Deterministic Lockstep in Networked Games

Paul Mieschke
Hochschule der Medien Stuttgart
Stuttgart Media University
Stuttgart, Germany
Nobelstraße 10, 70569 Stuttgart
mail@pmieschke.com

***Abstract*—Multiplayer games can increase player enjoyment through social interactions, cooperation, and competition. Their market popularity shows the success of especially networked multiplayer games, which pose new networking challenges to game developers. The main challenge is synchronizing game state across players. Research identifies deterministic lockstep, snapshot interpolation, and state-sync as primary methods for this task, each with distinct advantages and disadvantages.**

This work, and the master thesis this paper is based on, quantitatively evaluated deterministic lockstep, demonstrating its vertical (entity count) and horizontal (player count) scaling limitations and compares the method to snapshot interpolation. Lockstep supports minimum 16,000 entities for up to 10 players and a horizontal scaling of 40 or more players with 1024 entities. However, a negative correlation between entity and player count limits was observed, which was indicated by the maximum scaling configurations 30 players with 4096 entities or 20 players with 8192 entities. Snapshot interpolation faced a vertical limit with 4096 entities and 10 players and horizontally with 40 or more players and 1024 entities.

The paper further contributes by comparing results to related work, summarizing synchronization methods, proposing a hybrid architecture model of deterministic lockstep with snapshot interpolation for re-synchronization and hot-joins, and deconstructing Unity Transport Package's (UTP) network packets.

***Index Terms*—games, networks, multiplayer, deterministic lockstep**

I. Introduction

Since the inception of video gaming, multiplayer games have made it possible to increase player enjoyment through social interactions, cooperation and competition [1], [2]. "Pong", a local multiplayer game, requires players to be physically present in the same location. However, networked multiplayer games allow interaction over global distances, leveraging multiple computers connected via local area networks (LANs) and wide area networks (WANs) like the internet. This introduces networking challenges like latency, jitter or packet loss, but in particular, game state synchronization across different devices to ensure a cohesive player experience.

Synchronizing the game state in networked multiplayer games is a complex task due to the inherent difficulties of maintaining consistency across distributed systems. Based on the current research, there are three primary methods to address this challenge: deterministic lockstep, snapshot interpolation, and state-sync [3]–[5]. Deterministic lockstep relies on all clients running a synchronized simulation and only player inputs are shared, leading to low network traffic but requiring the game to run deterministically [6], [7]. With snapshot interpolation, the server or host sends periodic snapshots of the game state to clients, who then interpolate between these snapshots to achieve smooth gameplay, at the cost of potentially higher network traffic [7], [8]. State-sync combines deterministic lockstep and snapshot interpolation, sharing both inputs and snapshots. This method, however, comes with a high implementation effort [7], [9].

The choice among these synchronization methods depends on the specific requirements of a game genre. For instance, strategy games, which often involve managing a large number of entities, may benefit from the efficiency of deterministic lockstep, whereas fast-paced action games might prefer lower-latency methods like snapshot interpolation or state-sync. Beyond entertainment, networked synchronization methods have applications in distributed simulations and serious games, indicating their broader relevance [10].

The popularity of multiplayer games is shown by their significant share in the global games market. The most played games on platforms like Steam often feature multiplayer modes, highlighting the enduring appeal of shared gaming experiences [11], [12].

In conclusion, the development of networked multiplayer games presents unique challenges, particularly in game state synchronization. By carefully selecting among deterministic lockstep, snapshot interpolation, and state-sync based on game requirements, developers can enhance player engagement and capitalize on the growing market for multiplayer gaming experiences.

A. Scientific question

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 [4]. 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 [5] to eight [13] 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. $\vec{s}$ 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: $\max|\vec{s}|$. In summary, the scientific question can be formulated as: "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. 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, negating disadvantages of each other method.

B. Contributions

This paper, respectively the master thesis it is based on, contributes a summary of synchronization methods, a proposal of a hybrid architecture model of deterministic lockstep with snapshot interpolation for re-synchronization and hot-joins, a deconstruction of Unity Transport Package's (UTP) network packets and an evaluation and discussion of said scientific question.

II. Related work

The frequently cited paper "1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond" [13] 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 the game would have been limited to a maximum of 250 moving units.

While lockstep's vertical scalability is seemingly unlimited by bandwidth [14], horizontal scalability is more constrained, with, according to [4], [5], limits of only four players. [14], however, states a theoretical player limit of 3227.

Snapshot interpolation, used in games like Quake, shows horizontal scalability up to 100 players on a single server, limited more by processor capabilities than bandwidth [3], [15].

Recent advancements in multiplayer game technology are pushing these limits further. Fortnite and Planetside 2 represent the current generation's capabilities, with support for up to 100 and 2000 players. Fortnite can handle 40-50 thousand synchronized entities, although some game fidelity must be sacrificed for optimization, e.g. projectiles are not synchronized [14].

Research on input latency reveals varying tolerances across genres. Strategy games like Age of Empires and Warcraft III may have latencies up to 500 milliseconds without significantly affecting game outcomes [13], [16]. In contrast, fast-paced first-person shooters (FPS) and car racing games require lower latencies, with performance degradation noticeable at delays as low as 75 to 100ms, impacting accuracy and control [17], [18].

III. State synchronization overview

This section conveys an overview about game state synchronization and summarizes the methods deterministic lockstep, snapshot interpolation and state-sync. However, general knowledge in the computer science fields of networking and games is assumed. The corresponding thesis goes into more details.

Game state synchronization in networked multiplayer games is crucial for maintaining a consistent player experience across all clients. To achieve synchronization a definitive representation of the game, the game state, must be shared, that includes game entities like player characters, non-player characters (NPCs), and other game-related metadata like player resources.

With **deterministic lockstep** every client executes the complete game simulation and only player commands are either broadcast to other clients or transmitted to the server or host, who then distribute commands. Therefore, deterministic lockstep can be used with a client-server as well as a peer-to-peer (P2P) topology [3], [6].

In order for this to work deterministic lockstep is based on two principles. The game progresses in *lockstep* between all clients, and the game simulation must be *deterministic*.

Lockstep requires every client to execute the game simulation in unison. Therefore, all commands must be executed at the same designated game times, ensuring that every client's simulation progresses identically. This coordination is achieved through turns, discrete intervals where players issue commands. To maintain synchronization, the simulation pauses until all commands for the next turn are received. The simulation layer includes game logic relevant systems, must stay in-sync across all clients and can be differentiated to the the presentation layer, which contains systems associated with the view, e.g. user interface (UI) or particle effects. While the simulation layer uses fixed update intervals (ticks) and turns to stay in-sync, the presentation layer runs independently. This allows for immediate feedback through animations or sound effects, thus mitigating perceived input delays, that arise from commands being scheduled for later turns (250-500ms delay) [19], [20].

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. *Determinism* means that given the same initial conditions and sequence of inputs, the simulation yields identical outcomes across all clients. This requires careful management of, for example, random number generation, execution order, and data structure ordering to prevent desynchronization. The use of predictable random number generators (PRNGs) with synchronized seeds ensures consistency. Moreover, deterministic execution of game logic and entity updates is crucial. This involves synchronized data structure ordering, e.g. for entity update iterations, and avoidance of floating-point arithmetic, since slight differences of their implementations across varying hardware and compilers can lead to increasing game state discrepancies [6], [7], [13], [21], [22].

Ultimately, human error can still lead to desynchronization. Detecting game state divergence can be achieved with checksums, compact representations of game states, usually a number, compared across clients. When a desynchronization was detected, an agreed-upon definitive game state, typically from the server or host, is distributed to re-synchronize all clients [13], [19], [21].

Snapshot interpolation operates under an authoritative client-server topology, where the server maintains the definitive game state, executing shared player commands and broadcasting state snapshots to clients regularly. Clients act as "dumb terminals", rendering the game state based on received snapshots without performing any game logic simulation. This method requires the use of interpolation techniques, such as Hermite interpolation, to smooth out entity movements between snapshots. To mitigate the high bandwidth demand, quantization and delta compression can be employed, optimizing snapshot data size [3], [4], [8], [23].

State-sync combines snapshot interpolation and deterministic lockstep, allowing clients to not only receive state snapshots from the server but also simulate the game state locally. This method also involves transmitting player commands from clients and applies a client-server topology. Latency compensation techniques, such as reconciliation and rollback, adjust the game state based on newly received snapshots, ensuring consistency despite network delays. A priority accumulator queue further optimizes bandwidth usage by prioritizing the transmission of important game entities within snapshots, thereby balancing network load and game fidelity [7], [9], [24].

After covering three game state synchronization methods, they can be summarized visually in figure 1 and by category in table I:

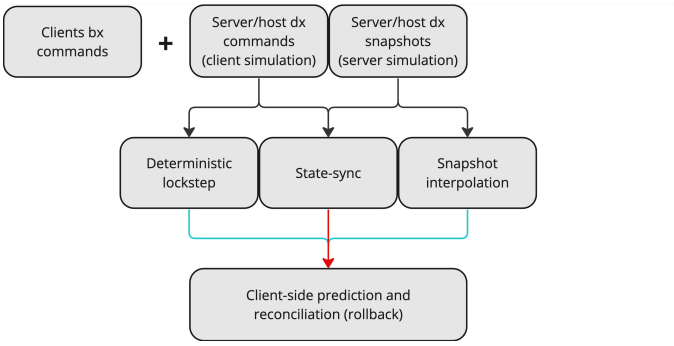

Fig. 1. Game state synchronization summary. The red arrow means "uses", and the blue arrow means "could use, but is not part of a default implementation".

TABLE I
GAME STATE SYNCHRONIZATION SUMMARY BASED ON CATEGORIES.

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 fog-of-war (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. real-time strategy (RTS) or turn-based games

IV. IMPLEMENTATION

A. OSI model integration

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 2. 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.

The foundational layers (one to three) comply their conventional roles, utilizing default protocols for packet transfer. At the transport layer (four), UDP is preferred due to its suitability for real-time game data transmission. UDP is further supported by the choice of the Unity Transport Package (UTP) as the underlying networking framework, that resides in layers five to seven.

Layer eight, designated as "netcode", is a bridge between low-level networking and game logic. This layer abstracts away networking complexity by providing APIs for session management and command transmission. Building on top of that, one synchronization system must be used, that keeps game states in-sync. 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.

The final layer nine contains the game logic, where gameplay mechanics and high-level networking logic converge.

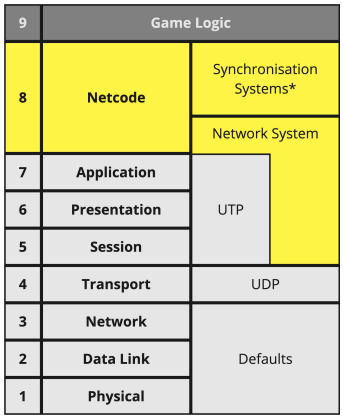

Fig. 2. OSI model integration. Layer 1-7: OSI layers and their employed implementations. Layer 8-9: custom layers and implementations developed on top of the OSI model [25]. * e.g. snapshot system, lockstep system, ...

B. Architecture model

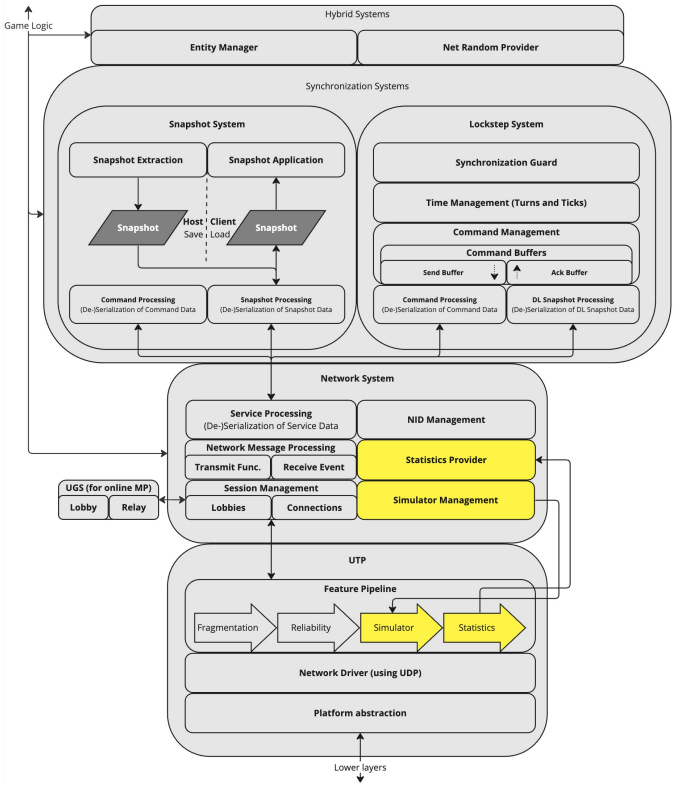

Fig. 3. Implementation architecture model. The integrated OSI model layers five to eight and partially nine are shown in more detail from bottom to top. Each component builds on top of lower components. Yellow highlighted components may be disabled in production for a performance increase.

The implementation architecture model is illustrated in figure 3. Unity Transport Package (UTP, OSI layers 5-7) is a low-level networking framework supporting client-server/host topologies and is developed for the Unity game engine. It provides convenient platform abstraction offering cross-platform, UDP socket-based network drivers [25]. Pipelines can be used to extend UDP with features like fragmentation, reliability and sequencing. Fragmentation ensures packets stay within the maximum transmission unit (MTU) limit, while the reliability stage implements a TCP-like acknowledgment system for guaranteed data transmission. Sequencing is part of the reliable stage and maintains the order of packets. There is also an unreliable sequenced stage when only order is essential. The simulator stage, only used during development, emulates various network conditions. A statistics stage provides upper layers with valuable metrics for analytics [26].

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. This involves establishing and disconnecting connections, alongside lobby management tasks such as hosting, joining, and configuring lobbies. Different procedures are needed for LAN and WAN environments, with WAN connections using the Unity Gaming Services (UGS) for relayed client-server communication and a lobby service. These procedures only differ up until this component. For later components the process is unified regardless of WAN or LAN.

Network message processing is central to transmitting and receiving network messages, involving the serialization and deserialization of message headers. Serialization converts game entity states into a byte format for storage or network transmission, while deserialization reverses this process. The system delegates further processing to specialized components based on the network message type byte identified in the header.

A first specialized processor is the service processing component, which manages service-related messages (header type byte is SVC) for functionalities like connection management, heartbeats, and lobby updates.

Additionally, the network system includes simulator management and statistics provision components for analytical purposes during development, allowing the simulation of non-optimal network conditions and the collection of performance metrics for later evaluation.

The final component is network ID (NID) management, A NID is a unique, synchronized ID across all clients and server. Game entities and connections, respectively players, all have an NID. There are various ID types: *local IDs*, managed by the game engine (Unity [27]), differ across client instances. *Static NIDs*, consistent across clients given the same game build is used, identify unchanging game entities like buildings. *Dynamic NIDs* are for entities spawned during gameplay, requiring a synchronization process across clients. Connection NIDs represent clients, respectively players.

When using a snapshot synchronization method, dynamic NID assignment and synchronization is straight-forward, with

servers or hosts assigning NIDs for new entities and distributing these through snapshots. However, deterministic lockstep involves a more complex process. When clients issue commands to spawn entities, the host registers NIDs before the actual entity spawn, ensuring uniqueness. This NID is then broadcast with the acknowledged command, and upon execution in a later turn, entities are spawned and assigned the NID across clients.

The first synchronization system is the snapshot system. It 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. 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 the service processing component, snapshot processing handles snapshot network messages (SNP), while the commands processing component manages command network messages (CMD).

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.

The ack buffer is a ring buffer and illustrated in figure 4. The ring buffer contains four segments. In practice, segments are represented by a list with dynamic length. 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 $currentTurn \mod segmentCount$, where $segmentCount$ is 4, and is used to access segments stored in an array data structure up to 3 turns ahead with the formula: $[t + (turn - currentTurn) \mod segmentCount]$. The ring is rotated counter-clockwise after every turn, which is done using the formula: $t = t + 1 \mod segmentCount$.

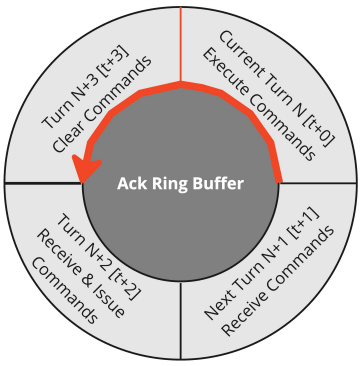

Fig. 4. Ack ring buffer. N is the turn number and t is the ring index. Illustrated in red is the rotation direction of the ring (counter-clockwise).

When receiving a command it is added to the segment $[t + (commandTurn - currentTurn) \mod segmentCount]$. 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, pausing the simulation (tick progression) in case commands for the next turn are missing. The lockstep timing of network messages is clarified in figure 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 depicts an exemplary network message exchange between three clients over time subdivided into turns, where the middle client is 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.

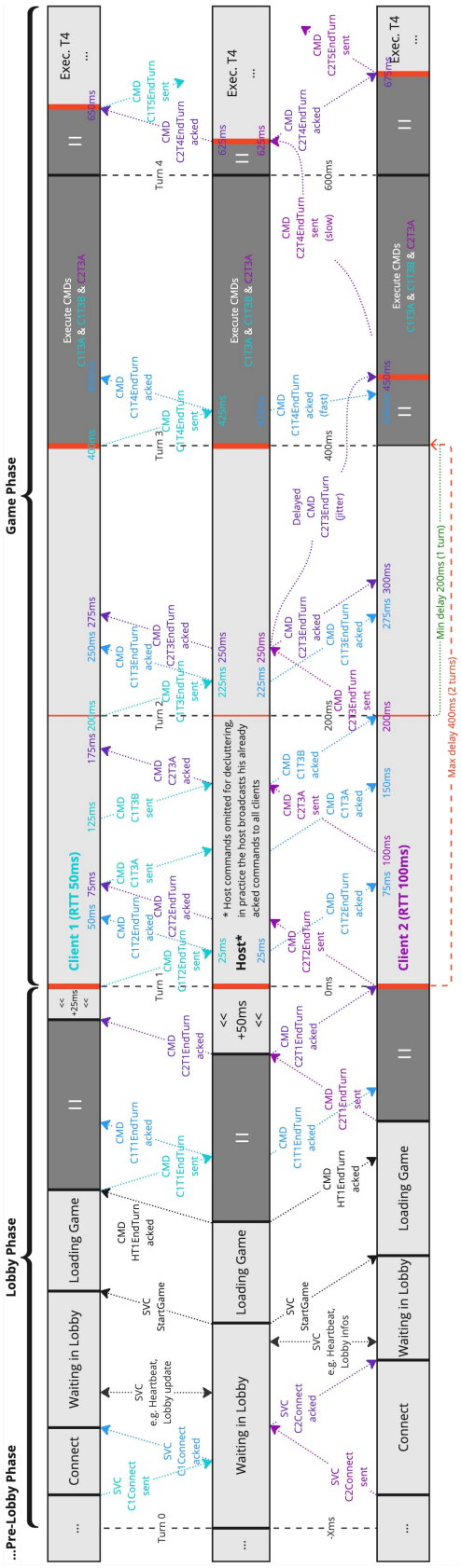

Fig. 5. Lockstep network message timing.

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 broadcast by the host to clients.

The general commands distribution flow is: client sends a CMD to the host $\Longrightarrow$ host adds the command to his ack buffer and acks the CMD (broadcasts the acknowledged command to all clients) $\Longrightarrow$ each client receives the acked CMD and adds the command to their ack buffer.

The figure is further explained in the thesis, which is out-of-scope for this paper.

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.

In the event of a checksum mismatch between host and clients, a resynchronization process is triggered: 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 resynchronization process all commands are ignored. There can be a maximum command loss of two turns. Also, the game will have a delay/lag based on the time the resynchronization process takes.

Hybrid systems bridge the gap between networking and game logic layers, functioning in both singleplayer and multiplayer modes through a unified interface. A cross-platform PRNG ensures determinism across game instance, synchronizing seeds over the network in multiplayer, but also fully operational in singleplayer. Seed initialization occurs during game start or resynchronization. The entity manager handles both static and dynamic game entities. It ensures deterministic execution by updating entities in NID order and is used during serialization and deserialization of snapshots.

C. Hot-join / Reconnect

Contrary to some literature [19], [28], with the implementation model of this work it is possible to support hot-join

and reconnecting to a running game. 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.

D. Deterministic lockstep and snapshot interpolation hybrid

Based on the presented implementation architecture model (figure 3), 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 lockstep for scenes with many entities or when bandwidth is limited.

Switching methods from 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 distributed by the host. This is a fluid transition with no delay. Switching from snapshot interpolation to 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.

V. Evaluation

Based on the implementation of section IV the scientific question of this work is evaluated and discussed. Before that, the underlying UTP networking framework used by the implementation is briefly analyzed with regards to its network packet structure.

A. 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 [29] was used to deconstruct and analyze UTP packets of a localhost loopback connection. Figure 6 captures the connection establishment and heartbeat process and highlights and explains important parts.

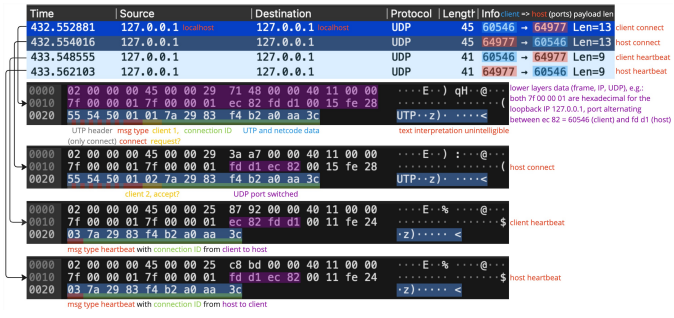

Fig. 6. UTP packet deconstruction: connection process and heartbeats.

Figure 7 demonstrates a reconnection sequence, where client port and connection ID is changed. Figure 8 shows data exchange using the reliable pipeline.

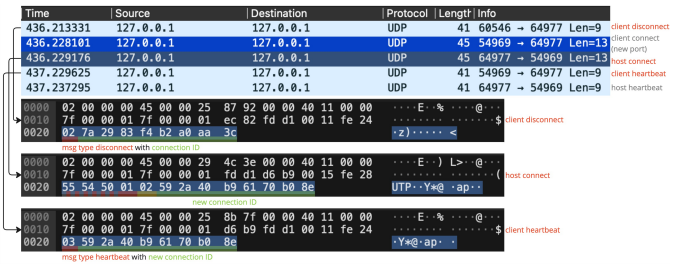

Fig. 7. UTP packet deconstruction: reconnection process.

Fig. 8. UTP packet deconstruction: sent data.

Lastly, figure 9 illustrates the functionality of the reliable pipeline, particularly the handling of packet acknowledgments during a temporary client disconnection.

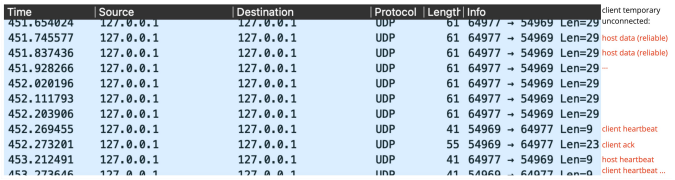

Fig. 9. UTP packet deconstruction: reliable pipeline.

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`).

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.

B. Evaluation setup

In order to quantitatively evaluate the scientific 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 said reasons.

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 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 and 16,000 entities require 1.3MB.

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 [30], [31].

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 [32], [33], [34], [31].

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 [35]. 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 [36].

3) Metrics: The statistics provider component of each client collects and writes certain metrics into a 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: *RelTime/Time* (time since evaluation start), *FpsSmoothed* (smoothed frames-per-second), *MaxAvgRtt* (maximum RTT of all connections of the client), *DeltaPacketsTx* (sent packets since last sample), *DeltaPacketsRx* (received packets since last sample), *DeltaPacketsTxRe* (re-sent packets since last sample), *DeltaPacketsRxRe* (duplicate packets received since last sample), *DeltaPacketsLost* (lost packets since last sample), *DeltaBytesTx* (sent bytes since last sample), *DeltaBytesRx* (received bytes since last sample), *DeltaBandwidth* (sent and received bytes since last sample), *DeltaTicksLagged* (lagged ticks since last sample), *DeltaDesyncs* (detected desyncs since last sample).

4) Process: Prior to the actual evaluation, hardware performance limits for a playable game standard of 30 FPS were determined. Tests revealed Windows machines could support up to 16,000 entities running five instances, while Macs handled up to 32,000 entities. This assessment, excluding multiplayer synchronization, focused on finding CPU and GPU limits for entity processing and rendering.

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 chosen 1-second snapshot interval, while not ideal for production games requiring below 100ms intervals, is justified by a lack of snapshot optimization, like compression. The DeltaBandwidth metric yields similar bandwidth usage between rare but large snapshots used by this implementation compared to a frequently sent and optimized version presented by [23].

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. Additionally, the 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.

C. Results

In the thesis, every metric is evaluated to discuss vertical and horizontal limitations in LAN and WAN. However, this in-depth analysis of all the collected data would be too long for this paper. Thereby, the paper continues directly with a comprehensive conclusion of the evaluation and discussion.

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 10.

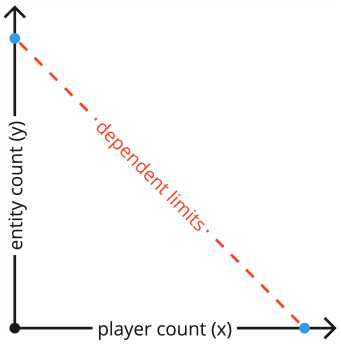

Fig. 10. Projected scaling graph through evaluation: dependent limits with many maximum scaling vectors.

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.

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

Compared to results of related work from chapter II, vertical limitations of deterministic lockstep exceed the 1500 entities limit of [13] 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 [14] 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 [14] 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.

VI. Conclusion

Multiplayer games can increase player enjoyment through social interactions, cooperation and competition. They present developers with the challenge of game state synchronization. This work evaluated deterministic lockstep and snapshot interpolation, identifying their vertical (entity count) and horizontal (player count) scaling limitations and comparing them. Lockstep was found to have no indicated vertical scaling limitation with a player count of up to 10 supporting 16,000 or more entities. Horizontally, 40 or more players were possible while handling 1024 entities. However, a negative correlation between entity and player count limitations was observed, suggesting hardware or implementation flaws as potential limiting factors.

Snapshot interpolation demonstrated similar player but lower entity limits, influenced by hardware limitations, bandwidth constraints and implementation flaws. The comparison of these methods with existing literature revealed new practical limitations for lockstep.

The evaluation of resynchronization as part of a hybrid lockstep-snapshot system showed limited success, indicating the need for further optimization before further tests of such a system can be conducted.

This work also contributed by providing an overview of game state synchronization, an architecture model for deterministic lockstep, including a hybrid approach with snapshot interpolation for re-synchronization and hot-joins, and a deconstruction of Unity Transport Package (UTP) network packets.

Future work could expand the evaluation of this work with more computers to find certain higher horizontal limitations. Snapshot interpolation should be optimized and re-evaluated incorporating compression, also leading to the fully implemented and tested hybrid synchronization model. 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] Thorsten Quandt and Sonja Kröger. *Multiplayer: The social aspects of digital gaming*. Routledge, 2013.

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

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

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

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

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

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

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

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

[10] 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.

[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] 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.

[13] 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].

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

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

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

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

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

[19] 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.

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

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

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

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

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

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

[26] 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].

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

[28] 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.

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

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

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

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

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

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

[35] *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].

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

Note: some sections of the paper were written with the assistance of a large language model (LLM). The LLM was input exclusively with text from the thesis, tasked to summarize this text, and its output was thoroughly validated, improved and incorporated into existing sections manually.