Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Hat Hunters Online

Bachelor's Thesis

Jan Nino Walter

jwalter@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

March 17, 2019

Acknowledgements

I want to thank my supervisor Manuel Eichelberger for all of his support during this work. The ideas from our weekly meetings helped me through a lot of problems and I am really grateful for his help. I also want to thank Nicholas Ingulfsen for creating the game this work extends, helping me out when I had questions regarding the game and testing the game together with me. At last, I am also grateful to my friends and family for testing the game and their support in general.

Abstract

This Bachelor's Thesis describes the online multiplayer mode of the real-time multiplayer game *Hat Hunters* [1]. The online mode allows to play the game at different locations over the internet. This thesis shows how the game state is synchronized among players and explains the techniques applied to solve network problems that arise when making an online mode. The core of the game is playable in online multiplayer and provides a smooth game experience

Contents

Chapter 1

Introduction

Playing a game together with friends is always fun. But it is not always easy to meet up and play a round, whether you live far apart from each other or there is too little time to gather in one place or it is simply too inconvenient to meet just to play a quick game. Thanks to current technology, physically meeting up is not necessary anymore. A lot of games offer online multiplayer, this allows to play a game together even though the players are far away from each other. Online multiplayer has another advantage, the game can let players play together who have never met before. This lets a player play a game even though she does not have anyone to play with right now. The objective of this work is to add an network based online mode to the existing local multiplayer game Hat Hunters created by Nicholas Ingulfsen. Adding an online mode makes it easier for new players to try the game out, which hopefully leads to more players playing the game. As a single match in Hat Hunters is rather quick, only lasting a few minutes, this game could particularly benefit from an online mode.

Online games face a fundamental problem. Every action is always delayed by the time a message needs to cover the distance between players. This delay cannot be circumvented. In practice, developers apply tricks to create the illusion that actions of players do happen immediately while keeping the game state consistent. This work presents the structure of the current online mode of Hat Hunters and explains the techniques applied to generate a responsive game experience.

1.1 Related Works

This work extends the game Hat Hunters made by Nicholas Ingulfsen, which is described in more detail in his thesis [2]. A short summary on the game is given in the following Section 1.2. This work extends the game from the current local multiplayer situated in one physical location to online multiplayer, where players are in separated physical locations. This is illustrated in Figure 1.1.

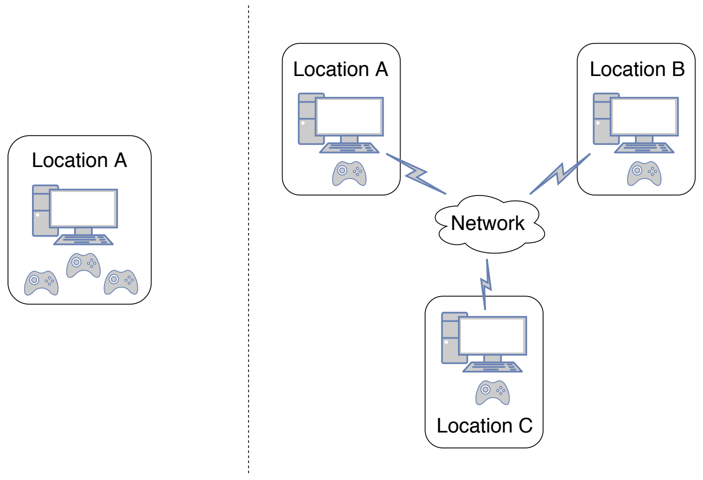


Figure 1.1: Game with local multiplayer on the left compared to online multiplayer to the right.

The theoretical background to implement online multiplayer in a game stems from the book *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games* [3]. The implementation of the online mode requires a time synchronization. This time synchronization is similar to the Network Time Protocol (NTP) [4]. It works in the environment of the game and takes several measures to improve stability.

1.2 Background

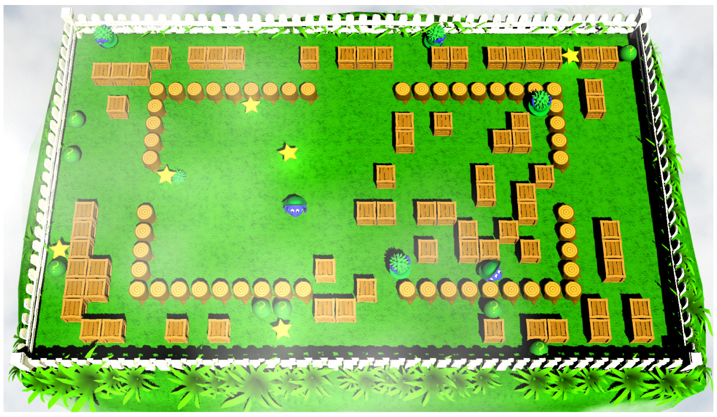


Figure 1.2: The running game

Hat Hunters is a local multiplayer game for 2 to 4 players. Each player controls a character, called a Hat Hunter. In addition to the player controlled characters, there are Artificial Intelligence (AI) controlled characters. All these characters look identical. The core idea of the game is, that the human players try to hide their character among AI controlled characters and at the same time try to figure out which characters are actually controlled by human beings. Figure 1.2 shows what the running game looks like, while Figure 1.3 shows important game elements.

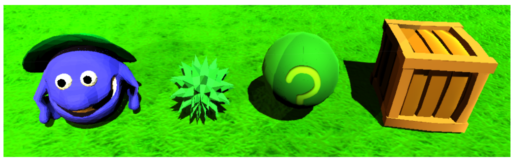


Figure 1.3: From left to right: character, green projectile, item container, box.

To win the game, players collect stars. These stars spawn in the game field at random locations. A player who discovered the identity of another player can use items, which can be obtained from item containers, to attack this player. Depending on the item, the player can steal this player's stars or get other advantages. One of these items is the green projectile. A player that obtained this item can shoot it. It bounces off obstacles and if it hits a character, this player loses a star. This star moves around the field and belongs to the first player that collects it. Players can also place boxes. A box cannot be passed, like stumps, so they can be used to block other players. In contrast to stumps they can however be destroyed by various items, for example by a green projectile. Stumps on the other hand are indestructible and remain at the same position during the whole game. The player who first collected a specified amount of stars wins the game. AI controlled players cannot win the game, no matter how many stars they collect. They are used to allow the player to hide, not to compete with. A player who gets close to the required number of stars changes to her personal hat, which allows other players to identify and attack him to try to prevent the player from winning.

CHAPTER 2

Network

2.1 Network Architecture

We were deciding between using a peer-to-peer and a client-server architecture. Both architectures are described below. Section 2.1.3 describes the decision to choose a client-server solution.

2.1.1 Client-Server

The core idea of this model is that the server has full control over the game state. Independent of the architecture, the game states of clients get out of synchronization due to the latency of every message, like described in the first section of Chapter 4. In this architecture, the server has the real state of the game, and the clients adapt their states to it. The server only manages the game, all players participate from clients. Clients do not communicate with each other directly, they only communicate with the server. If a client for example wants to perform an action, it sends a message to the server. The server then processes this action and sends the result to all clients. This is illustrated in Figure 2.1. The actual implementation for performing such actions in this game is explained in Chapter 4. This architecture is the one used by most games [3, pages 16, 84].

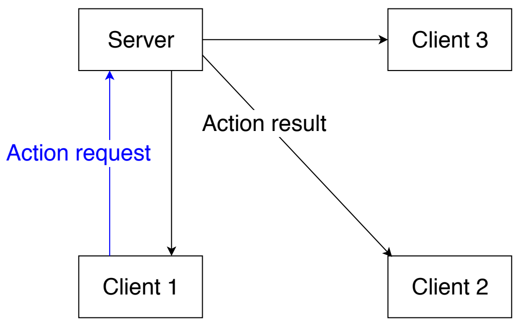


Figure 2.1: Client 1 wants to perform an action: It sends an action request to the server, which processes it and sends the result to all clients, including Client 1.

This solution requires a server, but where exactly should this server run? One possibility is to provide one or multiple dedicated servers which accept players and manage their games. One drawback is that this limits the number of games that can be played at the same time. Another drawback is the resources required for running such a server. That is why we decided to run the server on one of the clients. We will call the client running the server the host. The host runs the server, but its client implementation is exactly the same as the other clients. This approach has another advantage. It scales better, because every game has its own server and is thus independent of other games. But this comes at its cost, as the host now has to do the work for both the client and the server and thus runs the entire game basically twice. It is shown in Section 5.1 that this approach is nevertheless feasible, since the computationally expensive part of the game is the graphics, which need to be rendered only for the client, but not for the server.

The behaviour of the AI players is determined by the server. The server also handles events like spawning new items or stars or setting the character's initial positions. This simplifies synchronization, because it is at any moment of the game clear who is responsible and therefore needs less coordination.

2.1.2 Peer-To-Peer

In this model, there is no single entity controlling the game state. Every peer manages some game elements and has the real state of those elements. Thus, there is no single true state. A peer has for example full control over its own character. It has the character's real state and the other peers adapt their state of this character to the state of the controlling peer. If the controlling peer performs an action, it performs the action immediately and sends a message describing it to the other peers. Figure 2.2 shows this process. This also includes collecting stars, collecting items or getting hit by an item, as the controlling peer is the only one with the real state and thus position of the character. AI players are distributed among peers during the setup of the game. Each peer has full control over all AI players assigned to it. The goal of this distribution is to have varying latencies for AI players to merge them better with the human players. This is important, as hiding among AI players and trying to differentiate between human and AI players are essential features of the game. If all AI players are controlled by one peer, all of their actions would arrive at a second peer with the same latency. But actions from third peer with a significantly different latency could stand out compared to the AI players, thus exposing the third peer player. Spawning items and stars is not that easy to distribute over all peers, as the corresponding spawn rates need to be respected. Thus those tasks are assigned to one peer during the game setup. [5]

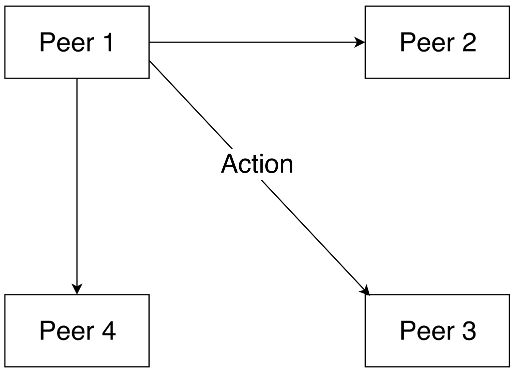


Figure 2.2: Peer 1 wants to perform an action, it performs the action and sends it to all other peers.

2.1.3 Decision

We decided to use the client-server architecture, because this architecture simplifies synchronization, as the server has the complete true game state. It could happen in the peer-to-peer model, that two peers announce that they collected the same star, as they were both close to it. Who will receive the star? All peers would need to run a consensus protocol to converge into the same game state again. This is both a time consuming process and complex to implement. The client-server model does not have this problem in the first place, as the server simply decides which player gets the star and accordingly informs the clients. The peer-to-peer architecture needs more coordination in general, as it needs to distribute AI players over the peers and assign the task of spawning items and stars. In the client-server architecture, the only need is to decide who takes the role of the server. By running the server on the host, the client-server solution scales just like the peer-to-peer solution, one of the latter's big advantages. The load is better distributed among peers in the peer-to-peer architecture, compared to the client-server architecture, where the host has to do a lot more than the other clients.

In the peer-to-peer architecture, a peer sends a message, like a movement message, directly to the other peers. It does not need to send it to the host first, which resends it to the clients. This reduces the latency and thus could lead to a smoother movement of the characters and improve the quality of the game. This however directly leads to another disadvantage of a peer-to-peer architecture. Every peer holds the true state of its character, this makes it easy for any peer to cheat. A peer could simply move its character faster or even change its position to immediately jump to a star. The other peers just adapt their states to these actions, they cannot verify them as they do not have the true state of the character. On the other hand, in the client-server architecture, the server has the whole true game state, thus it can verify the behaviour of all clients and prevent them from cheating. The only exception is the host itself, which can still cheat as it controls the server. Therefore a client only needs to trust the host to ensure a fair game, compared to the peer-to-peer architecture where a peer needs to trust every other peer.

Due to those reasons, we clearly decided to implement the client-server architecture.

2.2 Communication

The game uses Steamworks [6] to communicate between entities. Steamworks consists of a suite of tools which can be used for free if the game is on Steam [7]. More precisely, Steamworks.NET [8] is used, a C# wrapper for Steamworks, as Steamworks officially only supports C++. One of the tools of Steamworks is Steam Networking [9]. It allows to exchange messages between Steam users. As each user will probably be behind a Network Address Translation (NAT), it tries to traverse these NATs. If it does not succeed, it establishes a connection through a Steam relay server. Messages can be sent reliably using Transmission Control Protocol(TCP)-like packets or unreliably using User Datagram Protocol(UDP)-like packets. Sending messages reliably is useful for events that definitely have to arrive, where losing a message would cause a resend. An example would be the placement of a box. Such a message has to arrive as otherwise the client would be missing this box. When it does not matter that much if a message arrives, it is better to send it unreliably, as this is faster than sending it reliably. This would be the case for movement messages, as a lot of these messages are sent and when one message is lost, it is not really worth resending it as the next movement message will make it obsolete anyway.

It would have probably been easier and faster to use Unity's old high level scripting API [10]. It implements core features most online multiplayer games require. However this API was deprecated [11] and at the beginning of this project, while a replacement was announced, it was not available yet. For this reason, the only option was implementing it ourselves. So we decided to use Steamworks providing only basic low level features. While this approach is more flexible, it is also more complex. Using Steamworks also binds the game to Steam. However as it only provides low level features, it is not too hard to replace it if needed. To serialize messages before sending, the serialization library FlatBuffers [12] is used. This library is intended for game development and can access serialized data without parsing or allocating additional memory.

2.2.1 Delay On Reading Messages

Reading messages with Steamworks is done by polling. The game does that in Unity's update function. Unity calls all update functions once per frame, so messages are only read once per frame. Therefore in the worst case, a message would need to wait for 16.66 ms until it is read by the game. To lessen this delay, messages are also read in Unity's other update functions, lateUpdate and fixedUpdate. While it is unfortunate to have an additional delay on the communication, it is still a short delay. Thus it is not that much of a problem.

Creating another thread and polling for new messages from there does unfortunately not work that well, as one cannot use any Unity functionality from any other thread than the main tread, because Unity's API is not thread-safe. You could read messages on a separate thread, but you cannot apply any changes to the game state until the main thread reaches its next update function. The only messages that would benefit from such a solution are those that could benefit from it without changing the game state. This would only apply to time synchronization messages. They need to record their arrival time to determine the Round Trip Time (RTT). Without this varying delay before the message is read, the arrival time would be recorded more precisely. This would improve the precision of the whole time synchronization. As shown in Section 5.2, the time synchronization is already sufficiently precise, so we decided against implementing this solution. If later turns out that it is not precise enough anymore or new messages are added that also profit from this solution, it would be worth considering again.

Chapter 3

Time Synchronization

The update window described in Section 4.1 requires the server and clients to synchronize their time. While there are libraries that solve this problem, we could not find a library that is intended or recommended to be used with Unity. Such a library would also not use Steamworks for sending messages and therefore would at best open a new connection, but at worst require a new established connection including Internet Protocol (IP) address resolving and Network Address Translation (NAT) traversal. To avoid these problems, we implemented a simple clock synchronization between server and clients.

To measure the time, we used a c++ library [13], which claims to measure the elapsed time with at least 1 µs accuracy. We did not use the time function provided by Unity [14], as it only measures the time once per frame. It is intended to supply the time the game has been running, not the time per frame.

3.1 Protocol

3.1.1 Synchronization With Two Messages

The time synchronization is similar to the Network Time Protocol (NTP)[4]. The core idea is that the server clock is the leading clock. The clients exchange messages with the server to adjust their clocks to the one of the server. These adjustments happen in cycles. Such a cycle is described in Figure 3.1. The

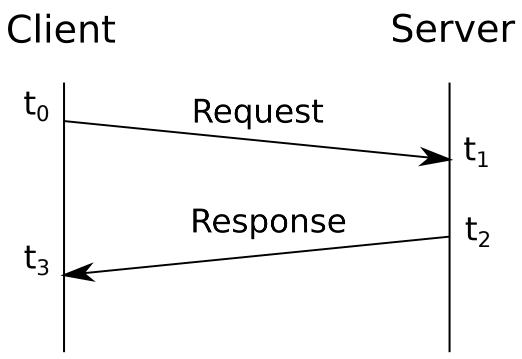


Figure 3.1: One time synchronization cycle.
Request: $\{t_0\}$
Response:$\{t_0, t_2 - t_1, t_2\}$

client initializes a time synchronization cycle by sending its current time t_0 with a request message. The server responds with a response message. This message contains $t_2 - t_1$, which corresponds to the handling time of the message on the server. It also includes t_2, the time when the server sent the response. With this message, the client can estimate the Round Trip Time (RTT) as follows:

$$RTT = t_3 - t_0 - (t_2 - t_1) \tag{3.1}$$

At last, the client sets its own clock to $t_2 + RTT/2$. It adds to the time when the server sent its response the estimated time it took the message to reach the client. This concludes one time synchronization cycle. Every client initiates such a cycle roughly once a second. In case one cycle takes extraordinary long and arrives after the next cycle has already started, cycles could mix up. To prevent this, the client only accepts response messages whose t_0 is equal to the t_0 of its last sent request. As explained in Section 5.2, the actual time correction the client applies gets damped by a constant factor of 0.3, evaluated through experiments. This counterbalances overcorrections and makes the client time more stable. It is also necessary to ignore exceptionally high corrections, as they are in most cases too high and need to be canceled in the next cycles. To achieve this property, the game keeps track of the average RTT. A correction will only be accepted, when the following condition holds on its estimated RTT: $RTT_i < 1.5 \sum_{k=0}^{i-1} RTT_k/i$. Further detail on this approach is provided in Section 5.2.

3.1.2 Synchronization With Three Messages

At first, we performed time synchronization using three messages instead of two. This algorithm is very similar to the current one described in Section 3.1.1. It only differs in the Message exchange, which is described in Figure 3.2.

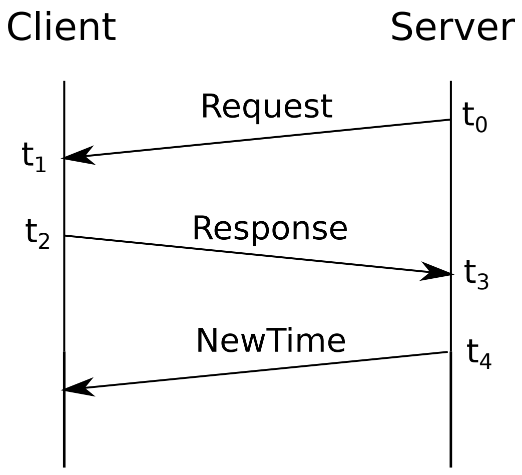


Figure 3.2: One time synchronization cycle using three messages.
Request: $\{t_0\}$
Response: $\{t_0, t_2 - t_1\}$
NewTime: $\{t_4 + RTT/2\}$

The server calculates the RTT using equation 3.1. As shown in Section 5.2, this algorithm performs worse than the one with two messages. In the beginning, we used this algorithm, but we later realised that the same goal can be achieved using only two messages by restructuring the message exchange.

CHAPTER 4

Game State Synchronization

Hat Hunters is a real-time game. Every time a player performs an action, it is immediately visible to all players. To ensure that all players have the same view on the game, the game state of all players has to be synchronized. Figure 4.1 illustrates an example, how two instances of the game can get out of synchronization. Two instances I_1 and I_2 of the game on separate computers display the state of the same character in the same match. Both share the same state at time t_1. I_1 starts moving the character by adding a velocity v at time t_2 and sends a message to I_2, informing it of this action. Due to the latency of the message, when the message arrives at I_2 at time t_2, the character already moved in the state of I_1, resulting in a different state between I_1 and I_2.

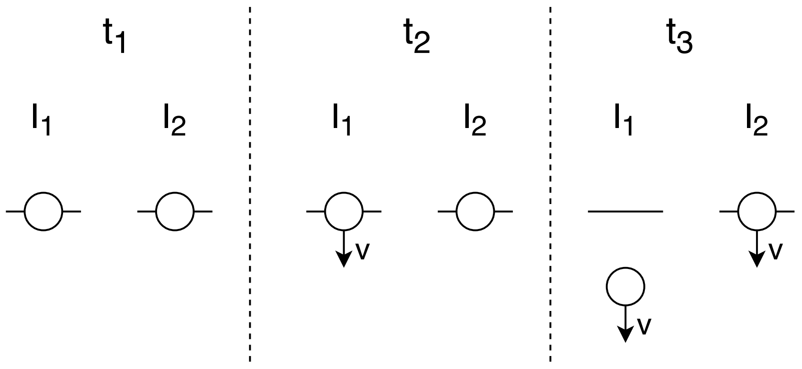


Figure 4.1: Two instances I_1 and I_2 display the state of the same character at time t_1, t_2 and t_3. Due to the latency of messages, the action I_1 performs at time t_2 results in a different state for I_1 and I_2 at time t_3

The latency of sending a message between computers is generally small compared to the reaction time of humans. So this state difference will be small as well. However if such differences are left unresolved, they will accumulate with every movement action, until both computer states are perceivably different. It is necessary to know which state is the true state of the character in order to resolve this difference in states. The instance that does not hold the true state will adjust its state to the true one. Knowing the true state is also necessary to determine if the character for example has collected a star or got hit by an item.

We use a client-server architecture for Hat Hunters, as stated in Section 2.1. This provides an easy answer to the question which computer has the true state, as in this architecture, it is always the server. To stay synchronized with the server when performing an action, the client can act as a dumb client. This is shown in Figure 4.2.

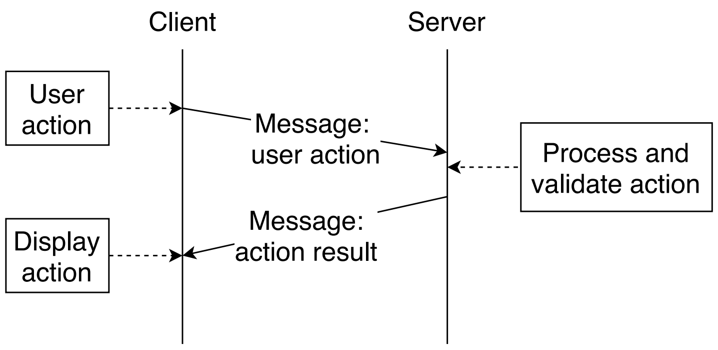


Figure 4.2: Client acting as dumb client. When the user performs an action, the client waits with displaying the result of the action until it receives the final result from the server.

In this solution, the client never performs an illegal action that the server did not permit, because the client always waits for the result from the server before doing anything related to this action. There is however still a problem. If the user makes an input to perform an action, the result of this action is not shown until the server response arrives. This means that the user has to wait for the duration of a message round until the game reacts to its input. Depending on the latency between client and server and the action the user performs, this can feel lagged and unnatural. But this does not hold for all actions. Actions like placing a box are implemented using a dumb client. It turns out that using a dumb client does not feel lagged for the user when placing a box, but it has the advantage that there is never a box placed in the wrong position. Correcting the position of a box would look unnatural, as boxes never move. There are however other actions like movement of the character, for which using a dumb client makes moving feel lagged and unresponsive. To prevent this, the client can make predictions on the server response and apply them before receiving the actual response. The client prediction process is shown in Figure 4.3.

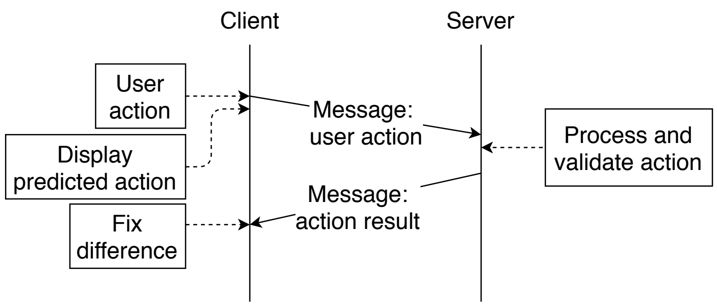


Figure 4.3: Client predicting result. The client displays its prediction of the server result. When the actual result arrives, the client needs to fix the difference between actual response and prediction.

Using client prediction, user actions feel to the user as if she is playing a single player game with zero latency on her actions and thus responsive. But the client still needs to hold its game state consistent with the server game state. So it needs to fix the differences that arise, but this can be problematic. In the case of character movement, if the client notices a difference of the character position and simply moves the character to the true position, the user will notice this. It is weird if the character suddenly moves on its own or ignores input from the user. Section 4.2.3 shows how the client fixes movement differences in the game. Fundamentally, client prediction allows the game to be more responsive at the cost of consistency between server and client [3, pages 83-89]. So depending on the game action, we used a dumb client or client prediction. More details on how game actions are synchronized is provided in the following sections of this chapter.

Looking back at the example depicted in Figure 4.1 we now have a mechanism to synchronize those differences. While those differences still remain, as the message holding the real position needs to reach the client, they do not accumulate over several actions anymore. To actually reduce the difference in character position, we implemented a mechanism that we call *update window*. It is explained in Section 4.1.

To synchronize the game state, the server sends messages to the clients containing updates on the state. These updates could either be full updates, which contain the whole game state, or incremental updates, which only contain the difference of the game state from the previous update. We decided to use incremental updates. This reduces the size of messages, as full updates have a lot of overhead due to their nature. It also makes writing and reading messages faster. The server does not need to copy the entire game state when writing a message, and the client does not need to read out the entire game state and change its game state accordingly. Full state updates have the advantage that it is more error resistant. It is not possible to miss a change, because even if no update is sent for a change, it will always be included in the next update anyway. All full state updates could be sent unreliably, as every message carries the whole state and it thus does not matter if one message is left out.

4.1 Update Window

The core idea of the update window is that the server can recompute all game states during the last 100 ms. Every action the client sends to the server is timestamped. If the server receives such a message, the server can use the update window to recompute the game state at the time when the client performed this action. The server can then apply the action and calculate all further game state updates until the server reaches its current game state. This new game state on the server contains the client action as if it really was applied at the time the client applied the action. This means that the latency, which normally offsets all actions sent by a client, is circumvented through this mechanism. This does however only work, if the time synchronization between client and server is precise enough. This is required to allow the server to find the right game state when the client applied its action. Chapter 3 explains how time synchronization is achieved.

Without the implementation of the update window, the server would perform all client actions in its current game state and thus later than they actually happened on the client. This means that the client needs to correct bigger differences to stay synchronized with the server, which ultimately is irritating for the player and makes the game feel lagged.

4.1.1 Detailed Mechanism

The game on the server progresses by updating its current game state once per server frame to obtain the new current game state. Every frame, instead of just updating the current game state, the server performs an update window iteration, which recomputes past game states, and then updates the current game state to the new current game state. Figure 4.4 depicts how one update window iteration works.

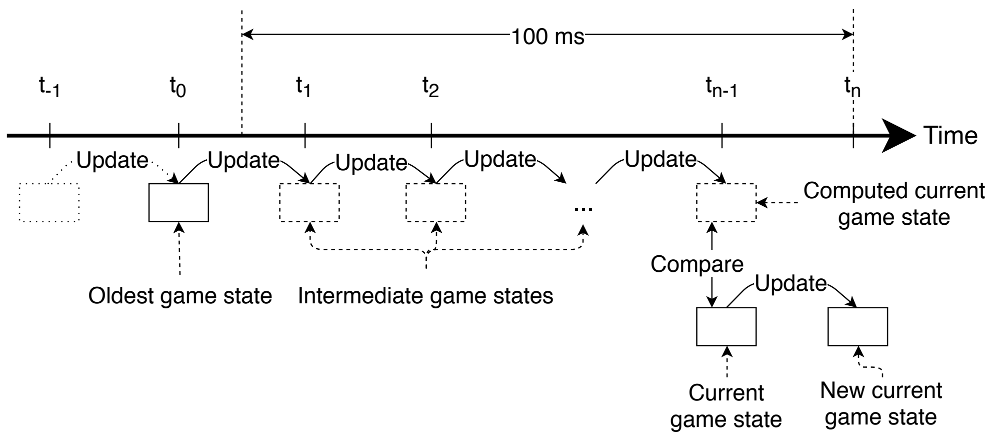


Figure 4.4: One update window iteration and the following update of the current game state. The oldest state is updated until the computed current game state is reached. $t_0, t_1, ..., t_n$ are the update times of their corresponding frames. All states that are older than the oldest game state are not stored and cannot be recomputed. Only the oldest game state and the new current game state are stored.

The server keeps a list of frames which stores for every frame in the list the time t_i when the game state update originally happened in the frame and all messages that have to be applied in this frame. In Figure 4.4, the frames with the times $t_0, t_1, ..., t_n$ are the frames which are currently in this list. The update window iteration starts from the oldest frame with its corresponding oldest game state. The server applies all messages that have to be applied in this frame and then updates the game state. This process is repeated for all subsequent frames until the frame of the current game state is reached, we call the resulting game state the computed current game state. The computed current game state and the current game state are then compared. When the server detects a difference, it adjusts the current game's state to the computed current game state. This whole process represents one update window iteration. Only now does the server update the current game state to the new current game state and thus advance the game. The server then notifies the clients of the differences it detected earlier.

Game state updates are time dependent. Mechanics like character movement, item movement or box place cooldown[1] use the time that passed since the last frame to update. This is the reason why the game stores the times $t_1, t_2, ...$ of each frame, because otherwise the server could not reproduce old game state updates consistently.

When the server receives a message from a client, it assigns the message to the following frame like shown in Figure 4.5, depending on the time stamp in the

[1]Box place cooldown is the time a player has to wait after placing a box until they can place a new box.

message. But the server also assigns messages, which it has not received from clients, to frames. This is done to allow the server to registers events, like moving an AI controlled character or collecting an item. Such events would otherwise disappear in the next update window iteration if they happen in an intermediate game state, as these game states are not stored and the event does not affect the oldest game state. So it is used like a note to remind the server to perform an event again in the next update window iteration, until the event reaches the oldest game state where it will be kept.

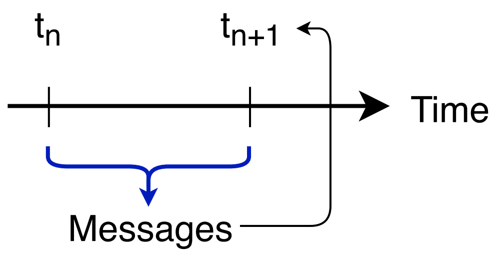


Figure 4.5: Arriving messages with a time stamp between two frames with times t_n and $t_n + 1$ are assigned to the frame with time $t_n + 1$.

We do not have an inverse function to the update function. So to be able to recompute past game states in the next update window iteration, the server keeps the oldest game state. Intermediate game states and the computed current game state are not stored, as they can be recomputed from the oldest game state and get outdated when new messages arrive. The oldest game state is always the state of the newest frame which is at least 100 ms older than the new current frame. This allows all messages that arrive within this 100 ms update window to be applied in their corresponding game state. Older messages are ignored, as they are too old and would probably do more harm than good. Therefore game states that are older than the oldest game state are not needed. For this reason and to limit the number of updates the server has to perform, they are not stored. The frames which correspond to those game states are removed from the frames list. We choose 100 ms as the time during which game states can be recomputed, because this should be longer than the latency of a message. At the same time, 100 ms are not too high, because the higher this time the more game state updates the server has to perform in each frame. When the game runs at 60 fps, the server recomputes around six game states in each frame.

To keep the oldest game state for the next update window iteration, but at the same time update the game state until reaching the computed current game state, the game needs to be able to copy states. Figure 4.6 shows how the next oldest state is copied and thus preserved.

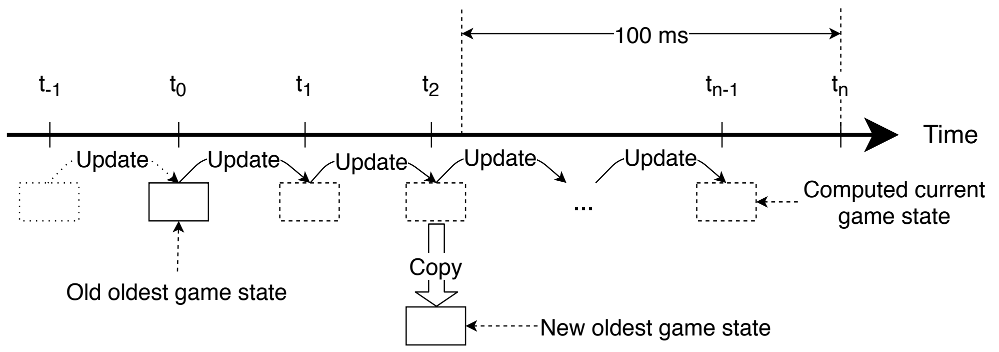


Figure 4.6: The update window iteration starts from the old oldest game state. When the newest game state is reached that is older than 100 ms, the current game state is copied into the new oldest game state.

The only states that are kept for the next update window iteration are the new oldest state and the new current game state, which is always kept. While copying game states is necessary, it is an expensive operation, as it needs to copy all objects contained in the game state. To make it even worse, copying a game state is necessary in every update window iteration, so it is performed once per frame. To avoid allocating new objects in every frame, which would be expensive, there are several copy pools, one for each type of copy. Every copy pool manages a list of unused copies. When the copy operation copies an object, it requests an instance of this object from the corresponding copy pool. The copy pool returns an unused copy if there is one in its list, or otherwise returns a newly allocated object.

It might seem weird that there is this difference between computed current game state and current game state. Why is there a need for the current game state, would it not be enough to just compute the computed current game state and use this state for the newest update to the new current game state? This would be a cleaner solution, as there would be no need to separately store the new current game state. Only the oldest game state would be stored to recompute all following game states, including the new current game state, as shown in Figure 4.7.

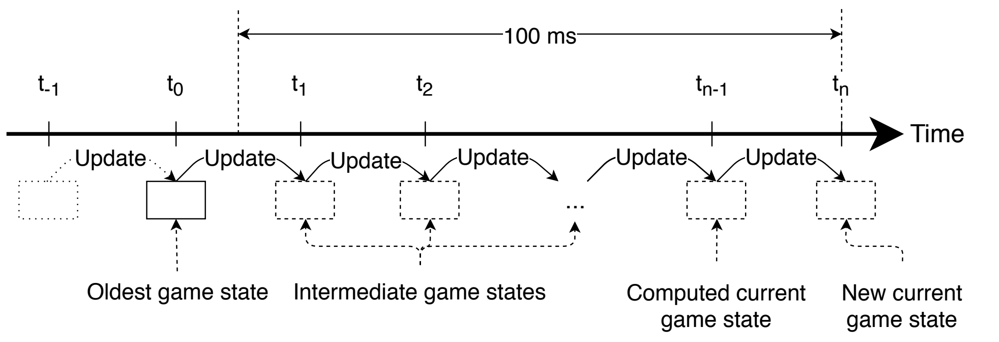


Figure 4.7: Alternative solution without separately storing the new current game state. Instead of updating the current game state, the computed current game state is updated to obtain the new current game state.

One reason for not using this alternative solution lies in the copy process of game states. The copies of game states are not exact copies of their originals, they only copy values which are relevant for the update window and synchronization. But there is a worse problem. In the alternative solution, the server is not able to find out if the computed current game state of this update window iteration has any differences compared to the corresponding new current game state of the previous iteration. It only has the computed current game state of this iteration, so it cannot compare them. This is illustrated in Figure 4.8

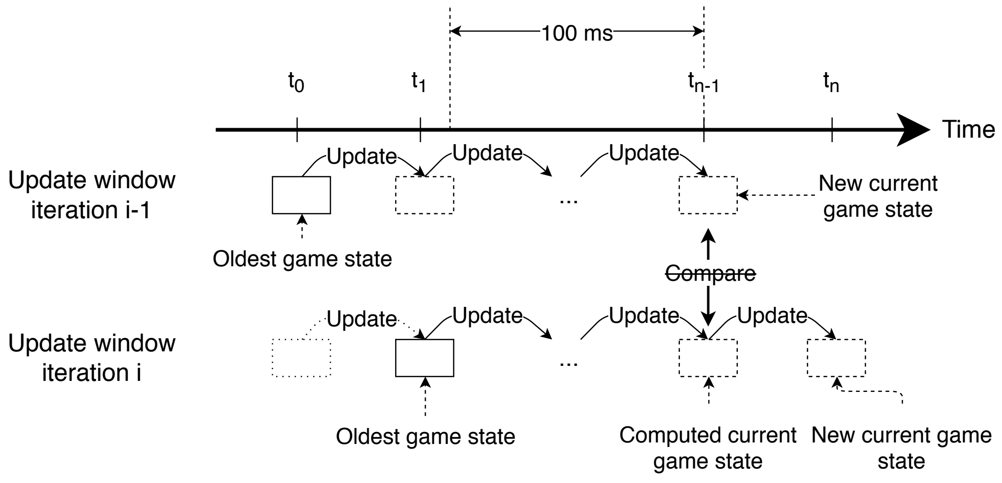


Figure 4.8: Problem of the alternative solution: in update window iteration i, the computed current game state can no be compared to the new current game state, because it was not stored in update window iteration i-1.

This is a problem, because the server does not only need the new current computed game state for itself, it also needs to share it with its clients. When the server does not know if there are any differences between the previous and the current computed game state, it cannot forward those changes to the clients and would thus get out of synchronization with them. To fix this problem, the server could instead send the whole computed current game state every frame, but then we would lose all the advantages of relative updates described in the first section of this chapter. The server could also just keep the new current game state from the last update window iteration. This would allow to find all differences between those game states and accordingly notify clients, similar to the current implementation. But this would also reduce the advantage of this alternative implementation, as the server still needs to store two game states. So the only advantage would be that it only compares the game states to notify clients, but does not adapt one game state to the other. This has only a small impact, as finding differences is the expensive operation, not adjusting them, which is why we stayed with the current implementation.

4.2 Character Movement

The player moves her character on the corresponding client. The input from the player adds a force to the character in the corresponding direction. This force changes the velocity of the character, which in turn defines the new position. The client needs to synchronize the movement of the character with the server. Acting as a dumb client does however not work, waiting for the server response makes the movement of the character feel lagged and unresponsive to the player. Therefore the client uses client prediction. The client predicts the result from the server by moving the character according to the player input. Because of the message delay, this prediction will not be exactly the same as the result from the server. Due to the update window on the server, it is only off by the latency of a message instead of the Round Trip Time (RTT).

All movement messages are sent unreliably. There are a lot of movement messages exchanged between server and client. So if one message gets lost, it is not really worth resending it, as the next movement message arrives soon making the lost one obsolete again. Messages arriving out of order is no problem on the server due to the update window and on the client because the client only accepts messages with a newer time stamp than the last one.

4.2.1 Sending Movement Update

The client needs a way to determine when to send messages to the server to inform it of the current movement of a character. Ideally the client sends updates only when there is a difference between its character movement and the one of the server and other clients. To find out whether there is a difference in movement, the client predicts the movement of its character on the server and sends a message when the position of the prediction differs too much from the actual position [3, pages 89-92]. The server and clients move remote characters by using the information from the last movement message they received to move the character with constant force until the next message arrives. Therefore the prediction of the client controlling the character is made using the information contained in the last movement message the client sent. As the force for the prediction is fixed to the one of the last message, the actual and the predicted character position will differ when the force applied to the character changes. When the position difference exceeds a threshold, the client sends a new movement message and adjusts its prediction of the server client position.

AI controlled characters use the same method to determine if an movement message needs to be sent. The only difference is that this is performed on the server and that the client receiving the message simply accepts it and corrects the position of the corresponding AI character accordingly.

4.2.2 Latency Compensation Using Update Window

The server receives a movement message from the client. This message includes a time stamp, holding the time when the client applied the movement action to a specified character. It adds the message to the corresponding frame in its frame list, depending on this time stamp. In the next update window iteration, when the server reaches this frame, it adjusts the movement of the specified character. It only sets the force applied to the character, the velocity and the position remain unchanged. The reason is that the server only applies the action of the client, it does not adopt the character state, as the server holds the true position and velocity of the character. When continuing the update window iteration, the character position gets updated with this new force, until reaching the current frame. This is illustrated in Figure 4.9.

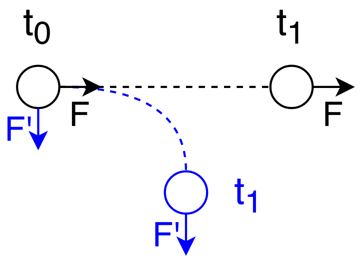
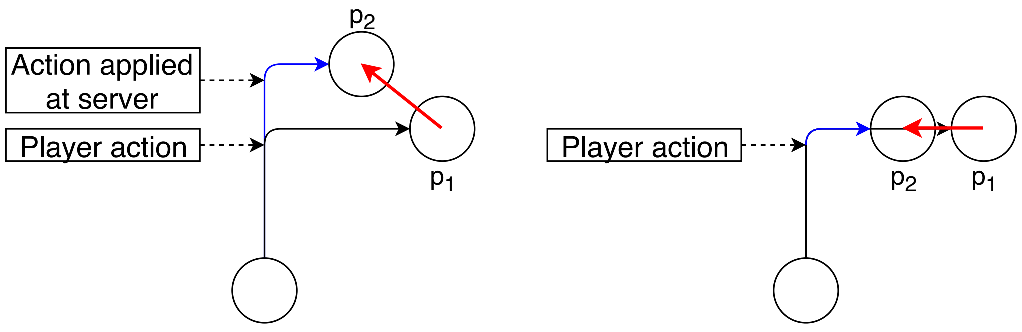


Figure 4.9: The current game state is drawn in black at time t_1. Due to applying the Force F' from the movement message at time t_0 in the update window iteration, the character ends up at a different position in the computed current game state at t_1 indicated in blue.

With this mechanism, the server can circumvent the latency of the movement message and end up in the same game state (drawn in blue) as if the server applied the character movement at the same time the client did. In the end of this update window iteration, the server sends a message to every client holding the new position, force and velocity of the character.

Without the update window, the server applies the movement later than the client. This results in the problem illustrated in Figure 4.10a.



(a) Without update window, the character gets corrected diagonally to the top left.

(b) With update window, the character gets only corrected backwards.

Figure 4.10: Movement correction on the client with and without update window on the server. Movement correction is indicated in red.

As the player action to move the client arrives later at the server, the character moves on the client according to the black arrow and on the server according to the blue arrow. When the character reaches position p_1 on the client, the message from the server arrives correcting the character to position p_2. This leads to a correction diagonally to the top left, indicated in red. This is clearly noticeable by the player, as the character is suddenly drawn back further out of the curve. On the other side, if the server uses the update window, the movement correction changes as shown in Figure 4.10b. The server applies the movement action at the same time as the client. So when the character reaches position p_1 on the client, the correction to p_2 from the server is only backwards. This is due to the latency of the response from the server to the client. This correction only slows down the character, which is barely noticeable by the player anymore.

4.2.3 Character Movement Correction

The client corrects the movement of its characters when it receives a movement message from the server. This happens for both, characters which are controlled by a player on this client and remote controlled players. Simply changing to the movement of the message does not work, as such sudden changes in position would look unnatural and irritating to the user. Therefore, character movement correction adjusts the movement over several frames, drawing the character to the correct position. It is illustrated in Figure 4.11.

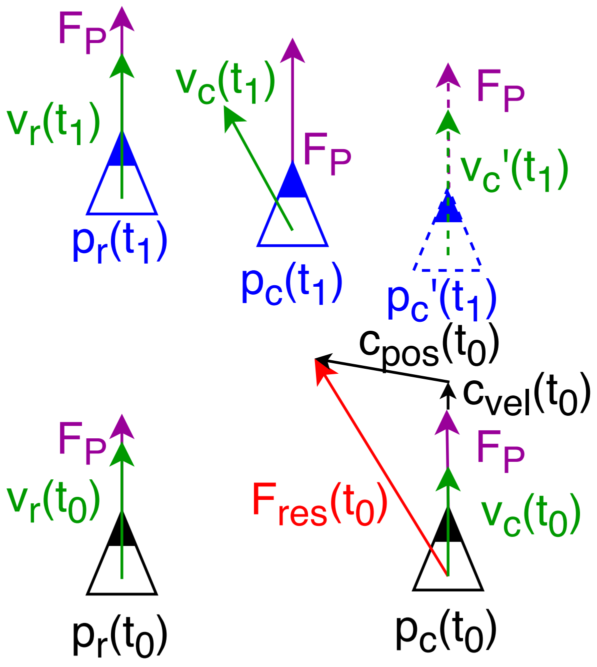


Figure 4.11: The current character with position $p_c(t)$ gets corrected to get closer to its real position $p_r(t)$

The same character is shown in two consecutive frames with times t_0 and t_1. $p_c(t)$ and $v_c(t)$ are the current position and current velocity of the character, as it is displayed in the game. When the client receives a movement message from the server, it sets the real position $p_r(t)$ and the real velocity $v_r(t)$ of the character to the values from the message. The force F_P is currently applied to the character. The character is at position $p_c(t_0)$, but it should be at position $p_r(t_0)$. In the next frame, the real position is $p_r(t_1)$ and without applying any corrections to the current character, its position would be $p'_c(t_1)$. $F_{res}(t_0)$ is computed as follows:

$$\begin{aligned} c_{pos}(t_i) &= \alpha(p_r(t_{i+1}) - p'_c(t_{i+1})) \\ c_{vel}(t_i) &= \beta(v_r(t_{i+1}) - v'_c(t_{i+1})) \\ F_{res}(t_i) &= F_P + c_{pos}(t_i) + c_{vel}(t_i) \end{aligned}$$

The force F_{res} corresponds to the force F_P, but it is adjusted depending on the position and velocity difference to the real character. α and β are constants that damp the position and velocity corrections and were evaluated through testing. F_{res} is applied to the current character state, resulting in position $p_c(t_1)$. This is the next current position of the character which will be displayed in the game. As F_{res} gets adjusted by the differences between current and real character state, $p_c(t_1)$ is closer to $p_r(t_1)$. This process is repeated in every frame, until the position and velocity difference is small enough.

F_P is the force applied to the character and its value depends on the kind of character. If the character is remote controlled, then F_P is equal to the force from the last server message. Otherwise, F_P is defined by the input of the player controlling the character. F_P is applied to the real character state as well, because otherwise the real character would just stay at the same place and get outdated as the character moves in the next frames.

The marked vertex of the triangles in Figure 4.11 indicate the looking direction of the character. In the local multiplayer version of the game, the looking direction of the character is defined by the direction of the velocity. But at position $p_c(t_1)$, the looking direction and the direction of $v_c(t_1)$ do not match. There is a problem with using $v_c(t)$ to determine the looking direction. The looking direction of the character does stand out a lot to the player. If the character would look in the direction of $v_c(t_1)$, it would feel weird to the player. The player made the character move in direction of F_P, but the character suddenly does not look in that direction anymore. The character moves in direction $v_c(t_1)$, but this is less noticeable to the player as long as the character still looks in the direction the player expects. Therefore, the looking direction is determined by a separately stored, from the character movement correction independent velocity.

The current and the real position of a character can be so far apart from each other that drawing the character to the real position does more harm than good to the player experience. Instead if the difference is big enough, the character changes its position instantly to the real position. To the player this looks like the character teleported to the new position. While this is still bad and disturbs the player, at least the player has immediately full control over the character again.

Having such a distance between real and current position can happen when the client and server position get separated by an obstacle. Teleporting allows to pass such an obstacle. Drawing the character like described above however cannot pass any obstacles and thus not resolve this problem.

4.3 Boxes

Placing a box starts with a player pressing the button to place a box. The client sends a message to the server, informing the server of this action. The server adds the message to the corresponding frame in the frame list. But in the next update window iteration, it does not directly place the box. Instead, it waits until this frame, containing the box message, would get removed from the frame list. Only then does the server place the box, if the corresponding character actually can place a box, at the character's position and send a message to all clients, instructing them to place a box at this location.

The reason for waiting and not placing the box immediately is, that now that the frame gets removed, the corresponding game state is fixed and cannot be changed by later messages anymore. In particular, the position of the character placing the box cannot change anymore. If the server would place the box immediately, the position of the placing character could still change, in which case the position of the box would change as well. This would be problematic, as boxes normally never move. So correcting the position of a box would look unnatural and thus stand out to the player. While waiting bypasses this problem, it has another drawback. From pressing the button until the box actually appears, the player has to wait for at least 100 ms until the game state gets removed from the update window on the server. However this is not as bad as it sounds, because the box is only placed when the character leaves the space of the box. While there appears a silhouette in place of the future box, it still turns out that the added delay does not bother the player.

Removing boxes is handled analogously to placing them, with the difference being that the server initiates the removal of a box. The client ignores events that would remove a box, like a projectile hitting a box. It waits for the corresponding instruction from the server. If the client would remove such a box right away, it could happen that this box was not removed on the server and no box removal message would arrive at the client. The client would have to notice this and place the box again. Similar to changing the position of a placed box, removing a box and then suddenly place it again would look weird to the player.

Both the message from the client and the message from the server are sent reliably. If one of those messages is lost, they would have to be resent in order to place or remove the box. It is even more important that the messages from the server arrive, because if they do not, the state of the game field would be out of synchronization with the client missing the message. As all game field messages only contain the relative update, this difference would not be corrected later. So it is important that those messages arrive, thus they are sent reliably.

Normally in the end of an update window iteration, the current game state is compared with the computed current game state. In order to compare placed boxes, it would be necessary to iterate over the whole game field. This is circumvented by placing or removing a box directly in the current game state when it is placed or removed in the update window iteration. It is safe to place a box like this in the current game state as the state of a box does not change, unless it gets removed. In that case, the removal action is also applied when the game state reaches the end of the update window and removes the box directly in the current game state as well.

4.4 Collection Of Items And Stars

Items and stars are only collected by characters on the server, they are ignored on clients. Clients wait for the corresponding message from the server to collect items and stars. A character triggering a collection does however not directly lead to the collection event on the server. Similar to the placement of a box, the server waits with collecting the item or star until this action reaches an old game state which will be removed in the next update frame iteration. Now that the server can be certain that the character will receive the item or star, it sends a message to all clients informing them of this event. Only now does the client perform the collection. In case the character collects an item, the server also decides what kind of item it will receive.

The server sends the message reliably, because it needs to arrive at the clients to inform them of this state change. This is especially true for collecting stars, as it does not only remove the star, but also increase the score of the collecting character. As the character score is not transmitted in any other way, these messages definitely need to arrive.

4.5 Green Projectiles And Moving Stars

Green projectiles and moving stars behave similar to each other, they move the same way, bounce off stumps and boxes and interact with characters they hit. For this reason their synchronization is almost identical. The remainder of this section explains how green projectiles are synchronized, but the same mechanism is applied to moving stars.

Placing a projectile is synchronized like placing a box. In contrast to boxes, after placement projectiles move with constant velocity. The velocity only changes when the projectile hits an obstacle. So in order to synchronize projectile movement, the server sends a message to all clients whenever the projectile hits an obstacle. Hitting an obstacle either breaks the projectile or makes the projectile bounce off. The client uses the movement information of such a message in case of bouncing off an obstacle to determine the movement of the projectile until the next message arrives. But the client does not wait for the server message when the projectile bounces off an obstacle, because waiting would result in the projectile moving too far into the obstacle and look weird to the player. Instead the client predicts the bouncing of the projectile and later adjusts its prediction to the message from the server. As projectiles move predictable, the client prediction is most of the time really close to the actual result from the server and thus only small corrections on the client side are necessary.

Projectiles break when hitting a character or bouncing off too often. The client does not break a projectile on its own, it waits for the server message instructing the client to do so. This prevents the client from breaking a projectile by mistake and later recreate them as they should still exist. A projectile hitting a character does not only break the projectile, the character also loses one star. Like before the client waits for the server message, instructing it to remove a star from the character and place it in the field. Such stars are moving stars and as mentioned earlier they behave similar to projectiles and are thus not further explained.

All those messages controlling projectiles are sent reliably. The server waits with sending those messages until the game state that caused those messages cannot be changed through the update window anymore. The same reasoning as for boxes can be applied for placing and removing projectiles. Movement messages are sent when a projectile bounces of an obstacle, depending on the field this can happen rarely so those few messages are sent reliably to prevent large corrections due to a lost message. Through client prediction, the delay due to waiting until the event is fix can be diminished, leaving the benefit of the certainty that the bouncing off the obstacle happened.

Chapter 5

Results

5.1 Host Running Server And Client

To find out whether running both the server and the client on the host itself is feasible, we used the Unity Profiler [15]. This tool shows how much computation time is spent on which area[1] per frame. While it is easy to get this information per frame, the Unity Profiler does not allow to get information like the average computation time across all frames. It is possible to export the measurement data from the Unity Profiler, but then the computation time is distributed among hundreds of functions instead of those areas. So even exporting the data does not allow to determine the average computation time of those areas across all frames. Therefore we took ten frames from the Unity Profiler and copied the values by hand. Table 5.1 shows the average computation time of those ten frames in the local multiplayer version of the game. The game was played in regular local multiplayer mode with a resolution of 1920x1080 and the graphics quality set to high. The computer running the test has an Intel Core i7-3770 processor and 16 GB RAM.

Area	Rendering	Scripts	Physics	Animation	Garbage collector	UI	Others	Total
Time[ms]	14.127	0.71	0.009	0.016	0.136	0.097	0.81	15.90
Percent	88.82	4.46	0.06	0.1	0.86	0.61	5.09	100
Max[ms]	15.8	1.61	0.01	0.46	1.36	0.68	1.56	21.48

Table 5.1: Average computation time of ten frames for each area, shown in absolute values and percent. The bottom row shows the maximum time per area during these ten frames. Most of the time is spent on rendering

Most of the computation time is used to render the game. In comparison, only little time is spent on running all of the game's scripts. The game can use

[1]The physics area is misleading, because the game uses a custom physics system instead of Unity's physics engine. The games physics are thus included in the scripts area.

16.66 ms to compute one frame of the game, so there are on average 0.75 ms left, which would be just enough to double the time spent on scripts. This could be a bit tight, but considering there is so much time spent one rendering, one could free some time by lowering the graphics settings. The server does not affect the work the renderer has to do, because the server only computes the game's logic and does not perform any graphical calculations. So running the game twice on the host for server and client should not be a problem and we decided that this approach is viable.

5.2 Time Synchronization

To analyze the performance of the time synchronization, we measure the difference between the time of the client and the server time estimated by the client. When this time difference is low, the client and the server time are closer together. High time differences indicate that the client and the server time are not well synchronized. A bad time synchronization could cancel the benefits of the update window altogether. Another reason to keep the time difference low is that the client corrects its time depending on the time difference and it is undesirable for the client time to fluctuate too much.

Table 5.2 shows a summary of the exact specifications of all experiments. Each experiment number identifies a plot which shows the estimated time difference between client and server for each cycle. An experiment can appear in more than one figure.

Experiment #	1	2	3	4	5	6	7-16
Correction damping	No	0.3	0.3	0.3	0.3	0.3	0.3
Cycle interval	1s	1s	2s	0.5s	1s	1s	1s
Connection type	LAN	LAN	LAN	LAN	LAN	LAN	LTE
#Messages/cycle	3	3	3	3	2	2	2
Discard outdated messages	No	No	No	No	Yes	Yes	Yes
Acceptance condition	No	No	No	No	No	Yes	Yes

Table 5.2: Specification of all experiments. Cycle interval is the time between two synchronization attempts and acceptance condition refers to whether Condition 5.1 was applied.

The average and the standard deviation of the estimated time difference between client and server for all experiments is listed in Table 5.3. Every run has a different initial difference of client and server time. Depending on that initial difference, the experiments have vastly different averages and standard deviations. Those values are more meaningful when the synchronization is in normal operation, after the initial difference is overcome. Therefore in Table 5.3 for both the average and the standard deviation, all values from the initialization phase are skipped by leaving out the first 60 cycles, were the initial time difference certainly does not affect the result anymore. The sections shown in figures also do not include the initialization phase, except when otherwise stated.

Experiment #	1	2	3	4	5	6
Average [ms]	-0.039	-0.871	-0.003	0.506	-0.035	0.082
Standard deviation [ms]	48.565	26.265	29.605	28.038	16.337	4.516

Table 5.3: Average and standard deviation of the estimated time difference between server and client. Values from the initial phase of the time synchronization are skipped.

Experiment 1, shown in Figure 5.1, still uses the old time synchronization, which exchanges three messages per cycle. It is explained in more detail in Section 3.1.2. The client corrects its time by the estimated time difference and it also accepts all messages, not just the ones from the current cycle. This experiment is performed on two computers, with both of them being in the same LAN. There are several spikes a lot bigger than the average time differences. They often get corrected by a similarly big spike in the opposite direction. The graph generally often cross the x-axis, implying that a significant part of the corrections performed by the client is due to noise, not because of an actual time difference between server and client. This is also visible in Table 5.3. The average time difference of Experiment 1 is low, but the standard deviation of the time difference is high. While the time difference changes a lot, the overall change is small. In order to make the time synchronization more stable, we damp the correction, meaning that the client does not correct by time difference d, but by $\alpha * d$, where α is the damping factor. Experiment 2 shows the result with applied damping. While big spikes still remain, the average fluctuations get smaller and more stable. The improvement is clearly visible in the standard deviation of this experiment, which improved from 48.565 ms in Experiment 1 to 26.265 ms. After trying out different damping factors, a factor of 0.3 seems to be a good choice. It strikes a balance between correcting too much or too little.

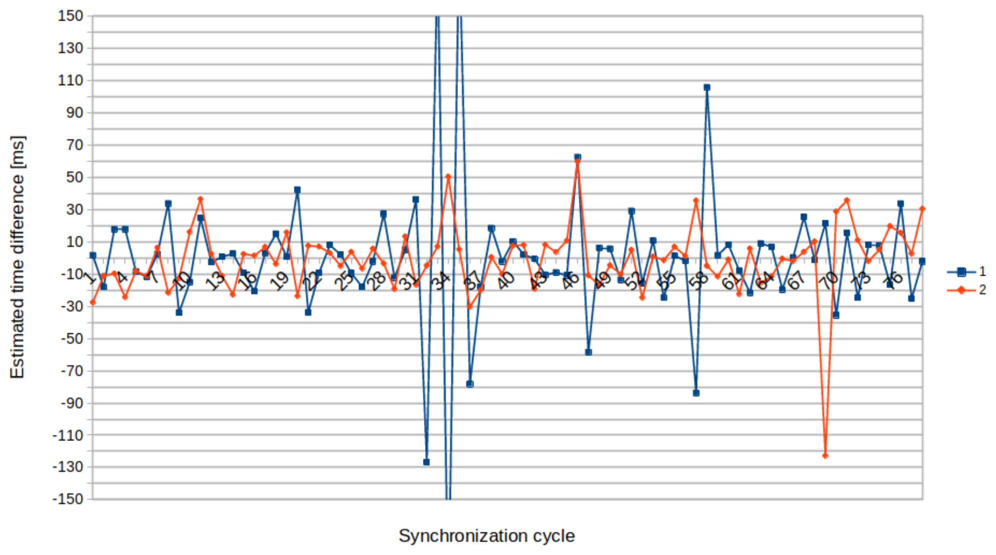


Figure 5.1: Estimated time differences between client and server for Experiment 1 and 2. The right side declares which graph belongs to which experiment. By damping the corrections, Experiment 2 gets more stable than the first one.

It is important to note that in all experiments with damping, the client does not correct its time by the estimated time differences between client and server shown in the corresponding graphs. The client corrects its time only by the damped estimated time difference.

Experiment 3 uses the same damping factor as Experiment 2, but it has a cycle interval of two seconds instead of one. Figure 5.2 shows that this results in more spikes than before. The standard deviation in Table 5.3 is also a bit worse compared to Experiment 2. One can also see in this Figure that Experiment 4, which has a smaller cycle time of 0.5s, has smaller average fluctuations. However the standard deviation does not surpass the one of Experiment 2. Therefore more experiments would be necessary to determine if there really is an improvement. We decided against a smaller cycle interval, because we did not want to send too many messages. But if in the future the synchronization turns out to be not precise enough, it could be worth trying to use a smaller cycle interval.

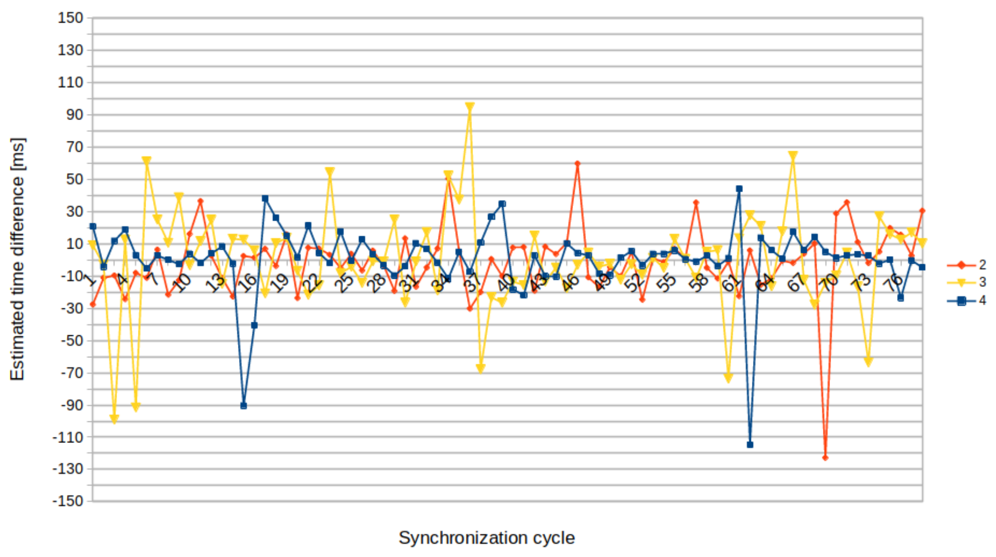


Figure 5.2: Experiment 3 with the highest cycle interval has more spikes, while Experiment 4 with the lowest cycle interval is more stable than the others.

In Experiment 5, only messages from the current cycle get accepted. It is also the first experiment using only two messages for the time synchronization, explained in Section 3.1.1. It is visible in Figure 5.3, that these changes make the time difference quite a lot more stable than they are in Experiment 2. The standard deviation of the time difference of this experiment improved as well. The newer time synchronization with two messages seems to not only be preferable because it generates less messages and thus less traffic, it has even improves the stability. But even in this experiment, the spikes still remain. A straightforward solution to remove those spikes would be to simply not perform the correction when the time difference is too big. The downside of this solution is that this would remove all intended high corrections as well. Especially in the beginning, the times can be quite far apart, so big corrections can be necessary. Making this solution work would probably get quite complicated. It turns out, that most time difference spikes happen when their time synchronization messages have a much higher Round Trip Time (RTT) than the other time synchronization messages. This high RTT makes the estimated time difference a lot bigger than it should be and thus causes an overcorrection. So for Experiment 6, all time synchronization messages with a too high RTT are ignored. This way, large corrections are still possible, as long as transmitting the time synchronization message does not take too much time. To determine, which RTTs are too high, the program keeps track of the average RTT of accepted time synchronization messages till now. A time synchronization message will be accepted and thus applied, if it satisfies the following condition:

$$RTT_i < 1.5 \sum_{k=0}^{i-1} RTT_k / i \tag{5.1}$$

This way, a higher RTT between client and server allows for more fluctuations in the RTT than if they have a smaller one. To initialize the average RTT, the first ten messages are accepted without applying Condition 5.1. If an outdated message arrives at this stage, it is still rejected and does not count to those initial ten messages. We can see in Figure 5.3, that Experiment 6 has no big spikes anymore and is generally very stable. Whenever the condition was not satisfied and the correction thus not applied, the graph shows a time difference of zero.

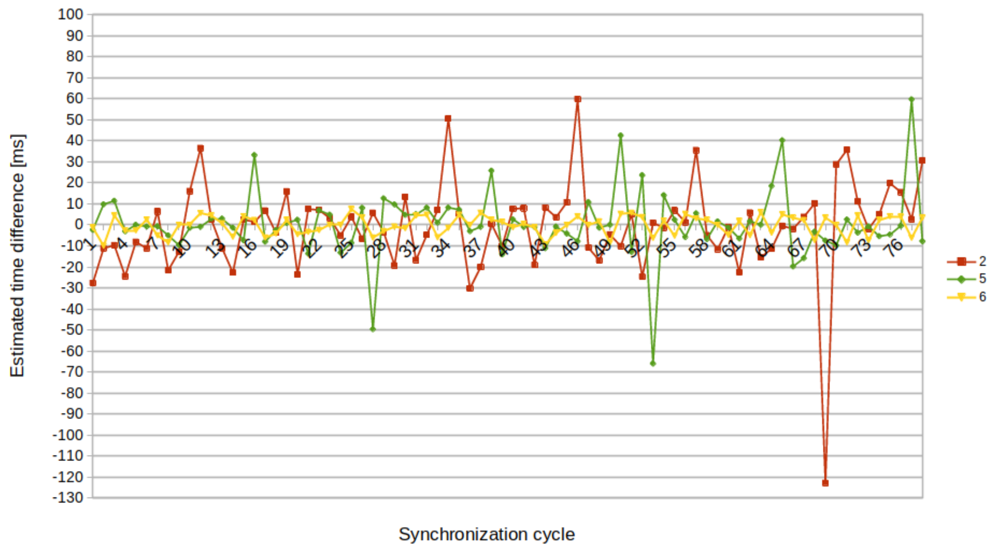


Figure 5.3: Using two instead of three messages and rejecting outdated messages makes Experiment 5 more stable than Experiment 2. Only using corrections satisfying the condition 5.1 finally removes the spikes and makes Experiment 6 very stable.

To better illustrate how this condition works in Experiment 6, Figure 5.4 shows the difference between all corrections the client receives, compared to the ones the client actually applies. Even though the client does not apply some of the received corrections, the later time differences do not get any bigger to redo these corrections. The standard deviation of Experiment 6 got smaller as well, reaching a value of 4.516 ms, smaller than all previous experiments. With the remaining time differences being this small, the synchronization would be precise enough to use for the game.

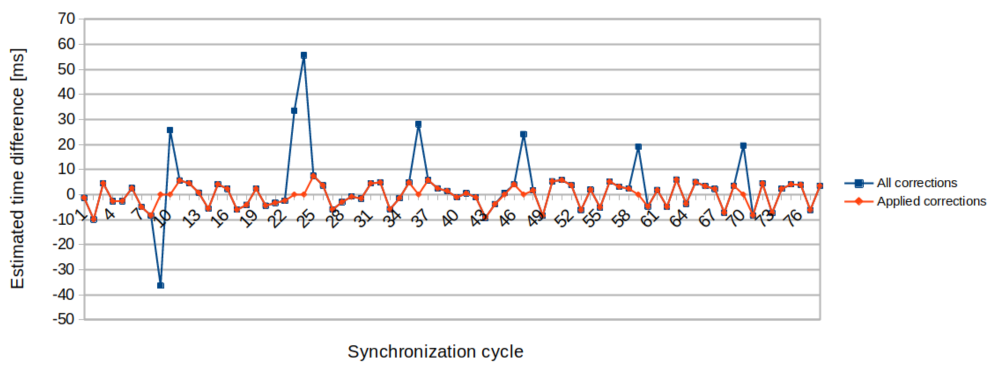


Figure 5.4: This chart shows all time differences the client receives and compares them those it actually applies as corrections. Big corrections are ignored, without really affecting the next time difference. This removes the spikes and results in less fluctuations.

All experiments to this point are performed with two computers in the same LAN, but this is a quite optimal environment with a low RTT between the server and the client. To test the time synchronization in a more realistic environment, we made ten experiments, where one of the computers was connected to the internet via LTE through a hotspot running on a smartphone. Those ten experiments are Experiment 7 to 16. In the LAN environment, the measured RTT was on average 26 ms, in the LTE environment it increased to 83 ms on average. Each experiment consists of 160 synchronization cycles and they are all performed with the same parameters listed in Table 5.2. The results of these experiments are provided in the following Table 5.4.

Experiment #	Average time difference $[ms]$	Average RTT $[ms]$	Time difference standard deviation $[ms]$
7	-0.015	84.857	5.731
8	0.375	83.355	6.339
9	0.126	82.709	6.264
10	-0.006	78.654	5.627
11	0.172	82.352	6.679
12	0.134	82.448	5.923
13	0.109	84.413	5.592
14	-0.059	80.946	5.408
15	0.068	81.526	5.011
16	0.161	87.826	7.713
Average	0.107	82.909	6.029

Table 5.4: Results of the ten experiments. Values from the initial phase are skipped for average and standard derivation, like in Table 5.3. The last row shows the average of the corresponding column. The experiment column identifies the experiments which are plotted in Figure 5.5.

The average correction is quite small. So it looks like most corrections cancel each other out and the time is changed by only a small amount. Over those ten experiments, the standard deviation of the time difference is on average 6.029 ms. As expected, it is higher than 4.516 ms, the standard deviation of Experiment 6 performed in the LAN environment. But it is still small, so even over multiple experiments, where the computers are connected over the internet, it appears that the time differences are small and thus very stable. A section of three of the ten experiments is shown in Figure 5.5. This time, the initial phase is also shown in the graph, that is the reason why Experiment 8 starts off with a quite high correction. In Experiment 8, there is a peak in cycle nine which causes a correction in the wrong direction and also by a too high amount. This correction, as well as the next one in cycle ten, would have normally been rejected, because they both had a too high RTT of 166.7 ms and 149.0 ms. But they were accepted, because they were still in the first 10 cycles were all not outdated corrections are accepted to find a meaningful average RTT. But at cycle 33, there is also a peak. This peak has a pretty high RTT of 115.6 ms, but it is just small enough to satisfy the condition 5.1. In experiment 7, the first 3 corrections are rejected. The server was probably not running the time synchronization yet, but the client already sent its request messages. When the response arrived, they were already outdated and thus rejected. As the standard deviation of the time difference already indicated, the figure shows that most time differences are below ten milliseconds. There are some occasional spikes, but even those remain around 25 milliseconds.

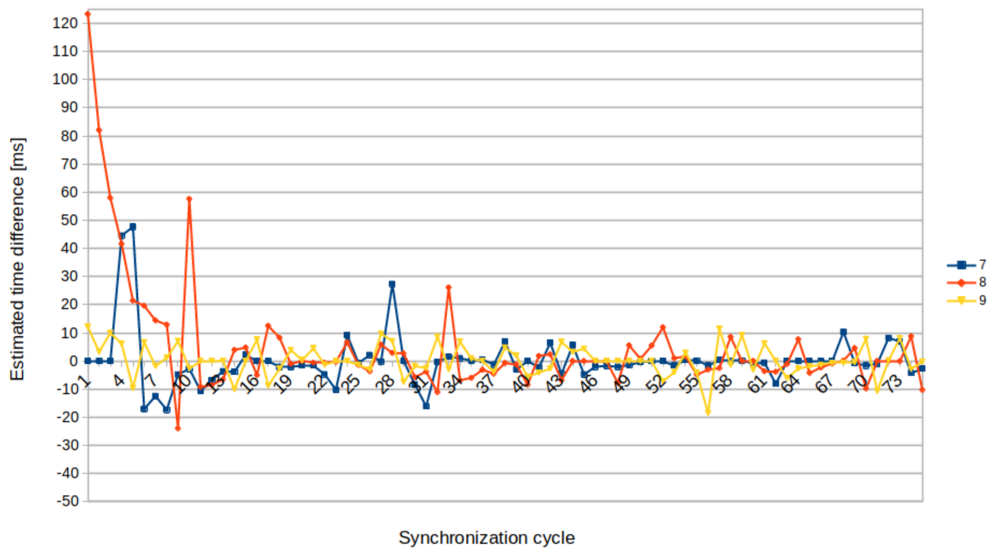


Figure 5.5: These are three of the ten experiments shown in Table 5.4. The corrections are very stable. While there are some occasional spikes, they are small enough to be acceptable.

The time needs to be precise enough to map an event to the frame in which it occurred or one of its neighbouring frames. If the game runs at 60 frames per second, one frame uses 16.66 ms for computation. So compared to the size of a frame, the time differences are still small and thus precise enough to use for this purpose.

Even though the time synchronization is quite stable using the parameters from Experiment 6, one could wonder why it is even this unstable. As described in the beginning of Chapter 3, the clock is a lot more precise, so why does it fluctuate so much?

One assumption of this time synchronization, like in NTP, is that the latency between client and server is constant. The more constant the latency, the more precise the synchronization gets. It can be seen in Figure 5.6, that the RTT fluctuates quite a lot. One possible reason for this behaviour is explained in Section 2.2.1. This reason would be magnified by the fact that during all time synchronization experiments shown above, messages were only read in the update function and thus only once per frame. Fortunately, the problem described in Section 2.2.1 can be solved for time synchronization. The solution is also mentioned in that section. Another reason might be that it is not clear how long it takes from calling the sending function for sending a message until it is actually sent. A similar delay could be found from the time a message physically reaches the computer until it actually appears to the program. If this delay is also varying, this could also be causing the unstable RTT. The highest RTT spikes that go up to 140 ms in LAN, could be caused by scheduling decisions of the operating system.

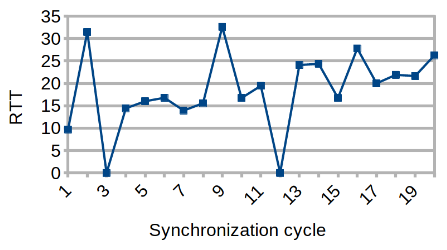


Figure 5.6: Even when removing the spikes using the Condition 5.1, the RTT is still unstable.

Chapter 6

Conclusion

6.1 Level Of Implementation

All basic features to play a match online on separated locations are implemented. Players are able to move their characters and collect stars and items. Among the items, they may use the green projectile, which destroys boxes and disburdens players of their stars. Stars lost that way move around the field and can be collected. Players can also place boxes, which cannot be passed by anyone. AI players are also functioning, so players may use them to hide themselves.

Client and Server are able to exchange messages. New messages can easily be added following the structure of existing messages. Steamworks is added to the game. It can be used for other features provided by steam, like achievements, tools for matchmaking and the Steam Cloud. Clients and server share a synchronized time. The update window is implemented on the server, allowing to recompute past states. There is a copy pool, which allows to manage copies of game elements that are needed for the update window.

6.2 Feedback

As the game is not yet updated on steam, there is no feedback from players on Steam yet. There is however feedback from friends that tested the game. The feedback is in general positive. Thanks to the implementation of the update window, movement feels a lot more fluid and natural. It feels almost as fluid as a local game. It is however possible that an increasing gap arises between the position of the client and the server if the character moves into the edge of an obstacle. This mostly results in the client teleporting the character to the position from the server, which is irritating to the player. Placing and destroying boxes through projectiles feels instant and the players did not notice the delay caused by waiting for the end of the update window even when specifically asked. The same is experienced when collecting items and stars. Movement of stars and projectiles also look natural and never caused any inconsistencies where it looked like a character was hit but this did not happen on the server. The placement of projectiles was however problematic. The delay from waiting for the end of the update window felt lagged to players and made it harder to aim accurately. This can be fixed by placing projectiles directly on the server and not waiting until the end of the update window, and if this is still not good enough one could use client prediction similar to how the client predicts character movement. So apart from projectile placement, the online mode works well.

6.3 Future Work

6.3.1 Completing The Online Mode

There is still some work that needs to be done before the game can be updated on steam. However the work that has been done till now solves a lot of basic problems, like movement of objects, player actions apart from movement and updates of static objects like boxes. So implementing the missing parts should be a lot easier as they can be based on the existing solutions. One part that is missing are items other than the green projectile. A game setup is not yet implemented, which would for example allow players to agree on a map to use. There is also no clear termination, so one player can start a new game while the other one leaves. As the game setup is implement, there is also no Matchmaking, so right now it is only possible to start an online game by adding the Steam IDs of the players to the code.

We intended to make a simple matchmaking between friends. A player can invite friends from her steam friend list [16] and then start the match. With this approach, less measures need to be taken against cheaters, as players probably cheat less when playing with friends. When a player still cheats, the other friends can urge her to stop and in the worst case simply not play with that player again. However this does not apply when a player plays online against strangers. Not playing with them anymore is not a threat to them as they can simply play with other players. Playing with friends also removes the need for a chat or a voice chat, as players can simply use third party software like Discord [17]. Playing with strangers would however benefit from a way to communicate, as they cannot agree on using a third party communication software without any means of communication.

But one could also implement a more involved matchmaking system which allows to play online against anyone. This has the advantage that a player can play the game with others even though she does not know anyone to play with right now. Such a matchmaking system could involve matching players with similar skill level or with a small latency to each other. As there are currently not that many players playing Hat Hunters at the same time, it might not be worth making an overly complex matchmaking system. Regardless of the type of matchmaking, the host has to be determined among all clients. This could be done by solving a small task on every client and the client which finishes the fastest becomes the host of this match. This should ensure picking the client with the most computing power, which is favorable as the host needs to compute more than a client.

The newer features in the game are not included in the online mode yet. Those features are a mode that lets players catch other characters with a hat, which removes the caught character from the game, eliminating characters that are hit by items too often and bushes, a new type of obstacle which grows over the field, gradually reducing the field size. Not all of these features necessarily need to be implemented in online mode, as only some modes can be allowed for online multiplayer.

6.3.2 Improvement Of Existing Features

If the need arises, one could further improve movement, for example by adding an update window to the client as well. This would, in theory at least, completely overcome the latency between server and client. However as the host is also a client, this would further increase the computational work the host has to do. It is still possible that the client and server position of a character separate, resulting in the character teleporting to the correct position of the server.

If the time synchronization turns out to not be precise enough, one could add a thread to the game which reads messages and adds timestamps in the time synchronization messages. This should improve the precision of the Round Trip Time (RTT) estimate, which in turn makes the whole time synchronization more precise. More information on this problem and its solution can be found in Section 2.2.1. Another way to improve the time synchronization would be to adjust the acceptance condition for time synchronization messages shown in Equation 5.1. In Section 5.2, there are examples of messages with a RTT that is quite a lot higher than the average RTT, but are still accepted. This is especially true for a high RTT, because the current acceptance condition accepts a wider range of RTTs for a higher RTT than for a lower. So it would be better to only accept messages which are very close to the average RTT. This could be done by storing the standard derivation of the RTT and only accept messages within a multiple of that range. However if the RTT suddenly does change and increase, for example when a different route is chosen, it would be bad if from then on all time synchronization messages get discarded. One option would be to still apply messages which do not satisfy the acceptance condition, but only to a maximum allowed amount. As a third option to improve time synchronization, one could try to decrease the cycle interval and thus send time synchronization messages more often.

6.3.3 Additional Features

There are still ways to further extend the game. One could add new features to the game itself to make the game more engaging for players. This ranges from adding new items to adding completely new game mechanics. In order to provide more motivation to keep the player coming back to the game, achievements [18] or a leaderboard could be added to the game. There is already the possibility to create custom modes by changing the game's parameters. One could add a method to exchange custom created modes with other players. This would allow player to get access to interesting custom modes while not having to make them themselves and could also increase the motivation to create custom modes as they can be shared with other players.

Bibliography

[1] Hat Hunters on Steam. [accessed 12. Mar. 2019]. [Online]. Available: https://store.steampowered.com/app/874880/Hat_Hunters

[2] N. Ingulfsen. (2018, Jul) Hat Hunters on Steam. [accessed 13. Mar. 2019]. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2018-FS/BA-2018-02.pdf

[3] G. Armitage, M. Claypool, and P. Branch, *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games.* Wiley, May 2006. [Online]. Available: https://www.wiley.com/en-us/Networking+and+Online+Games%3A+Understanding+and+Engineering+Multiplayer+Internet+Games-p-9780470018576

[4] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, Oct 1991.

[5] F. Bevilacqua. (2013, Aug) Building a Peer-to-Peer Multiplayer Networked Game. [accessed 26. Feb. 2019]. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

[6] Valve Corporation. Steamworks. [accessed 22. Feb. 2019]. [Online]. Available: https://partner.steamgames.com/?goto=%2Fhome

[7] Valve Corporation. Steam, a digital game distribution platform. [accessed 25. Feb. 2019]. [Online]. Available: https://store.steampowered.com/about

[8] R. Labrecque. Steamworks.NET. [accessed 25. Feb. 2019]. [Online]. Available: http://steamworks.github.io

[9] Valve Corporation. Steam Networking (Steamworks Documentation). [accessed 22. Feb. 2019]. [Online]. Available: https://partner.steamgames.com/doc/features/multiplayer/networking

[10] Unity Technologies. Unity - Manual: Online Multiplayer Overview. [accessed 25. Feb. 2019]. [Online]. Available: https://docs.unity3d.com/Manual/UNetOverview.html

[11] Unity Technologies. UNet Deprecation FAQ. [accessed 25. Feb. 2019]. [Online]. Available: https://support.unity3d.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ

[12] W. van Oortmerssen. FlatBuffers. [accessed 23. Feb. 2019]. [Online]. Available: https://google.github.io/flatbuffers/index.html

[13] S. H. Ahn. High Resolution Timer. [accessed 7. Mar. 2019]. [Online]. Available: http://www.songho.ca/misc/timer/timer.html

[14] Unity Technologies. Unity - Scripting API: Time.time. [accessed 7. Mar. 2019]. [Online]. Available: https://docs.unity3d.com/ScriptReference/Time-time.html

[15] Unity Technologies. Unity Profiler. [accessed 26. Feb. 2019]. [Online]. Available: https://unity3d.com/learn/tutorials/topics/best-practices/unity-profiler

[16] Valve Corporation. Steam Friends List. [accessed 15. Mar. 2019]. [Online]. Available: https://support.steampowered.com/kb_article.php?ref=1184-UOZV-2743&l=

[17] Discord Inc., "Discord - Free Voice and Text Chat," [accessed 16. Mar. 2019]. [Online]. Available: https://discordapp.com

[18] Achievement (video gaming). [accessed 16. Mar. 2019]. [Online]. Available: https://en.wikipedia.org/wiki/Achievement_(video_gaming)