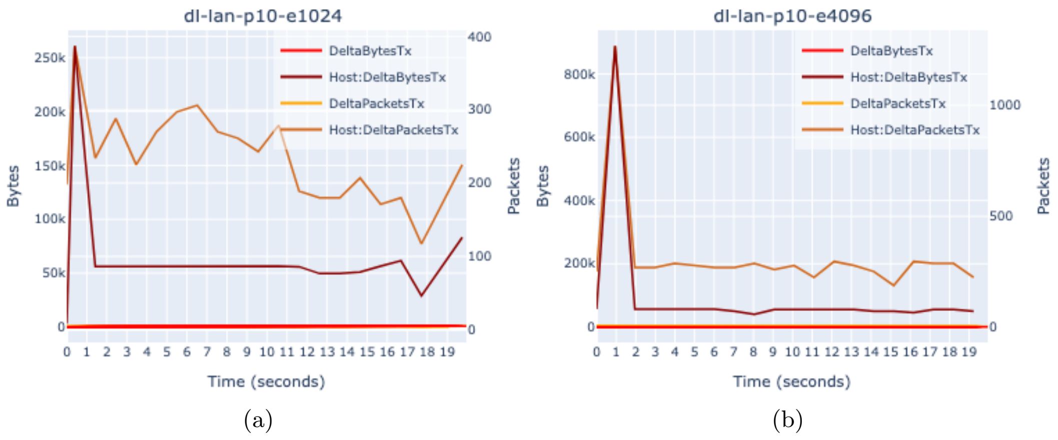
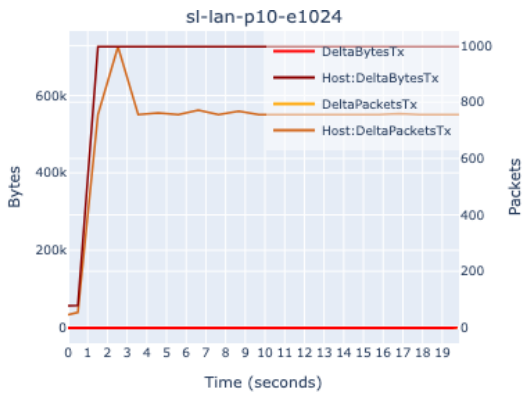
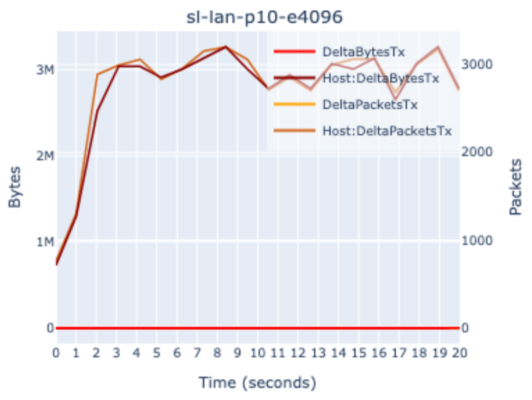


Figure 6.47: Detail graph: [Bytes1].1

Figure 6.47: While the transmitted bytes are relatively constant, a more variable packet transmission count can be observed for the host. Therefore, packets are not always of the same size and are fragmented and packaged based on current conditions. With `e4096`, both metrics behave similarly. No correlation is found based on these two graphs, since in dl the bytes and packets don't scale with entities.



(a)



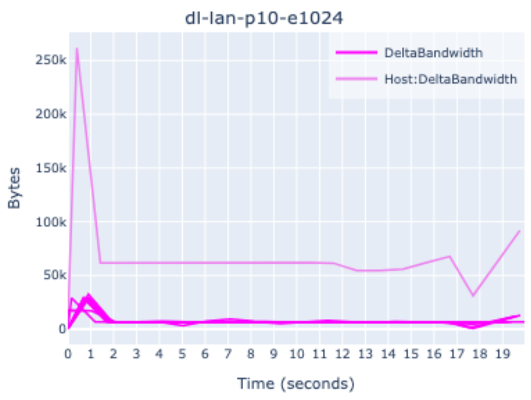
(b)

Figure 6.48: Detail graph: [Bytes1].2

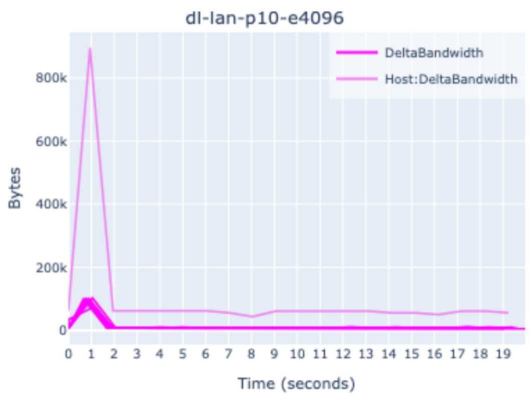
Figure 6.48: With sl, the host transmits a constant byte amount of 730kB for 1024 entities rising to about 3MB for 4096 entities (730kB * 4 = 3MB). The transmitted packets count also rises from close to 800 to about 3000 (800 packets * 4 = 3200 packets). Therefore, packets and bytes correlate positively. Also, note the significant difference between bytes and packets transmitted by the host and clients, especially in sl. This makes determining averages in overview graphs challenging.

[go back to first mention]

[Bytes2] Bandwidth should not increase with entity count (dl)



(a)



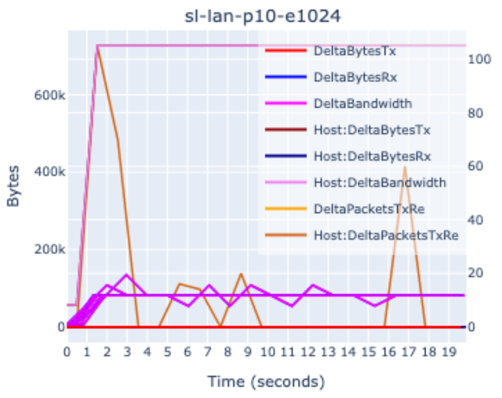
(b)

Figure 6.49: Detail graph: [Bytes2].1

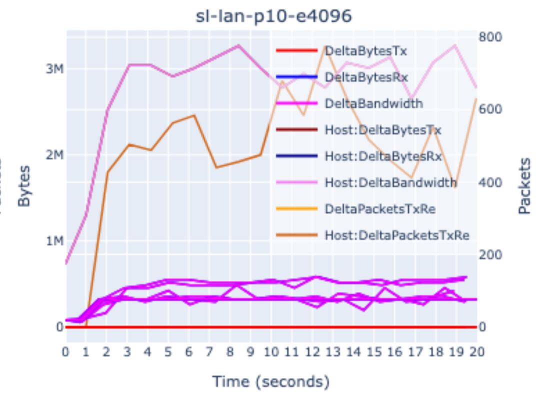
Figure 6.49: Generally, the bandwidth indeed is relatively constant. But, the initial entity synchronization spike increases the average, since more entities are synchronized in the beginning when scaling vertically.

[go back to first mention]

[Bytes3] Bandwidth does not include re-transmission



(a)



(b)

Figure 6.50: Detail graph: [Bytes3].1

Figure 6.50: For 1024 entities, the host bandwidth is about 720kB with mainly regular transmissions and few re-transmissions. Client bandwidth is about 80kB. The host's transmitted bytes amount should match up with the client's received bytes number: 80kB * 9 clients = 720kB.

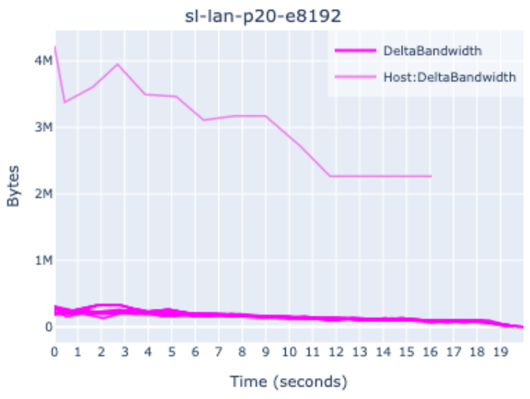
For 4096 entities: the host bandwidth rises to about 3MB, which is approximately 720kB * 4. Client bandwidth is about 300kB-500kB (should be 80kB * 4 = 320kB). The host's DeltaBytesTx (= DeltaBandwidth, few bytes received only) should match up with all client's bandwidth: 320kB * 9 clients = about 3MB. However, in this scenario many re-transmission were recorded. Those are ignored by bandwidth. The implementation adds bytes to the bandwidth only once per packet, and not for later re-transmissions. The actual bandwidth must be higher.

[*go back to first mention*]

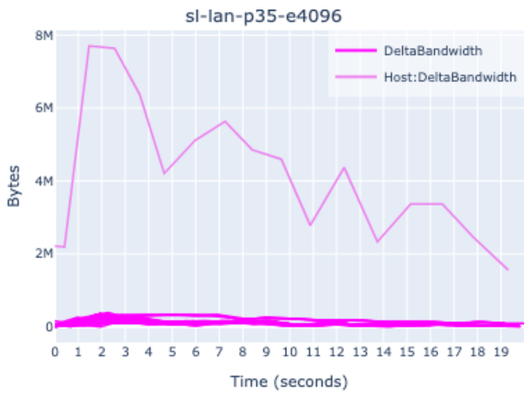
[Bytes4] Investigation of sl-lan-p20-e8192 and sl-lan-p35-e4096

Figure 6.51: In config `sl-lan-p20-e8192` the host crashes after 16 seconds. This leads to a lower average bandwidth. `sl-lan-p35-e4096` illustrates a high uplink requirement for hosts (up to 8MB). It is usual, that uplink speeds are lower, than contracted downlink speeds, so a host with adequate uplink speed must be selected. Another insight is that higher player counts lead to a lower average bandwidth ([Packets1]).

[*go back to first mention*]



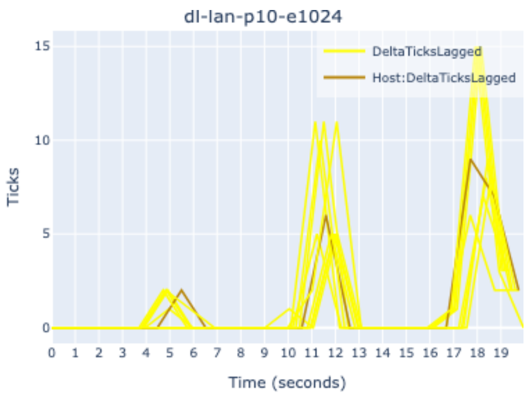
(a)



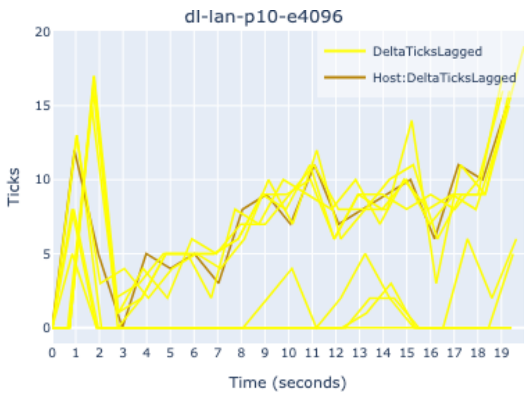
(b)

Figure 6.51: Detail graph: [Bytes4].1

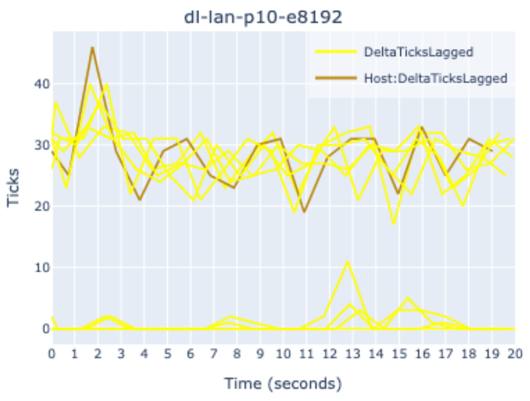
[Lags1] Increasing lags with entity count



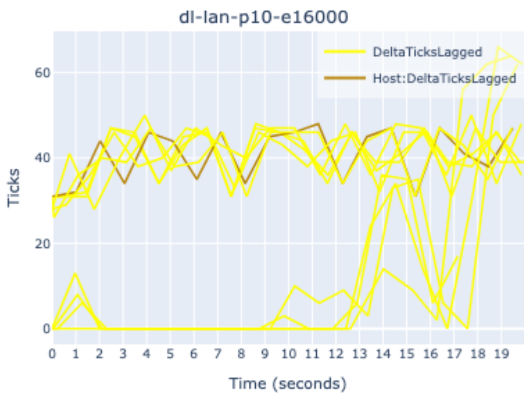
(a)



(b)



(c)



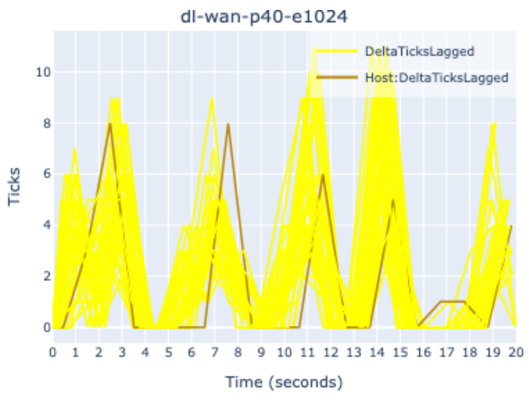
(d)

Figure 6.52: Detail graph: [Lags1].1

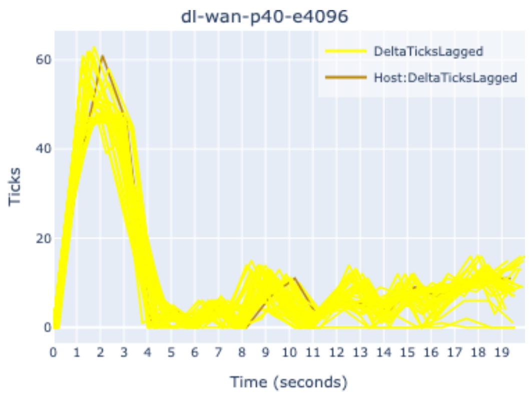
Figure 6.52: Few lag spikes with 1024 entities are shown. A rising lag tendency with 4096 can be observed. With 8192 entities there is a constant lag of 30 indicated (every second tick is lagged), which is playable, but not ideal. There seem to be some slower clients, that must be waited for. With 16,000 entities the game reached an average of 40 lags per second and is barely playable and limiting. Since entity count does not correlate with bandwidth when using dl, the observations must be linked to hardware limitations and some lagging behind clients, that degrade the performance for all clients – a known disadvantage of deterministic lockstep.

[*go back to first mention*]

[Lags2] Investigation of dl-wan-p40-e4096



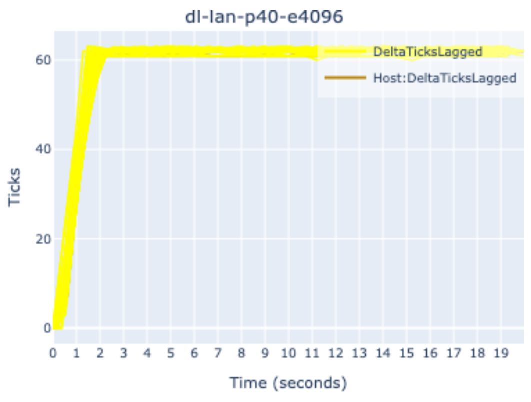
(a)



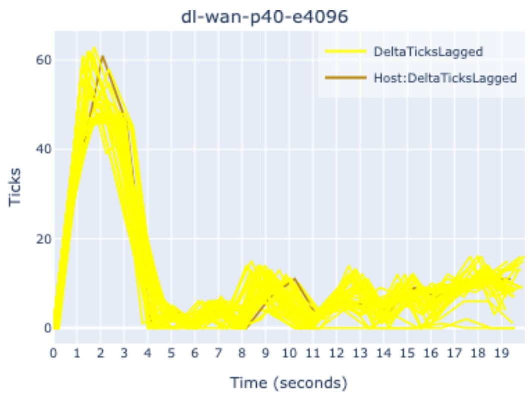
(b)

Figure 6.53: Detail graph: [Lags2].1

Figure 6.53: Indeed, there seems to be no problem. Similar performance to 1024 entities is shown, despite the initial entity synchronization lag spike.



(a)



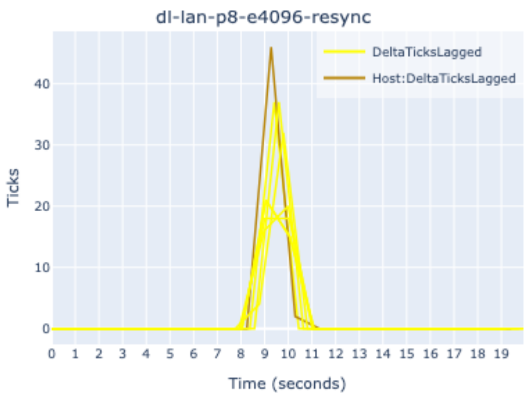
(b)

Figure 6.54: Detail graph: [Lags2].2

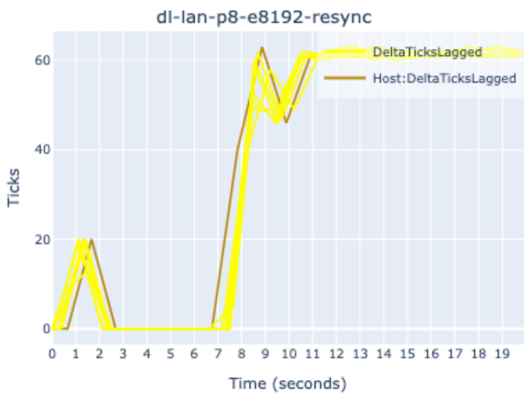
Figure 6.54: WAN performs better than lan, again. Unfortunately, there is no explanation by data.

[*go back to first mention*]

[Resync1] Investigation of resyncs



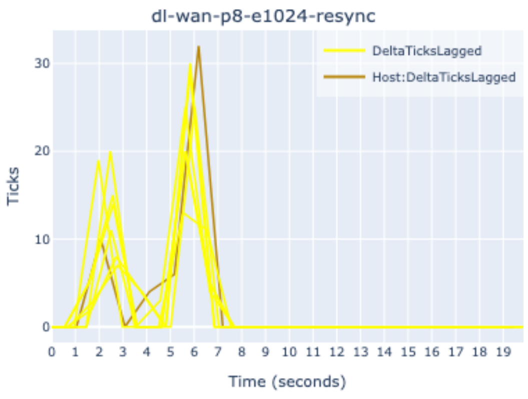
(a)



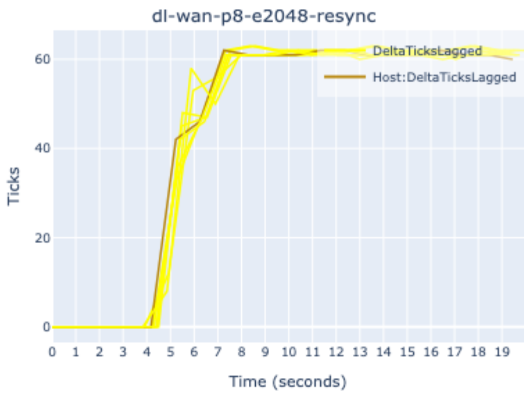
(b)

Figure 6.55: Detail graph: [Resync1].1

Figure 6.55: In lan at second 7 to 8 the resynchronization was performed. With 4096 entities the game could continue progressing after the resync. But with 8192 entities the progression was completely halted. So, 4096 entities and 8 players is the limitation in lan.



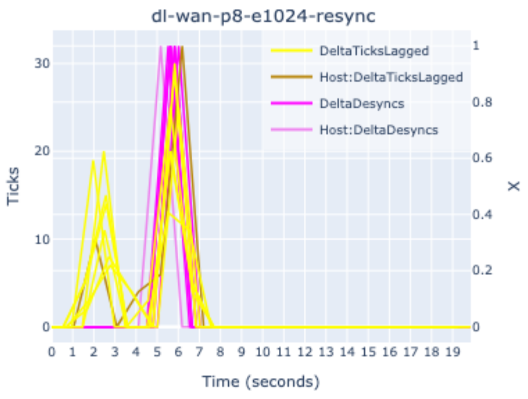
(a)



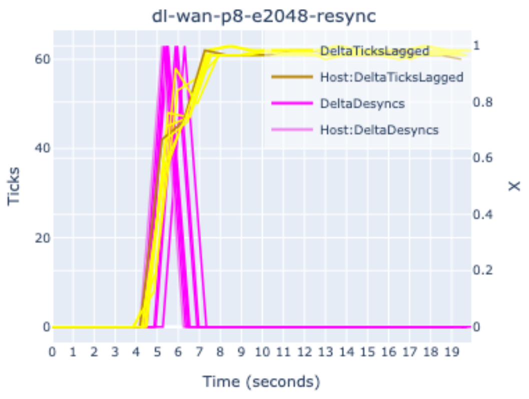
(b)

Figure 6.56: Detail graph: [Resync1].2

Figure 6.56: In wan the resync was performed at second 4 to 5 and the maximum supported configuration is now even lower at 1024 entities and 8 players. It is not possible to resynchronize the game state with 10+ player or 1024+ entities in wan, respectively 4096+ entities in lan.



(a)



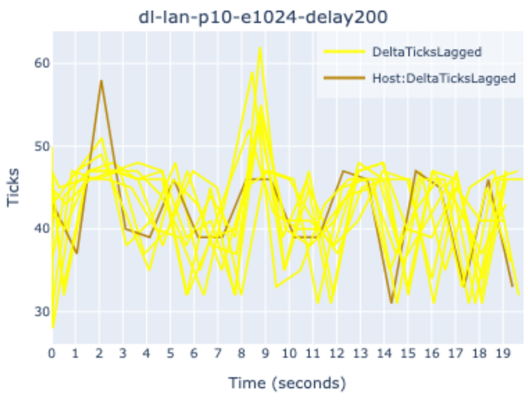
(b)

Figure 6.57: Detail graph: [Resync1].3

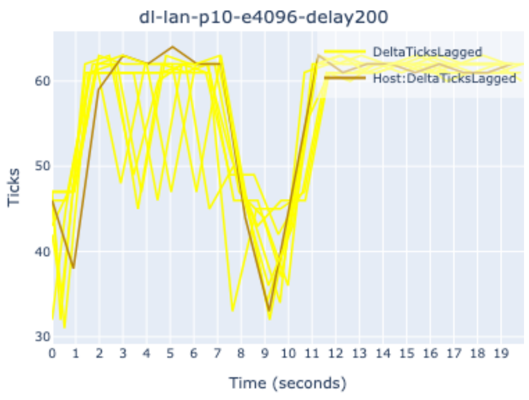
Figure 6.57: No other desyncs except the intentional one are detected and other metrics evaluated also do not provide insights for the reason of the bad resynchronization performance. So, an implementation problem is assumed.

[*go back to first mention*]

[Latency1] Investigation of latency for dl



(a)



(b)

Figure 6.58: Detail graph: [Latency1].1

Figure 6.58: 1024 entities is barely playable and 4096 entities are unplayable.
Figure 6.59: The host has double the RTT delay, since the latency/delay simulator setting is applied to incoming and outgoing packets. Clients have a high delay of 200-400ms as well. Because the turn duration is only 250ms, this means RTTs of over 250ms result in permanent lags and very slow simulation progression. Dynamic turn durations could improve this problem. This is valuable information, but not

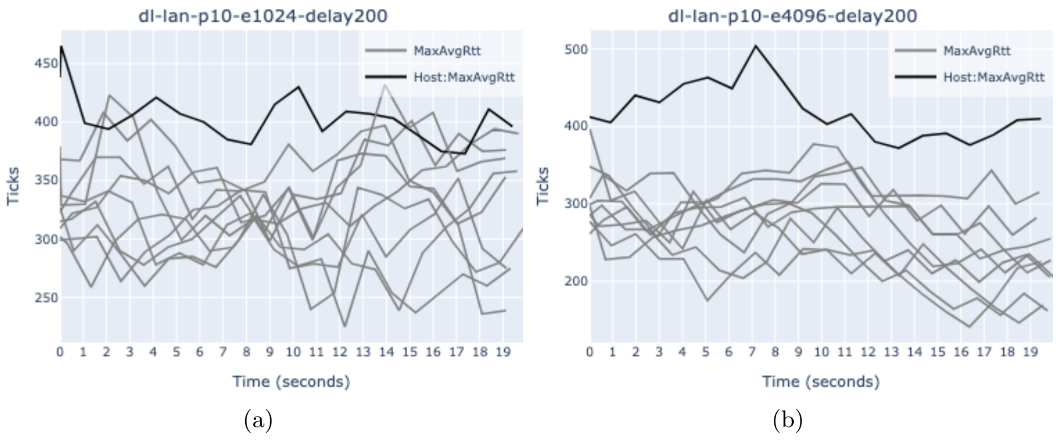


Figure 6.59: Detail graph: [Latency1].2

an explanation for the dl failure with 4096 entities, since RTTs are similar in both configs.

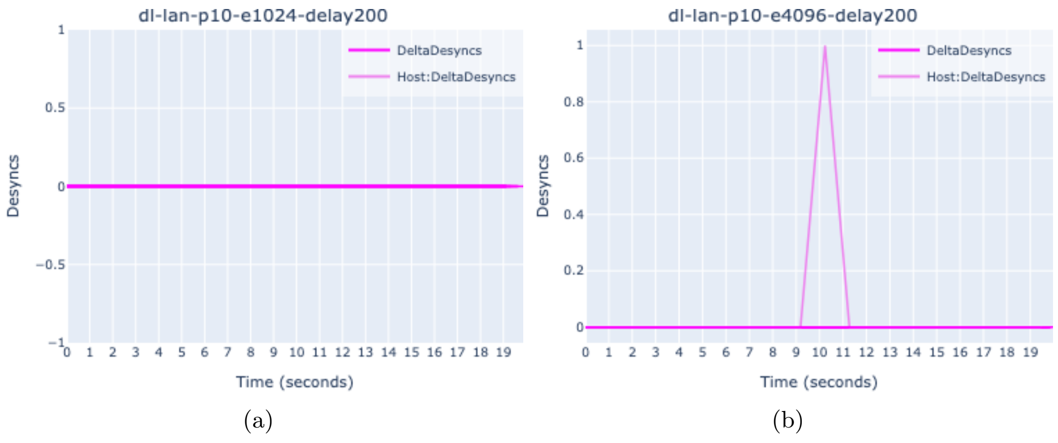


Figure 6.60: Detail graph: [Latency1].3

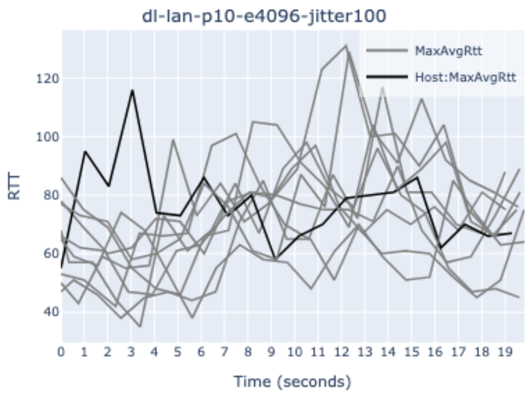
Figure 6.60: In fact, there was an unintentional desync after 10 seconds with the 4096 entities config. As previously mentioned, resyncs are only possible in dl for up to 8 players and 4096 entities. That is why `e4096` with 10 players failed. On a positive note this shows, that the desync detection works.

[*go back to first mention*]

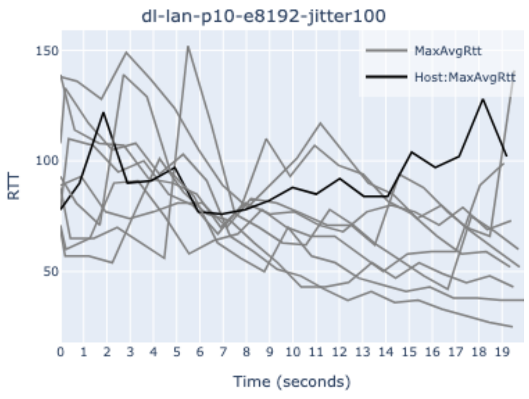
[Jitter1] Investigation of jitter for dl

Figure 6.61: Indeed, the RTT is variable with a maximum of about 100ms and an added local network inherent delay of 20 to 50 milliseconds.

[*go back to first mention*]



(a)



(b)

Figure 6.61: Detail graph: [Jitter1].1

[Jitter2] Investigation of jitter for sl

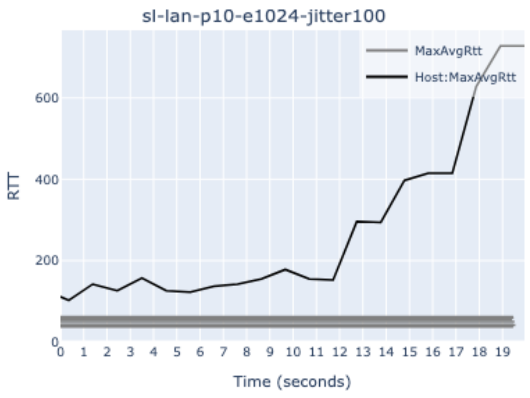


Figure 6.62: Detail graph: [Jitter2].1

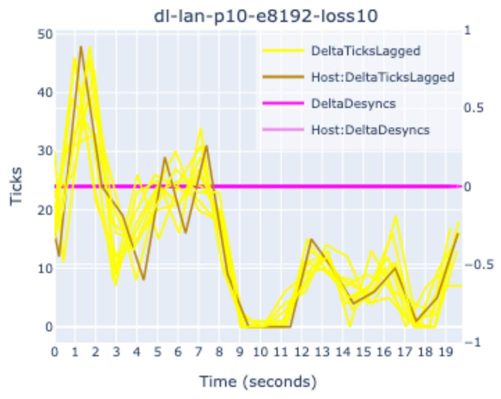
Figure 6.62: A rising host RTT in the end can be observed, but not explained though the available data.

[*go back to first mention*]

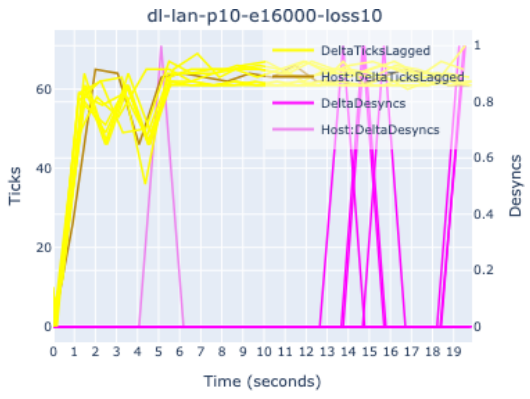
[Loss1] Investigation of loss for dl

Figure 6.63: Again, desyncs are introduced though packet loss, similar to latency. No desync is shown with 8192 entities, which makes this configuration playable with few lags. However, there are multiple desyncs with 16,000 entities, which is definitely above the determined resynchronization limit.

[*go back to first mention*]



(a)



(b)

Figure 6.63: Detail graph: [Loss1].1

6.4 Conclusion and comparison

After evaluating and discussing the various metrics, configurations and scenarios a conclusion to the scientific question can be formulated. Deterministic lockstep has no indicated vertical scaling limitation with a player count of up to 10 supporting 16,000 or more entities. A horizontal scaling limitation could not be found either under given circumstances and deterministic lockstep is confirmed to work with 40 or more players while handling 1024 entities. However, performance degrades when scaling both dimensions, which demonstrates dependent limits as a negative correlation between entity and player scaling. For instance, a scaling configuration of 40 players and 4096 entities or 30 players and 8192 entities was not possible. The projected scaling graph therefore can be depicted by figure 6.64 and assigned to category (c) according to figure 1.2.

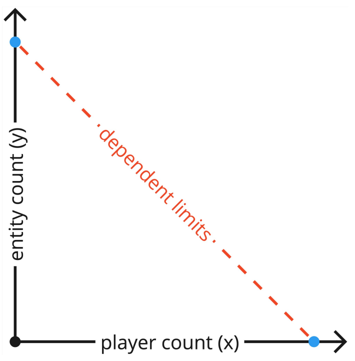


Figure 6.64: Projected scaling graph through evaluation: category (c): dependent limits with many maximum scaling vectors.

The main reason for performance degradation are simulation pauses induced by lagged ticks, which in turn can be caused by hardware limitations, high RTTs, respectively latency and jitter, or implementation flaws. Regarding hardware limitations, some computers already had low FPS with 8192 entities and 10 players. This confirms a disadvantage of deterministic lockstep: scaling horizontally also increases the probability of slower clients, that degrade game experience for all players.

Unfortunately, the resynchronization functionality as part of the lockstep-snapshot hybrid system did not achieve desired results. In LAN 4096 entities and in WAN only 1024 entities could be resynchronized between a maximum of eight players. This limitation also resulted in latency, jitter and packet loss hindering deterministic lockstep scaling, since they introduced desyncs, that can only be handled in said configurations.

The unoptimized snapshot interpolation implementation achieved a vertical scaling limit of 4096 entities with 10 players and a horizontal scaling limit of 40 or more players with 1024 entities and therefore has a lower entity limit compared to deterministic lockstep. Jitter and packet loss have a negative impact on snapshot interpolation performance, although it is not perceivable through the available data what is the cause. The main problems of snapshot interpolation turned out to be hardware limitations as well, bandwidth bottlenecks, or implementations flaws.

The differences between LAN and WAN evaluations were small. Both network options are therefore viable for either method.

Compared to results of related work from chapter 2, vertical limitations of deterministic lockstep exceed the 1500 entities limit of [14] with 16,000 entities, which presumably is mostly due to hardware advancements, since the paper is over 20 years old. Furthermore, that the method is theoretically unlimited by bandwidth [19] can be confirmed.

Horizontal limitations were found to be higher than 4 to 8 players supporting 40 players, but only with 1024 entities in total, 20 players with up to 8192 entities and 10 players also achieved acceptable results with 16,000 entities. Resynchronization was limited to a maximum of 8 players, though. The theoretical limit of 3227 players proposed by [19] could not be evaluated due to missing computers.

Vertical and horizontal snapshot interpolation limitation numbers of related work could not be reached due to implementation flaws and missing optimization.

Chapter 7

Conclusion

Multiplayer games can increase player enjoyment through social interactions, cooperation and competition. The popularity of such games is shown by current market trends. Especially networked multiplayer games frequently achieve great success, but confront game developers with additional networking challenges in the already complex field of game production. The primary challenge is game state synchronization across all players. Based on the current research, there are three main methods for this task – deterministic lockstep, snapshot interpolation and state-sync – and each of them has its own advantages and disadvantages.

This work quantitatively evaluated and discussed the vertical (entity count) and horizontal (player count) limitations of deterministic lockstep and compared the method to snapshot interpolation in chapter 6. Results showed, that Deterministic lockstep has no indicated vertical scaling limitation with a player count of up to 10 supporting 16,000 or more entities. A horizontal scaling limitation could not be found either and deterministic lockstep was confirmed to work with 40 or more players while handling 1024 entities. However, both scaling dimensions correlate negatively, which was indicated by the maximum scaling configurations 30 players and 4096 entities or 20 players and 8192 entities. The major reasons for these limitations were attributed to either hardware, high round-trip times (RTTs), respectively latency and jitter, or implementation flaws.

An unoptimized snapshot interpolation implementation achieved a vertical scaling limitation of 4096 entities with 10 players and a horizontal scaling limit of 40 or more players with 1024 entities and therefore was found to have a lower entity limit compared to deterministic lockstep. The main problems of snapshot interpolation turned out to be hardware limitations, bandwidth bottlenecks, or implementations flaws and lack of optimization.

Furthermore, the results were compared to related work from chapter 2 and new practical vertical and horizontal limitations of deterministic lockstep could be determined, whereas proposed theoretical limits could not be reached. Vertical and horizontal snapshot interpolation limitation numbers of related work could not be raised due to said implementation flaws and missing optimization.

A resynchronization functionality as part of a lockstep-snapshot hybrid system did not achieve desired results. In a local area network (LAN) 4096 entities and in a wide area network (WAN) only 1024 entities could be resynchronized between at most eight players. Therefore, a hybrid system, that changes the main synchronization method based on game and network conditions is conceivable and would theoretically work based on the proposed model and evaluation results, but, before implementing and evaluating such system in its entirety, the snapshot interpolation implemention would have to be optimized first.

Other contributions of this thesis included an overview of game networks and the three game state synchronization techniques in chapter 3 and 4. An architecture and implementation model for deterministic lockstep including a hybrid approach combining it with snapshot interpolation for re-synchronization and hot-joins in chapter 5. And finally, a network packet deconstruction of the implemented networking framework Unity Transport Package (UTP) covered in chapter 6.

Future work can expand the evaluation of this work with more computers to find certain higher horizontal limitations. Furthermore, the snapshot interpolation implementation could be improved with snapshot compression and other optimizations in order to re-evaluate its limits and compare them to deterministic lockstep. This would also improve the re-synchronization process performance, that was proposed in chapter 5 as part of a lockstep-snapshot hybrid model. This model could be implemented and evaluated in its entirety, given the snapshot interpolation system can be improved. Based on the deterministic lockstep implementation additional topics like hot-joins, host migration and dynamic turn durations could be evaluated and discussed as well. Finally, an application of the deterministic lockstep implementation model into a consumer-ready game will yield valuable further quantitative and, additionally, qualitative evaluation results.

References

[1] Joshua Glazer and Sanjay Madhav. *Multiplayer Game Programming: Architecting Networked Games*. Addison-Wesley Professional, 2015.

[2] Blake Bryant and Hossein Saiedian. “An evaluation of videogame network architecture performance and security”. In: *Computer Networks* 192 (2021), pp. 108–128.

[3] Glenn Fiedler. *Networking for Physics Programmers*. `https://www.gdcvault.com/play/1022195/Physics-for-Game-Programmers-Networking`. [Online; accessed 16-January-2024]. 2015.

[4] Glenn Fiedler. *Deterministic Lockstep*. `https://gafferongames.com/post/deterministic_lockstep/`. [Online; accessed 16-January-2024]. 2014.

[5] Ruoyu Sun. *Game Networking Demystified*. `https://ruoyusun.com/2019/03/28/game-networking-1.html`. [Online; accessed 16-January-2024]. 2019.

[6] Glenn Fiedler. *Snapshot Interpolation*. `https://gafferongames.com/post/snapshot_interpolation/`. [Online; accessed 16-January-2024]. 2014.

[7] Glenn Fiedler. *State Synchronization*. `https://gafferongames.com/post/state_synchronization/`. [Online; accessed 16-January-2024]. 2015.

[8] Kathy A. Mills Bessie G. Stone and Beth Saggers. “Online multiplayer games for the social interactions of children with autism spectrum disorder: a resource for inclusive education”. In: *International Journal of Inclusive Education* 23.2 (2019), pp. 209–228.

[9] Thorsten Quandt and Sonja Kröger. *Multiplayer: The social aspects of digital gaming*. Routledge, 2013.

[10] Helena Cole and Mark D. Griffiths. “Social Interactions in Massively Multiplayer Online Role-Playing Gamers”. In: *CyberPsychology & Behavior* 10.4 (2007), pp. 575–583.

[11] pwc. *Perspectives from the Global Entertainment & Media Outlook 2023–2027.* `https://www.pwc.com/gx/en/industries/entertainment-media/outlook/downloads/PwC-GEMO-2023-PDF_V07.0_Accessible.pdf`. [Online; accessed 24-January-2024]. 2023.

[12] Motion Picture Association. *Theme Report 2021.* `https://www.motionpictures.org/wp-content/uploads/2022/03/MPA-2021-THEME-Report-FINAL.pdf`. [Online; accessed 24-January-2024]. 2022.

[13] J. Clement. *Most played games on Steam in 2023, by hourly average number of players.* `https://www.statista.com/statistics/656319/steam-most-played-games-average-player-per-hour/`. [Online; accessed 18-January-2024]. 2024.

[14] Paul Bettner and Mark Terrano. "1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond". In: (2001). [Online; accessed 21-January-2024].

[15] Nathan Sledon et al. "The Effect of Latency on User Performance in Warcraft III". In: (2003).

[16] Tom Beigbeder et al. "The Effects of Loss and Latency on User Performance in Unreal Tournament 2003". In: (2004).

[17] Preetam Ghosh et al. "Improving end-to-end quality-of-service in online multiplayer wireless gaming networks". In: *Computer Communications* 31.11 (2008), pp. 2685–2698.

[18] Ahmed Abdelkhalek et al. "Behavior and Performance of Interactive Multi-Player Game Servers". In: *Cluster Computing* 6 (2003), pp. 355–366.

[19] Josip Petanjek. "Next Generation of Networked Games". In: (2023). [Online; accessed 24-January-2024].

[20] Hampus Liljekvist. *Detecting Synchronisation Problems in Networked Lockstep Games.* 2016.

[21] Yuan Gao. *Netcode Concepts.* `https://meseta.medium.com/netcode-concepts-part-1-introduction-ec5763fe458c`. [Online; accessed 26-January-2024]. 2018.

[22] Glenn Fiedler. *UDP vs. TCP.* `https://gafferongames.com/post/udp_vs_tcp/`. [Online; accessed 27-January-2024]. 2008.

[23] Unity. *Pipeline use.* `https://docs-multiplayer.unity3d.com/transport/current/pipelines/`. [Online; accessed 27-January-2024]. 2023.

[24] Petroc Taylor. *Average mobile and fixed broadband download and upload speeds worldwide as of April 2023.* https://www.statista.com/statistics/896779/average-mobile-fixed-broadband-download-upload-speeds/. [Online; accessed 27-January-2024]. 2023.

[25] Gabriel Gambetta. *Fast-Paced Multiplayer.* https://www.gabrielgambetta.com/client-server-game-architecture.html. [Online; accessed 27-January-2024].

[26] Glenn Fiedler. *Snapshot Compression.* https://gafferongames.com/post/snapshot_compression/. [Online; accessed 28-January-2024]. 2015.

[27] Forrest Smith. *Synchronous RTS Engines and a Tale of Desyncs.* https://www.forrestthewoods.com/blog/synchronous_rts_engines_and_a_tale_of_desyncs/. [Online; accessed 31-January-2024]. 2011.

[28] David Monniaux. "The pitfalls of verifying floating-point computations". In: *ACM Transactions on Programming Languages and Systems* 30.3 (2008), pp. 1–41.

[29] Edward Rowe. *Generating Predictable Random Numbers in Unity.* https://blog.redbluegames.com/generating-predictable-random-numbers-in-unity-c97b7c4895ec. [Online; accessed 01-February-2024]. 2020.

[30] Unity. *About Unity Transport.* https://docs-multiplayer.unity3d.com/transport/current/about/. [Online; accessed 03-February-2024]. 2023.

[31] Unity. *Namespace Unity.Networking.Transport.* https://docs.unity3d.com/Packages/com.unity.transport@2.2/api/Unity.Networking.Transport.html. [Online; accessed 05-February-2024].

[32] Microsoft. *Serialization in .NET.* https://learn.microsoft.com/en-us/dotnet/standard/serialization/. [Online; accessed 05-February-2024].

[33] Glenn Fiedler. *What Every Programmer Needs To Know About Game Networking.* https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/. [Online; accessed 15-February-2024]. 2010.

[34] Wireshark. *Wireshark - The world's most popular network protocol analyzer.* https://www.wireshark.org/. [Online; accessed 05-February-2024].

[35] *Qualcomm Atheros QCA61x4A.* https://oemdrivers.com/network-qualcomm-atheros-qca61x4a-wireless. [Online; accessed 22-February-2024].

[36] *IEEE 802.11ac-2013.* https://standards.ieee.org/ieee/802.11ac/4473/. [Online; accessed 22-February-2024].

[37] Apple. *MacBook Pro (16", 2021) - Technical Specifications.* `https://support.apple.com/kb/SP858?locale=de_DE`. [Online; accessed 22-February-2024].

[38] Apple. *MacBook Pro Wi-Fi specification details.* `https://support.apple.com/en-gb/guide/deployment/dep2ac3e3b51/web`. [Online; accessed 22-February-2024].

[39] *IEEE 802.11ax-2021.* `https://standards.ieee.org/ieee/802.11ax/7180/`. [Online; accessed 22-February-2024].

[40] *Vodafone Station [with] Wi-Fi 6.* `https://www.vdsl-tarifvergleich.de/vdsl-hardware/all/vodafone-station-mit-wi-fi-6/`. [Online; accessed 22-February-2024].

[41] Unity. *Relay locations and regions.* `https://docs.unity.com/ugs/manual/relay/manual/locations-and-regions`. [Online; accessed 23-February-2024].

[42] Unity. *Unity.* `https://unity.com/`. [Online; accessed 04-February-2024].

[43] Unity. *Overview of services.* `https://docs.unity.com/ugs/en-us/manual/overview/manual/unity-gaming-services-home`. [Online; accessed 05-February-2024].

Acronyms

LAN	local area network. 7, 15, 19, 32, 33, 39, 52–54, 62, 70, 87, 89, 96
WAN	wide area network. 7, 15, 32, 33, 39, 48, 52–54, 62, 82, 87, 89, 96
Tx	transmit. 16, 96
Rx	receive. 16, 96
Dx	distribute. 16, 96
Bx	broadcast. 16, 97
NID	network ID. 34, 41, 43, 97
RTT	round-trip time. 16, 19, 24, 37, 39, 40, 42, 50, 55–57, 72, 86, 88, 97
UTP	Unity Transport Package. 2, 3, 11, 30, 32, 33, 44–50, 89, 98
QoS	quality of service. 12, 13
RTS	real-time strategy game. 12–15, 28
FPS	first-person shooter game. 13, 15, 28
FPS	frames per second. 50, 52, 54, 55, 65–69, 87
P2P	peer-to-peer. 17, 18, 23, 28, 99
NAT	network address translation. 18
STUN	session traversal utilities for NAT. 18
TURN	traversal using relays around NAT. 18
OSI	open systems interconnection model. 19, 29–31, 99
UDP	user datagram protocol. 19, 30, 32, 45, 46, 98
TCP	transmission control protocol. 19, 30, 32

NPC non-playable character. 20

UTC universal time coordinated. 23

UI user interface. 24

PRNG pseudo random number generator. 25, 26, 41

FOW fog-of-war. 28

IP internet protocol. 30, 45, 46

API application programming interface. 30

MTU maximum transmission unit. 32

UGS Unity Gaming Services. 33, 39, 43, 50, 98

ASCII American Standard Code for Information Interchange. 46

CSV comma-separated values. 50

V-Sync vertical synchronization. 65

Glossary

local multiplayer game A non-networked multiplayer game played on one computer. 7, 15

networked multiplayer game A networked multiplayer game played on multiple computers. 2, 7–11, 15, 20, 48, 88

locally networked multiplayer game A networked multiplayer game played on multiple computers in a local area network (LAN). 7, 15

online multiplayer game A networked multiplayer game played on multiple computers in a wide area network (WAN). 7, 15, 33

server A headless version of the game (without visual output) handling multiplayer communication and synchronization. 13, 15–18, 20, 21, 23, 28, 32–35, 37, 96, 97

client A game instance of a player sending commands and receiving data from another client, server or host. 7, 8, 15–26, 28, 32–35, 37, 39–43, 45–50, 52, 53, 58–60, 65–70, 87, 96–98

host A combination of server and client on one computer handling both tasks. 13, 16–18, 20, 21, 23, 26, 28, 32–35, 37, 39–43, 45–50, 58, 60, 65–68, 70, 89, 96–98

authoritative Ownership and source of truth for game states. 16, 17, 20, 21, 28

network packet Data packaged together sent over a network. 2, 7, 11, 13, 16, 19, 30, 44, 45, 49, 51, 89, 96–98

transmit Send a network packet to one other client (Tx). 16, 19–21, 23, 30, 33, 34, 36, 39, 44, 45, 47, 51, 55, 57, 59, 94, 97, 98

receive Get a network packet (Rx). 16, 20, 23, 32–37, 39–41, 51, 58, 94, 98

distribute Send a network packet received from a client (origin) to all other clients excl. the origin (Dx). 16, 21, 23, 34, 40, 42, 43, 94

broadcast Send a network packet to all other clients (Bx). 16, 20, 21, 23, 34, 37, 39, 41, 94, 98

network message Data transmitted between clients and server or host. 16, 19, 30, 32–35, 37, 38, 44, 46, 48, 97, 99

netcode Networking related code. 26, 30

input A physical player interaction (e.g. click, key press). 7–9, 12–14, 20, 21, 24, 97

command A player input that translates to a game logic relevant event. 20, 21, 23–25, 28, 30, 34–37, 39–41, 48, 49, 52, 57, 74, 96, 98

game entity A game object whose state may be synchronized. 7, 9, 10, 12, 13, 20–22, 34, 41, 43, 97

game state A definitive representation of a game. 2, 7, 11, 16, 17, 20, 21, 23–28, 30, 35, 40–42, 49, 82, 88, 89, 96–99, 102

snapshot The game state at a specific time. 7, 8, 10, 11, 14, 20–22, 26, 28, 30, 34–36, 41–44, 48, 49, 52, 55, 57, 58, 61, 77, 89, 97, 98

checksum A value generated based on data, that is used to determine data integrity. 26, 33, 36, 40, 41

network ID (NID) Network ID (NID), a unique, synchronized ID across all clients and servers. Game entities and connections, respectively players, all have a NID. 34

SVC SerViCe, a service related network message. 33–35, 37, 39–42

CMD CoMmanD, a game logic related network message. 35, 37, 39–41, 48

SNP SNaPshot, a network message containing a snapshot. 35, 37

bandwidth Network bandwidth is the maximum data transfer capacity of a connection in a certain amount of time (usually per second). 9, 13, 14, 16, 17, 19, 21, 22, 28, 43, 59, 60, 78, 87, 88

RTT Round-trip time (RTT), time it takes a network packet from its origin to the destination and back. 16, 19, 24, 97

latency The delay of information exchange due to the time network packets need to reach their destination (RTT). 7, 9, 12, 13, 17–19, 21, 28, 32, 33, 39, 53, 62, 63, 83, 85, 87, 88, 98

jitter Variation of latency over time. 7, 9, 19, 32, 33, 40, 42, 53, 63, 87, 88

packet loss An undesired event in which network packets do not reach their destination. 7, 9, 19, 32, 33, 42, 53, 64, 85, 87

deterministic lockstep A multiplayer game state synchronization method, where each client runs a deterministic simulation in lockstep (synchronized timing). Only commands are transmitted by clients and broadcast by the host. 2, 3, 7–14, 20, 21, 23, 26, 28–30, 34–36, 43, 48, 49, 52–54, 58, 61, 62, 68, 81, 86–89, 98

snapshot interpolation A multiplayer game state synchronization method, where only the host runs the simulation, receives commands and broadcasts snapshots to all clients. 2, 3, 7–11, 13, 20, 21, 28, 43, 49, 52–54, 56–59, 61–64, 87–89, 98

state-sync A multiplayer game state synchronization method, that combines the deterministic lockstep with the snapshot interpolation method. 2, 3, 7–9, 12, 20–22, 24, 28, 88

turn Time in which the player may issue commands scheduled for a later turn. 12, 13, 23, 24, 34, 36, 37, 39–43, 52, 60, 72, 83, 89, 98

tick Fixed subdivisions of a turn in which the simulation runs. 24, 37, 39, 41, 52, 60–63, 86

hot-join A feature, where players are able to join an already started multiplayer game session. 2, 11, 26, 28, 36, 42, 89

Unity A cross-platform game engine [42]. 32, 34, 98

UTP Unity Transport Package (UTP), a low-level user datagram protocol (UDP) networking framework for the Unity game engine [30]. 11, 30

Unity Gaming Services (UGS) Unity Gaming Services (UGS) "is an end-to-end platform that is designed to help [developers] build, engage, and grow [a] game" [43]. 33

List of Figures

List of Tables

List of Algorithms