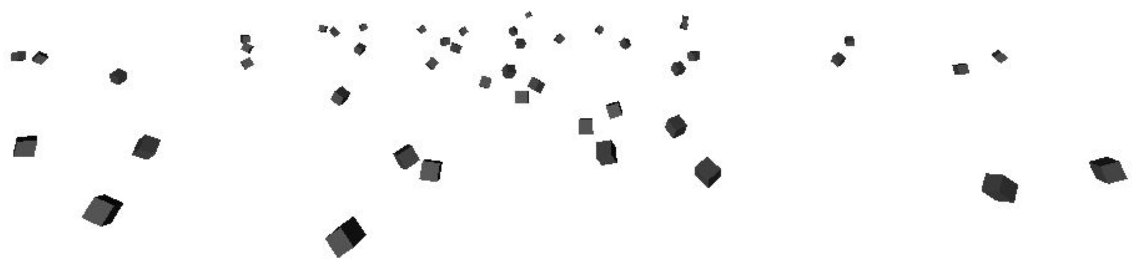
Challenges with the network programming of a simple multiplayer C++ Game



Johan Hagelin
Graduation Thesis
IoT18
Stockholms Tekniska Institut
2020-05-24

1. Abstract

This report is about the development of Splite, a multiplayer adversarial real-time 3D-graphical game made in the programming language C++ using UDP and OpenGL. The focus of this report is discussing problems that may arise when developing networked games and how to work to prevent them, shifting the focus from the end result of the game. The initially predicted problems just scratched the surface of networked games, thinking lag and game state stutter to be the biggest problem. As time went on, it became more apparent that problems with simple definitions have dire challenges when it comes to solving them.

1.1 Sammanfattning

Denna rapport handlar om utvecklingen av Splite, ett onlinespel med motståndare som utspelas i 3D, gjort i programmeringsspråket C++ med UDP och OpenGL. Fokuset av rapporten är att diskutera problem som må uppkomma vid utveckling av spel med nätverksprogrammering och hur man kan motverka problemen. Fokuset ligger inte på slutresultatet av spelet, utan om resan dit och förbättringar till framtida versioner. De ursprungligt förutspådda problemen utgjorde bara ytskiktet, då det blev allt tydligare att utmaningar inom nätverksprogrammering med enkla definitioner inte nödvändigtvis är lätta.

2. Forewords

I have always been obsessed with games. Ever since my brother lended me a Gameboy at the age of five I was stuck. Concepts surrounding games such as lag, latency, servers, and connection, are all regular terms that gamers use regularly, knowing what they are but not how they do what they do. As a learning system developer, every day has led me closer to understanding how things work beneath the surface, which increases my interest in it. It makes things seem less like magic and more like something I could do as well. I took on this project because I deeply enjoy learning about games and making them.

Every contributing user on the following programming discussion forum websites ought to be thanked for their many discussion threads, old or new, for freely giving insight into understanding concepts and solving common problems:

- Stackoverflow.com
- Softwareengineering.stackexchange.com
- Gamedev.stackexchange.com
- Reddit.com/r/programming

3. Contents

4. Purpose

The purpose of this report is to learn several things about developing networked real-time games. Not only to learn of the problems a developer has to face, but also how to combat them.

The goals of this project is to create a game from the ground as much as possible, and to learn about combating the challenges of real-time online communication. This is why C++ was chosen, as well as using UDP for not only time-sensitive and loss-tolerant data, but also connection upkeep and important messages. It was reasoned that there are many applications within the Internet of Things and other related software industries that may benefit from real-time communication, and that the teachings of this project would contribute to an expertise that would be attractive in the job market.

5. Background

The purpose of this chapter is to mention subjects that were initially thought to be in need of addressing when creating the game. Both problems that need to be handled, and concepts that ought to be created for the game.

5.1 Reliability system

Servers and clients need a way to communicate, and there is a demand for design of the content and the way that endpoints communicate in a way that facilitates a realistic player experience.

UDP is the IP (Internet Control) traffic protocol chosen for this project. It is a fast protocol as it is connectionless, but it does not guarantee that all packets are received, or even in the intended order. UDP is to be preferred in contexts where transmission times are critical and functionality is loss-tolerant, where the order of arriving packets are of less importance (Glenn Fiedler, 2008). Reliability in this real-time game is instead more defined by the capacity to send and receive quickly and frequently.

5.1.1 Connection

Maintaining a connection between client and server with UDP, a protocol that is connectionless, can be a challenge. The model for how connections are handled has to be defined at the application layer by the program by sending, receiving, and analyzing messages in a manner that, defined by the reliability system, extrapolates that a connection is present.

5.2 Network model

The network model of a game is the way client and server communicate what happens within the game. Such a model needs to be defined, as to facilitate the base upon which the game is built. The solution to this is discussed in chapter 6.3.

5.3 Game state stutter

A client accepts that the game is in the state that is reported by the server. Physical limitations of the server prevents the game state from being updated at a high frequency, leading the client to use the latest update from the server. This problem may be more apparent when not every message is delivered, leading to an even longer span between updates, as can be common with a connectionless protocol like UDP.

Let us assume a game in which a player appears to be running, such that a server message informs of where that player is. A naive game client would update the player on the client's screen only when the server delivers an update about where that player is. This leads to an image in which players appear not to be running, but teleporting ahead several units at a time, leading to unnatural behaviour and a potentially unsatisfying experience for the players.

5.4 Latency problem

The time it takes for a message to be delivered between server and client always takes time. The time varies; one message might take 5 ms to deliver, another may take 500 ms. It can be said that the longer it takes for messages to be delivered, the more lag there is. If the time to deliver is considered acceptable, lag can be considered absent. A real-time game that does not account for lag will suffer grave consequences. The problems which may arise are different depending on the philosophy of authoritative information, and who it belongs to.

With the assumption of server-side monopoly on the correct information, a lagging client may have an outdated state of the game. This naturally leads to the client acting in accordance to the outdated state, sending those actions to the server. The actions may not have the desired effects, as they are implemented on a much newer game state that the server considers to be the truth. This may lead to an unsatisfying experience for the lagging client, and potentially for the other clients as well.

6. Method

Since one goal of the project was to do things as from scratch as possible (see chapter 4), manual makefiles and mingw32-make were used to compile the projects with g++. The code uses the standard windows C/C++ library winsock2 to send and receive messages. OpenGL is used to output graphics and handle mouse and keyboard input.

6.1 Connection

Since UDP is the selected internet protocol, an implementation to handle connection has been developed. UDP is connectionless, meaning that the sender knows not if the receiver received anything. For the server and client to maintain a connection, messages must be sent between server and client continuously in a defined way at the application layer. If the messages arrive in accordance to the arbitrary rules for what a connection is, then a connection is said to be present.

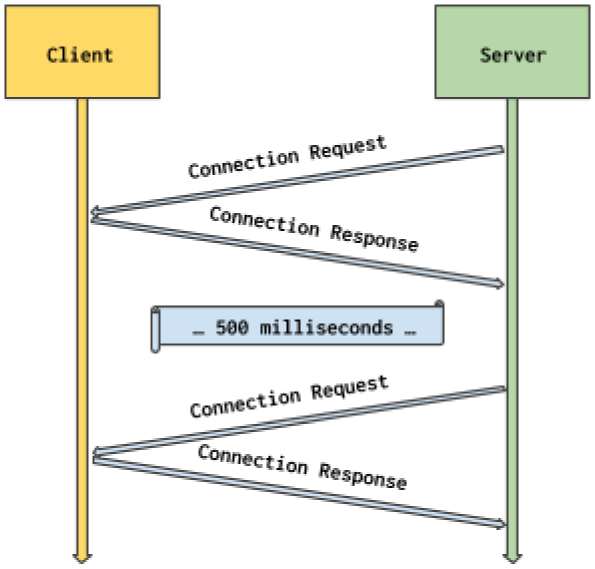


figure 1. Connection

The rules for this project is such that the server keeps a *last_seen* policy per client. If the *last_seen* property of a client has exceeded an upper time limit, then it is considered disconnected. In order for this not to happen, the server occasionally sends a "connection request" message to the client to give the client a chance to show that they are still there. The client should then send a single message in response to that, as seen in figure 1. If that message reaches the server, the *last_seen* for the client is set to the current time. This also occurs when input messages reach the server.

6.1.1 Connecting to a server

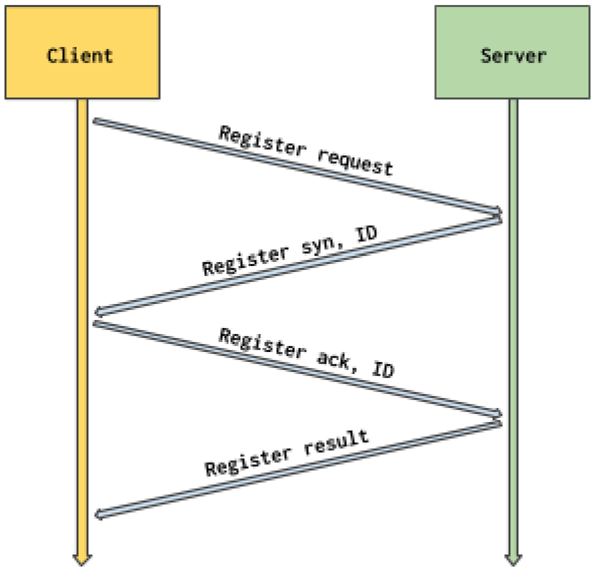


figure 2. Connecting to a server

The initial connection can be likened to the handshaking in TCP (figure 2). When a client wants to connect to a server, it sends a *register request* message to an IP address and port of a known server. If the server has a spot open, it will send a *register syn* message to the client with a unique ID, otherwise it will send a negative *register result* message. With a positive register result, the client will respond with the same ID in a *register ack* message, to tell the server that it knows its ID. The server then sends a *register result* message in which it tells the potential client if they are accepted or not. If the register was successful, the client is connected with the server, as seen previously in figure 1.

6.2 Network model

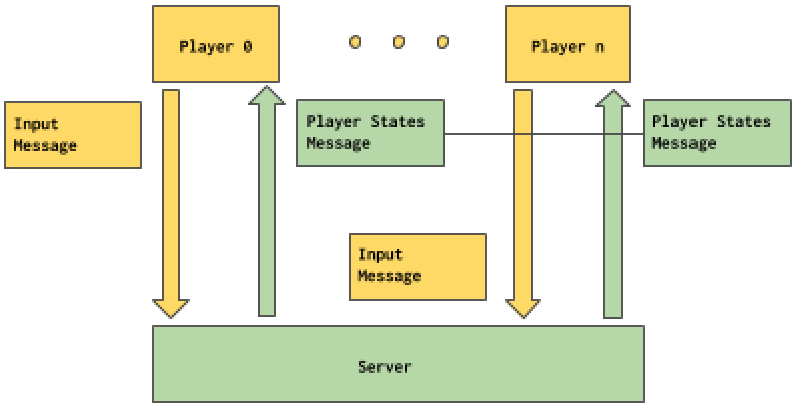


figure 3. Network model

Games can be defined by four abstractions: rules, game state, actions, and results. A player performs an action according to the rules in the context of the game state, resulting in a change of the game state. In a multiplayer game with client-server topology, the server delivers updates that alter the game state of the client, and clients only send actions to the server, which alters the game state on the server in accordance to the action.

The network model is designed in that manner. The server has a monopoly on the correct information regarding player states and game states. The players may only contribute with input, and everything that actually happens in the game is determined on the server, which the clients promptly accept as authoritative, letting go of their own interpretation of the game state.

Player states are periodically broadcasted to every player at once. Player input messages are sent sporadically every time the player makes an action, either making a movement or moving the camera (figure 3). **Note:** The direction the player is facing is the only piece of player state knowledge that a client will not be overwritten by the server, as such would increase the effects of local state snapping (see chapter 7.2).

6.3 Message design

In order to read the content of a message properly, some information of how to read it is contained in the message. The first byte in every message describes the type of message. Examples of message types are: *register request*, *register result*, *input*, *player states*, *kicked*. After the message type the rest of the message content follows, typically with a client ID if sent by clients.

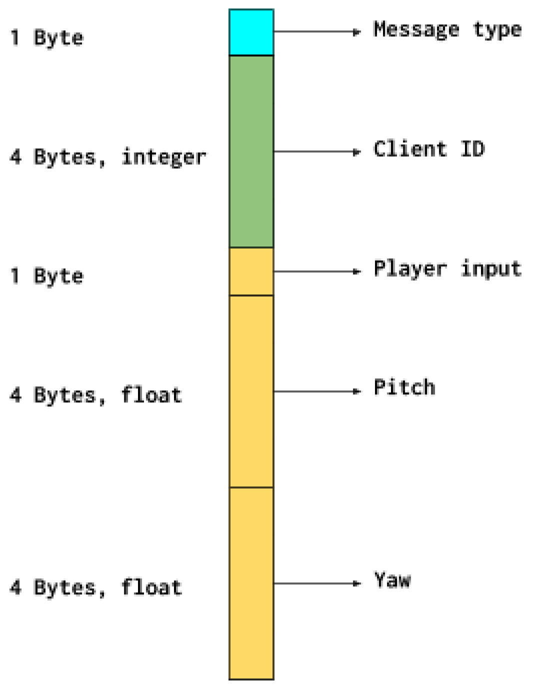


figure 4. Player input message design

Messages are designed with data tightness in mind. For example, the message content for player input contains the message type, a client ID, player movement input, pitch, and yaw. The player input is represented with a compressed byte in which most of the bits carry significance. The first bit is forward, the second bit is backward, the third bit is up, etcetera. Then follows yaw and pitch, which are both 32-bit floating point inputs for camera and direction. In total, that is 13 bytes of message content besides the message type. These bytes are packed serially after one another in the buffer, as seen in figure 4.

When messages are received by an endpoint, either server or client, the message is deserialized and extracted from the message buffer in the same order as they were serialized on the sending endpoint. That is to say, for a player input message, the server reads a byte, then a 32-bit integer, then a byte again, and finally two 32-bit floats, into a corresponding player input data structure.

6.3.1 Message design of varied-size message

Messages are typically set-size. However, sometimes the point of certain messages are inherently varied in the size of the content in the buffer, depending on unpredictable data. In those cases it is also written into the message how to parse it further. An example of such a message type is player states, where the variation lies in the amount of players connected. A player state has 36 bytes of data in total, and contains information about client ID, position, velocity, pitch and yaw. As the amount of players varies, a byte, *num_players*, is written into the player state message to indicate the amount of player states in the message content. The receiver, when reaching the *num_players* byte, then reads 36 bytes *num_players* amount of times, into separate player state objects.

6.4 Ticks

The server has a tickrate of 60 times per second, which gives approximately ~16ms per tick. The tick system employs a reversed flow, where the loop begins by listening for messages. However, the tick system has a tick beginning by checking the connection of clients and sending messages if appropriate (see chapter 6.1). It then ticks (simulates) the game, taking all the actions from the players into account, then broadcasts the player states and sends other various game state messages to relevant players if necessary. The remainder of the tick waits and listens for messages from clients, responding to register messages (see chapter 6.1.1), and logging player input and connection responses. The next tick then starts with that new information, as seen in figure 5.

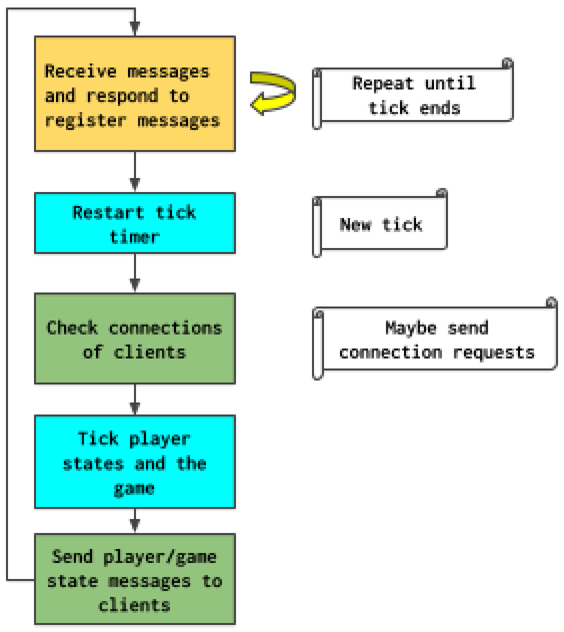

Receive messages and respond to register messages
Repeat until tick ends
Restart tick timer
New tick
Check connections of clients
Maybe send connection requests
Tick player states and the game
Send player/game state messages to clients

figure 5. Server tick flow.

6.4.1 Client ticks

The structure for client ticks is similar to the server. It begins with checking for connection, then processes input, sends the input to the server, then ticks the game based on the latest information from the last tick, displays the graphics, then listens for messages for the remainder of the tick time, see figure 6. **Important:** note that only *new* input is sent to the server.

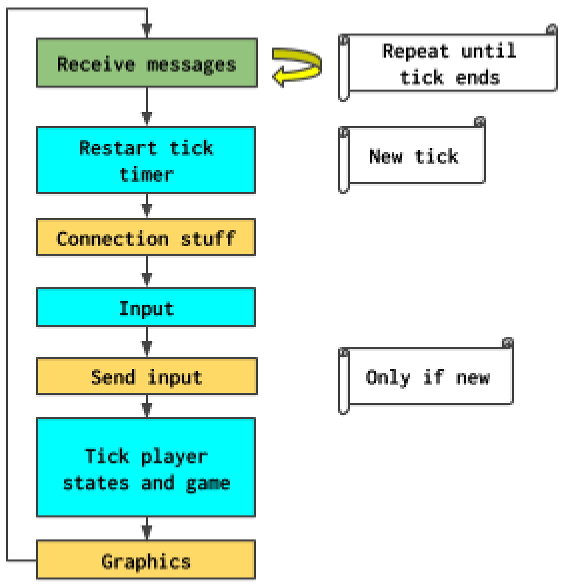


figure 6. Client tick flow.

6.5 Latency

Lag is defined here as the time it takes for a round-trip message between endpoints responding at the application layer. It is a factor that is unpredictable in quantity and variation. It was not particularly handled in this project. However, there are ways of battling lag (Mark Mennell, 2014).

6.6 State synchronization

The problem of game state stutter (see chapter 5.1) requires a way to handle the downtime between updates. The chosen method for this project is for the client to simulate the game given the latest information from the server. The only simulated thing here is player movement, where the relevant data points are position, velocity and direction. The rules dictating the movement of the player according to these data points are the same on server and client, meaning that given the same input, both the server and client tick simulation would give the same output.

Every tick on the client applies the movement of each player by applying velocity in their direction, altering the position values of the player states. This happens no matter if an authoritative update has arrived from the server or not. Unsightly in 2020, there is no acceptance of variation or diversity on the client side. As a side note, there is no retardation of velocity in the game rules, which means that a player has a constant velocity in the direction of their choosing, and stops instantly when inputting not to move.

6.7 Graphics

3D Graphics have been implemented using OpenGL from scratch. Players are represented by cubes placed at the positions of the players. The player cubes are rotated based on where the player is looking to make it appear like players are alive. All other game objects are also cubes.

6.8 The game

Players may move forward, backward, left, or right. They can also jump, though for no reason as it grants no advantage over other players and there is no air control. The game is about moving into cubes and consuming them. Whichever player has consumed the highest amount of cubes after all cubes are consumed is declared the winner. The background fluctuates colorfully for the winner and darkly for the losers.

7. Result

The game ended up in a manner that works fairly well. Players can play the game together and interact with game objects. The game experience can be considered rough, but working. There was no time to develop the handling of latency to the degree that a multiplayer game could be argued to need, and there were other more apparent problems that took the spotlight, namely simulation deviation and snapping.

7.1 Localized simulation deviation

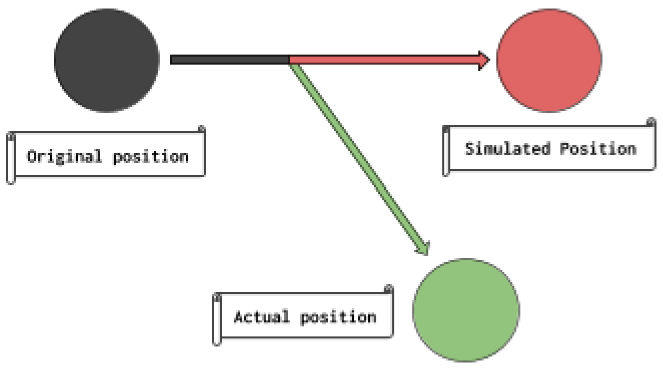


figure 7. Deviation between simulated position and actual position of a player.

Player A knows of the direction and velocity of another player B. B will therefore move for player A. However, there are cases in which player B appears to keep moving *even though* player B inputs to stop or change direction. This is caused when networking issues hinder game state updates to reach the observing player, resulting in simulation of outdated information. Figure 7 displays a representation of how positional information can deviate between localized simulation and server information.

7.1.1 Snapping

What follows the deviation is a direct effect of a network model where clients apply any update instantly. As long as the connection is upheld (see chapter 6.1), player and game state updates will eventually arrive. The player with deviation will at that point accept the game state update (see chapter 6.2) and thus the position of the players. Players therefore appear to snap into place, likened to instantaneous teleportation.

The effects of this phenomenon are more noticeable as messages are lost more frequently or latency grows larger. The difference between actuality and simulation grows bigger. The bigger the gap, the more noticable the snap.

7.2 Local state snapping

Local state snapping occurs when a player sends differing input to the server, but the message gets lost. The player appears to move locally, moving in the direction of their choosing. When the periodical game state updates are received by the player once again, the position of the local player is set to whatever the server thinks is correct (see chapter 6.2), resulting in a snap for the player. This may be more disturbing for player experience than observing other players teleporting, as the camera snaps with the new player position (Note: The direction of the camera is unaffected). This disturbs the flow of the player, as they may have to readjust their strategy and movement patterns to the new position.

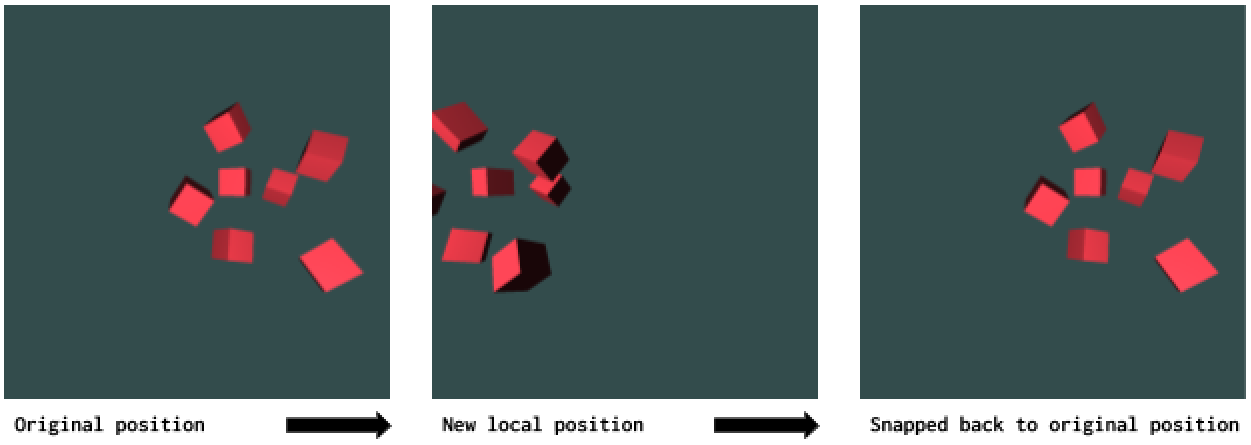


figure 8. Local snapping.

Figure 8 displays an example of player input not being registered by the server. The player inputted to move to the right, but was snapped back to the original (server) position.

Another similar but different local state snapping may occur. Assume that a player is running, then stops and submits no new keystrokes or mouse movement beyond the stop, but the new input to stop doesn't reach the server. The server will still assume the player is running, as the input from the player has not been altered, and continue to alter the player state in accordance to the direction and velocity. The player will receive the periodical player states and accept them as usual and accept them. This results in hardcore game state stutter that is not only disturbing, but very incorrect to what the player intended. The player will be stuttering forward at the server tick rate, which is lower than the client tick rate, making the stutter more apparent.

7.3 Movement issues

There appears to be an issue with jumping in the game where players cannot jump if they are moving forward or backward, if they are connected to a server. The player moves forward and upwards before stuttering, that is to say being reset to the ground after a low amount of ticks in the air. However, if there is no connection, and the players are simulating ticks by themselves, there is no problem with jumping.

A similar issue appears when players attempt to move in a diagonal direction relative to the front of the camera (that is to say a combination of forward or backward and left or right). The player moves along in accordance to forward or backward, moving locally to the desired diagonal position for a single tick, but ultimately snapping back to reality, that is to say just forward or backward, very quickly because the server said so.

The cause for these issues are unknown and would require further investigation.

8. Discussion

The issue with simulation deviation stems from the fact that the server and client will never be able to communicate in a totally synchronized manner such that the order and timing of the messages alone can run the game in the intended fashion. The phenomenon that causes this is the variation and fluctuation of the time it takes for a message from sender to receiver. The term that refers to this variation of latency is called *jitter* (Logan Rivenes, 2016). There are several solutions to combat jitter. Prediction is a good place to start, but it can be improved from the arguably primitive version in this project. Right now, simulations on the client are simple. If the client does not receive updates to the game, it will simply tick onwards with the information from the previous tick and continue with life.

8.1 Prediction history

One improvement that can be made to the simulation is to keep a history of previously simulated and authoritative states. Not only this, but also keeping a prediction timeline into future ticks as well. In such a solution the takeaway is that a certain deviation is tolerable, meaning that the client does not immediately apply updates, or part of updates, if the deviation of the update and the simulation is considered acceptable. Figure 9 represents a scenario in which some deviation is acceptable. The constant micro-snapping from small deviations where there is no tolerance may be too big a price to pay for a near-completely accurate network model. This approach argues that a slight deviation is preferable to snapping, from the player experience angle at least.

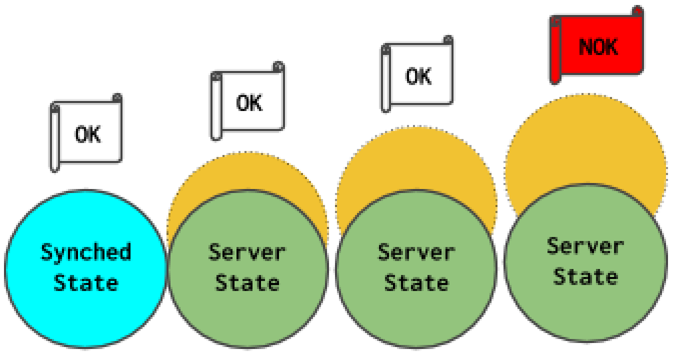


figure 9. Representation of deviation becoming too great.

(Sidenote: Not only that, but there are cases where the deviation decreases by natural means, such as when players simply move in small areas or realign with similar positions again. In those scenarios the acceptance of deviation means that no drastic measures like snapping has to occur.)

A way of doing this is discussed in an educational walkthrough of a UDP multiplayer C++ game, very similar to this project (Joe Best-Rotheray, 2017). In his solution it is assumed that the client and the server are both operating on the same tickrate. This leads to some mistakes that he corrects in a later article (Joe Best-Rotheray, 2019), but is irrelevant for the example given here.

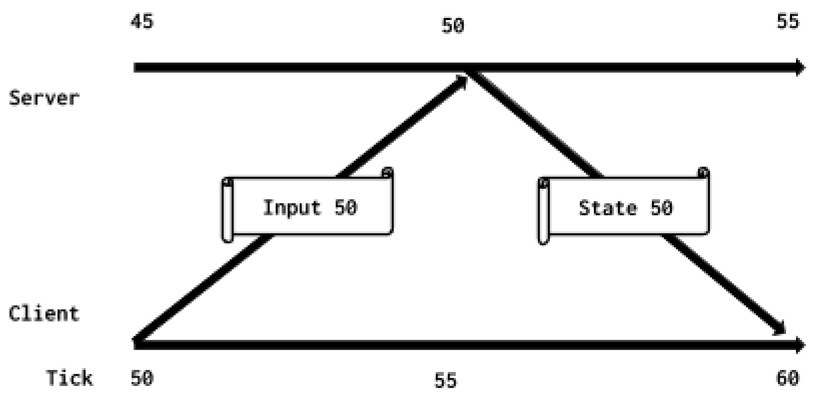


figure 10. Stylized client-side prediction graph (Joe Best-Rotheray, 2017)

Like the solution for deviation in this project (see chapter 6.6), Joe describes a way to handle downtime between server messages. In Joe's solution, clients perform a prediction of resulting changes to game state based upon their most recent input.

Looking at figure 10, the client appears to be 5 ticks ahead of the server at all times. The client not only simulates 5 ticks ahead, but acts as if it is there already. State updates from the server leads to a comparison between the predicted game state and the delivered game state. In the case that there is too big of a deviation between the prediction and the server state, there is considered to be a misprediction. That is to say that the 5 ticks ahead that the client acts upon has been faulty. Thereafter, the history is "replayed" from the just-now received state up until the current predicted state of the client (Joe Best-Rotheray, 2017). This approach may also result in snaps due to jitter and out-of-order messages, but it is an improvement in theory, and perhaps also in practice if adhering to the corrections made in a later article (Joe Best-Rotheray, 2019).

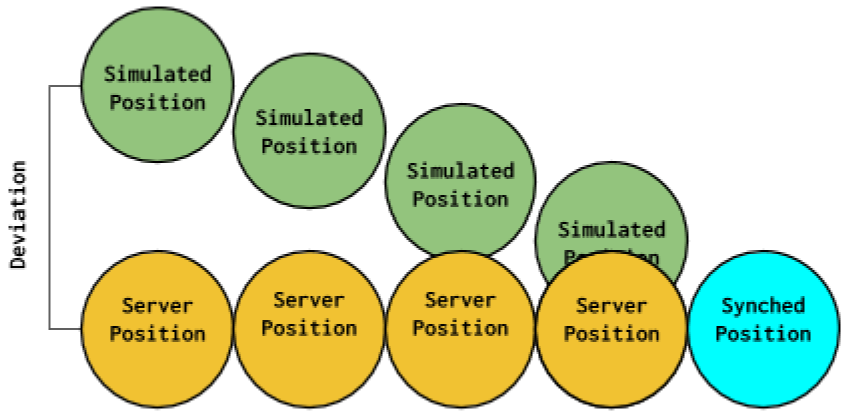


Figure 11. Example of interpolation between deviated simulated data and authoritative data over time.

8.2 Preventing snaps

Preventing snapping can be done by interpolating between predicted states and authoritative states. Instead of instantly accepting the new update, either as a re-prediction a la Joe or as this project does it, the state gradually tends towards the authoritative data values over time. In the event of a player standing still on the ground with a variance in position due to accepted deviation, interpolation may be observed by watching the player glide over to the authoritative position. This can be noticed even in offline games like Skyrim, where corrections to positions are not instantaneously applied, but interpolated with the faulty ones in order to facilitate smooth moonwalks.

Figure 11 depicts interpolation between positions over time such that the deviation between the authoritative position and simulated position lessens over time. The interpolation can happen linearly, exponentially, logarithmically, or whatever "...-ly" is appropriate. It varies and is up to the individual network model depending on the goals and priorities of the game in question.

8.3 Preventing local state snapping

A reason why local state snapping occurs in this project is that some input messages do not reach the server. A simple yet effective way of preventing local state snapping is by simply sending input at every client tick instead of every time there is a new input. This would likely increase the amount of input messages that reach the server, increasing reliability (see chapter 5.1).

It would also increase the bandwidth. Increasing the throughput of network messages may have unforeseen circumstances. The message buffer may get congested before the server has had time to read every message, resulting in a tick with unhandled messages leftover for the next tick. The leftover messages are handled the next tick, resulting in even less time for even newer messages, leading to a vicious cycle down to application layer latency hell, where messages just take longer and longer to handle before the buffer is so full that most messages are dropped and the ones that make it through are outdated already.

8.4 Combatting jitter

As jitter may cause messages to arrive out of order, it may be a good idea to number the packets such that outdated data is handled appropriately, either by disregard or assimilation. With a prediction history it would be possible to resimulate ticks from when the message originated or was thought to arrive at. The possible implementation of this is left as an exercise to the reader.

8.5 Closing thoughts

Throughout this project several things have been learned. Creating a working game from as much of a scratch as is tolerable has been beneficial to increasing the expertise in network programming. The subjects mentioned in chapter 5 only scratched the surface of all the things that need to be handled when creating a real-time game. Networked games open up a big world with many topics, much more so than can be covered in a report of this caliber. How to handle all these networking things is often a game of whack-a-mole, where the solution of one problem leads to another problem. It is a landscape of constant learning.

9. References

Glenn Fiedler, UDP vs TCP, https://gafferongames.com/post/udp_vs_tcp/, published 2008 october 1, retrieved 2020 may 15.

Joe Best-Rotheray, Making a Multiplayer FPS in C++ Part 5: Client-Side Prediction, https://www.codersblock.org/blog/multiplayer-fps-part-5, published 2017 december 31, retrieved 2020 may 14.

Joe Best-Rotheray, Making a Multiplayer FPS in C++ Part 8: Client-Side Prediction Revisited, https://www.codersblock.org/blog/multiplayer-fps-part-8, published 2019 june 7, retrieved 2020 may 14.

Logan Rivenes, What is Network Jitter?, https://datapath.io/resources/blog/what-is-network-jitter/, published 2016 june 21, retrieved 2020 may 14.

Mark Mennell, Making Fast-Paced Multiplayer Networked Games is Hard, https://www.gamasutra.com/blogs/MarkMennell/20140929/226628/Making_FastPaced_Multiplayer_Networked_Games_is_Hard.php, published 2014 september 29, retrieved 2020 may 12.

10. Appendices

Github for the project: https://github.com/mrryyi/Splite/

11 Glossary

- *Application Layer*
  The layer at which applications operate. The application layer does not define the details of the IP layer protocols, but rather communicates through it.
- *Broadcasting*
  Sending the same message to all connected clients.
- *Camera*
  Analogy to explain what sees the observable game world.
- *Game state*
  State of the game, an abstract concept (not a data structure), altered by changes to player states and game objects, etcetera. Not to be confused with game state messages that are specific in what they alter.
- *Message*
  Data sent over the internet.
- *Message content*
  Data buffer content of a message that is not overhead, that is to say, message content excludes the UDP header, as well as the networking layer header.
- *Player state*
  State of a player, containing data such as position, direction and velocity.
- *Pitch and yaw*
  Angles along aircraft principal axes. Related to camera direction maths.
- *Retardation*
  Opposite of acceleration.
- *Snapping*
  Snapping occurs when the client accepts states from the server and sets the local states to received states instantly, possibly resulting in a jagged player experience
- *Authoritative*
  Correct according to the governing body in the network model, often the server.