Delay-based vs Rollback solutions to lag in peer-to-peer online games

Seth Messier (V00975362) & Dario Monette (V00983158)

Introduction

Online gaming in its current state is a multi-billion dollar industry, with a large share of said worth being produced from online multiplayer games. Players expect quality experiences to engage with, and in return for providing them, give developers money through some sort of monetization model and/or upfront payment. Given the scale at which many large multiplayer online games run in the modern day, with the most popular ones having millions of concurrent players connected at once. Having to acquire and maintain the architecture to be able to support such a large number of players with classic client-server connections, hence why more and more research has been done into trying to find ways to allow for these online games to be run using peer-to-peer connections. But just being able to support a large number of players does not mean much if said players are not retained due to sub-par quality of the provided experience. When it comes to quality, it is up to the developer to ensure their experiences match player expectations, but once the sending of packets to other players and/or servers is involved, they must hand over some level of control to the internet architecture. This handing-off takes control away from the developer and ties game-quality to the current player's network in some capacity. Application level networking solutions (known as "netcode") can be integrated to soften the detrimental effects of poor network conditions, both for local and remote players, and ensure that virtual experiences are enjoyable even when network conditions are sub-optimal. In this paper we evaluate two common types of netcode when integrated with a simple online game, and seeing how they act under certain network conditions, to give developers better insight into what netcode solution may be best for their application. This report will be split into multiple sections: Problem Definitions, in which we outright define what "poor quality" means in regards to online games and experiences; Application Overview, in which we cover the architecture and software from/in which all tests in this paper were ran; separate sections for Delay-based and Rollback netcode respectively; and a section comparing both netcode types and discussing possible hybrid solutions.

Problem Definitions

Desynchronization and Delay

It is important to note that phenomena such as network delay and packet loss are "problems" that lead to worse connections for players, but developers of online games are usually not in a position to reduce these effects themselves. The problems that developers must deal with are instead the effects of these networks on their games. These effects are not necessarily due to a poor connection, but due to the mechanics of any sort of networked application. The main problem that stems from the mechanics of networking is the need to maintain a constant and consistent gamestate between all players at all times. When playing with multiple people on the same device ("local multiplayer"), this is trivial, as there is only one machine that needs to maintain said state. But when more than one device needs to maintain this game state, there is the possibility that said states differ in some way from one another. At any point in time in an online multiplayer setting, there are two states the network can be in: a synchronised state, in which all clients possess identical game-state information, and a desynchronised state, in which at least one client has differing game-state information that differs from other clients. Due to the fact that information takes time to travel over the internet, there is no way to avoid desynchronisation outright, as This state of desynchronization is the main problem that netcode wishes to minimize the effects of, trying to emulate the quality of local multiplayer in which players can trust that the state they are being shown is true to the true current game-state.

To measure desynchronization, we use the concept of delay. The delay of a connection is the time between a packet being sent at one end and received at the other, or in other words, the time during which the game-states within two connected peers differ after a change in state is made on one peer. Desynchronisation is the problem that netcode wishes to address, and delay is the metric by which we measure the magnitude of the problem. Desynchronisation can lead to the following phenomenon which can impact the quality of online experiences in different ways:

- Local players have a state with updates that remote players lack (Local-ahead)
    - The local player is able to execute actions that other players are possibly unable to anticipate.
- A remote player has an state with changes that our local game does not have (Local-behind)
    - The local player may make actions that they would not have if they were to have the true game state at that moment.
- The game states of multiple players changes simultaneously in different ways
    - Game-states may conflict with one another if two conflicting actions were performed simultaneously. This must be resolved in some way.

By understanding these phenomena and how they affect the player experience, we can begin to design ways to try and minimize their impact on player enjoyment.

The Effects of Desynchronization on Quality

Like any networked application, online games are affected by delay in different ways, akin to the utility functions described in a paper by Shenker [1] that was discussed earlier in this course. But instead of the utility function being a function based on bandwidth, it is a function based on delay. Turn-based games, in which players do not act simultaneously, are akin to an “elastic” program from the original paper. Delay can cause larger waits between turns, but as long as clients wait until all information from the current player’s turn is submitted before moving onto the next, no problems will be had due to desynchronisation. Meanwhile, faster-paced games such as fighting games that have more than one player acting simultaneously and require players to make split-second decisions are more akin to the hard-real-time type of program discussed. If the delay is too large, players might be unable to react to anything their opponent is doing, making the game practically unplayable for them. What type of game is being developed can change what type of netcode solution is needed, or if netcode is really necessary at all.

Combating the Effects of Desynchronization

Deciding which effects of desynchronisation we wish to combat is important, and will change how said netcode type helps improve the experience of players. For example, perhaps our netcode works with the goal of preventing players from entering a conflicting game-state with another player though preventing certain inputs at certain times. This might achieve its goal of preventing players from entering states that would cause confusion or break the rules of the game, but it wouldn’t solve the other two problems desynchronisation can cause mentioned prior. In this paper we are concerned with only the first two problems mentioned in the above list (local-ahead and local-behind), as the game developed to test netcode types does not have any "invalid states" for players to enter due to player-to-player actions.

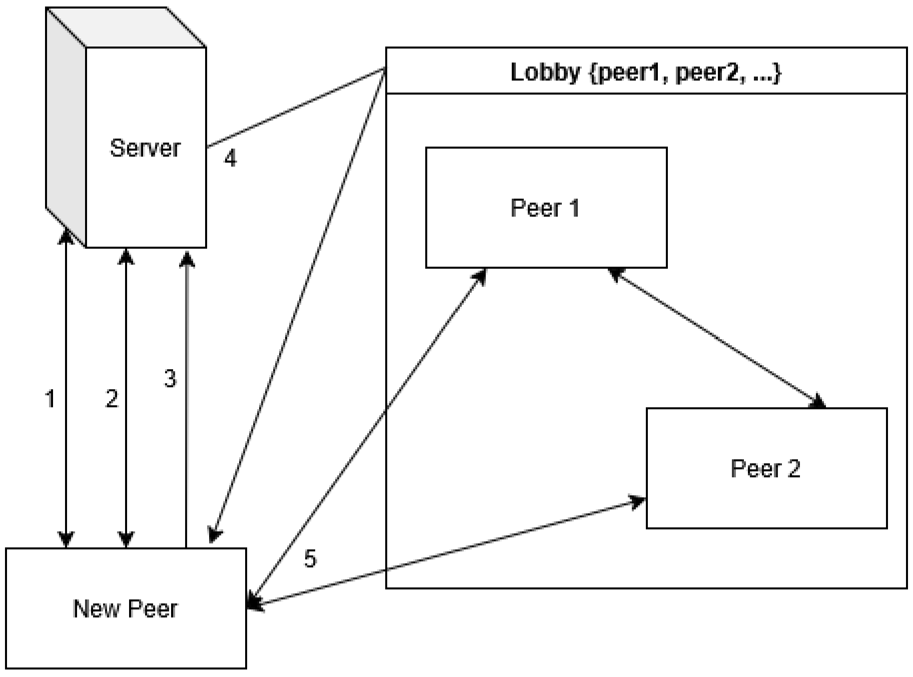
Application Overview

The application we made for this project consists of a simple browser-based online game in which players can connect from any computer with an internet connection. The game itself is written in JavaScript, making use of the WebRTC protocol to allow for peer-to-peer connections to be established through the web browser [2]. Due to how WebRTC functions, we need not just STUN/TURN servers to facilitate NAT traversal, but some intermediate server that peers can exchange information through before NAT traversal can be established (a "signaling server"). This signaling server also acts as the server that serves the game to players and allows for users to look at currently existing lobbies. This server is written in TypeScript, and uses WebSockets to establish connections with peers for the sake of WebRTC connection establishment. The STUN and TURN servers we use are free ones hosted by Google. Once a player has joined a game, they only maintain a connection to the server for the sake of allowing other players to connect with them and allowing the server to display the number of active players within a lobby (assuming dropped WebRTC connections are players that left their games).

The Process for Serving and Connecting Peers

The process for serving the game to users and connecting them to their peers is as follows:

1) The user gets a webpage holding information in the currently existing game lobbies from the web servers. (This page can also be used to request a new lobby if allowed)
2) The user uses an HTML GET request to both get the game to run as a web page and specify which lobby they want to join. (This also is where the netcode to be used is decided, though it is tied to the lobby)
3) The user loads the webpage and uses a specific lobby-dependent link to connect to the webserver with a WebSocket connection. If the lobby is full, the connection will be denied and the player will be given a "Lobby is Full" error. They maintain this connection with the server to allow the server to know that it is still a part of the lobby.
4) The user gets a list from the server of a list of WebSocket connections that allows the peer to coordinate a new WebRTC connection with each peer. This process follows the standard WebRTC connection establishment protocol, using our web server as a signaling server.
5) The user, once connected to all peers with WebRTC connections, broadcasts a "join" packet to all peers to let them know it has successfully connected. Once this is done, all game-related information is sent through these WebRTC connections.



Delay-based Netcode

Introduction

Delay-based netcode works with the goal of eliminating the possibility of the “local-ahead” state described earlier, in which players are seeing a gamestate that is ahead of their peers upon making an input. This is done through introducing input delay to actions made by the local player in an attempt to emulate the delay experienced by remote peers. The effectiveness of this method of netcode is dependent on how the amount of delay to calculate is applied. It should be noted that this strategy does not inherently change how the game-state of the rest of the application is presented, nor will it work to prevent aforementioned local-behind states.

Core Mechanism

Delay-based netcode requires two mechanisms to function: the delay calculation function, and the system for actually introducing input delay to the player. The delay system runs constantly, using information from packets sent by peers to determine the amount of time packets take to travel. Our system used a “ping-pong” system, in which clients send “ping” packets to their peers, those peers calculate the delay of the packet based on the attached timestamp, and return a “pong” packet containing the calculated delay. How delay is calculated does not matter as long as the measurement is a good estimate or exact value of the connection delay, how we combine the delay of multiple connections to produce a singular delay value (referred to as “effective delay” in this report) to be used by our input delay system. How the input delay system should be implemented is dependent on how the game program itself is constructed. One could use a buffer system for all inputs, or just delay the visuals of the game. Our implementation does the latter, showing the player their delayed position, but keeping their actual position stored (with the same logic applying to projectiles).

Calculating Effective Delay

This section assumes that the delay data being sent to each peer is truly indicative of the actual delay in the connection between them and their peers. Effective-delay should in theory predict the delay between the local peer and all other peers as best as possible, which means that with only two peers in a network, it becomes trivial to calculate (effective delay = singular connection delay). How we go about calculating effective delay is much more challenging when we have to assimilate the delays of more than one other connection into a single value. We tried a few different methods in our experiments, and in the end found two that showed promise.

DELAY-MAX

Let D_i be the delay of our peer$_0$ to (peer$_i$) where 1 <= i <= # of peers = n.

$D_i = Max(delay\ samples\ for\ connection\ i)$

$Effective\ Delay = Max(D_1, D_2, ..., D_n)$

By calculating delay this way, we try to assure the player is delayed enough as to not be ahead of any other peer in terms of their own state. This is the biggest benefit of this method, consistently giving this property as long as the supplied delay data is accurate. Due to only taking the value of one statistic, if the maximum delay value is unstable, effective delay will also quickly change, which can cause discomfort for players due to rapidly changing input delay. Along with this, it disregards the delay of this peer to all other peers besides the one with the worse connection. In a situation where only one peer has a poor connection, all other peers but them will have to deal with large input delay, which can be seen as unfair. This matters less if the maximum delay is not much larger than the median delay of all connections.

DELAY-AVG

Let D_i be the delay of our peer$_0$ to (peer$_i$) where 1 <= i <= # of peers = n.

$D_i = Average(delay\ samples\ for\ connection\ i)$

$Effective\ Delay = Average(D_1, D_2, ..., D_n)$

Calculating delay this way provides a smoother effective delay value than is less prone to quick changes and provides players with a more predictable amount of input delay. But the delayed state shown to the local player is guaranteed to still be ahead of at least one other remote peer, lowering the effectiveness of the netcode at achieving its main goal. This is the main tradeoff between this and DELAY-MAX, one provides the most accurate state representation by sacrificing player comfort, and the other reduces the effectiveness of the system in showing an accurate state for giving players a more comfortable experience playing.

Performance of System

The delay-based system was tested under the effects of delay and jitter to attempt to see what scenarios caused the system to exhibit poor performance, and which the system excelled under. A qualitative approach was used, in which footage of both systems running the program were compared to see how delayed they were at any given time.

Constant Delay:

Under conditions in which all connections to peers have little to no jitter, but a constant amount of delay, delay-based netcode is able to provide an accurate representation of the player's local state on its peers. If peers have a similar level of delay, DELAY-AVG produces a good estimation on the ends of both peers, but as delay of peers grows further apart, DELAY-AVG produces less accurate results. In this case, DELAY-MAX works better, as it ensures that the state shown is as accurate as possible for the peer with the worst connection.

Changes in Delay:

When the delay of a network changes, we only know once our peers give us a new set of connection information. This comes with the unfortunate side effect that the accuracy of our delay information is also tied in some amount to the delay of our peers. If delay is mostly constant, it does not matter that new information has not reached us, as this previous information still reflects the current state of the connection. If this information were to no longer reflect the state of our connection, that becomes a problem. When the delay of a network changes, we will not receive that information for a length of time equal to the delay of our peer in said connection. Afterwards, how long it takes for that information to affect our effective delay depends on the method used:

- (Delay-MAX & New delay is higher): New delay information is maximum delay in buffer => will be applied immediately as maximum delay for connection. **Instant**
- (Delay-MAX & New delay is lower): New delay information is lowest delay in buffer => will be applied only after old delay data is out of the delay list for connection. **Depends on length of delay list/buffer.**
- Delay-AVG: Smooth transition from old to new delay as delay takes up more and more of connection delay list. **Depends on length of delay list/buffer.**

By these metrics, and in testing, Delay-MAX is best at reacting to changes in delay when it gets higher, but does so jarringly, and takes longer than Delay-AVG when delay gets lower. Delay-AVG always takes some time to adjust, but provides a smoother transition between the old and new delay.

Jitter:

These differences are reflected best when the system is used with networks in which there is a notable amount of jitter (>20ms) with or without a constant amount of delay. In these networks, Delay-MAX will sometimes exhibit a “jumping” behaviour in which the effective delay quickly raises from the higher bounds of the connection’s delay to its lowest, introducing a large loss in input delay for players. This large loss can lead to a short period of confusion, as the player might not have time to react to the improved conditions of the network. The chances of this occurring can be reduced by keeping a larger sample size of previous connection delay amounts, but doing so will not prevent this from occurring. Delay-AVG on the other hand does a much better job of working under networks with a large amount of jitter, assuming that jitter is distributed normally. By using the average of a set of samples of previous delay values, we get a solid estimate of the true average delay of a network with jitter. The problem occurs when multiple connections with large amounts of jitter exist, as under these circumstances Delay-AVG only gives a solid estimate if the average delays of these connections are similar, but Delay-MAX can no longer provide solid performance as backup when the averages are far apart.

Spikes/Dropped Packets:

In connections where packets are dropped, “spikes” in delay can occur in packets facilitating the existence of delay data, causing estimated delay to exceed the true delay of a connection. Detecting when received delay data has been affected by being dropped and re-sent is important, as we do not wish to compensate for what is not actual delay. If we do not add some sort of filtering mechanism for dropped packets, a packet that has had to be re-sent can easily show double of what a network’s true delay is. In networks with smaller amounts of delay (<25ms) this may not affect our system much, but networks where there are already high amounts of delay and/or jitter present can lead to massive spikes in effective delay, and the game seeming to “freeze” for a period of time. This effect is much more pronounced with DELAY-MAX, but is also apparent with DELAY-AVG. One can set a threshold or maximum effective delay to prevent these “freezes”, but it is much more useful to try and filter out re-sent delay information packets instead. This can be done with a dynamic threshold (ex. disregard any packets with greater than double the most previous delay packet’s delay) or by some other filtering system (ex. For Avg/Max calculations of delay buffers, don’t use the maximum element).

High Effective Delay and Player Experience:

In situations where effective delay is large (>100ms), whether or not we are getting an accurate state of our player, the quality of player experience may be impeded by the introduced input delay. In small amounts, this delay may be able to be anticipated and played around by players, but if these amounts get too large, users may start to get annoyed by the introduced lag of their inputs and loss of responsiveness. The exact amount that will cause this annoyance will differ from player to player, but lower will almost always be preferred. This is one of the largest drawbacks of delay-based netcode, as for players who are newer to a game and have not learned to play with delayed inputs or less accepting of input delay, it can heavily affect their enjoyment.

Overall:

Delay-based netcode works best in scenarios when all peers have connections with similar levels of delay and jitter (mean delay +/-30ms). Situations with high variance of network qualities/delay makes delay-based netcode inaccurate for showing the true state of other peers in a network. If packet loss is not accounted for and filtered out, it can lead to large spikes in input delay that greatly affect player experience. Though delay-based netcode is still accurate in networks with higher delay, it may cause problems if players are not used to playing with a larger amount of input delay. DELAY-AVG is the best of the two systems for effective delay calculation, with the exception of networks in which the gaps in network delay for different connections is large.

Implementation and Computational Performance

The implementation of delay-based netcode only requires the following to be added to the target application: a system for storing delay data given by peers, a system for responding to and routinely sending packets to establish delay of connections, and a system for buffering inputs. A system to store delay data can be as simple as a set of arrays, one for each peer, that stores delay history for each. Keeping a history and using more than one sample of delay for a connection’s true delay value helps avoid spiking effective delay values and gives a more accurate representation of the delay of a connection. The “ping/pong” packet system described earlier is a simple way to obtain delay data, but requires that the global timestamps between clients can be trusted to be equal. One could also estimate delay based on RTT, but doing so would be much less accurate. Support from a server could help coordinate the sending of packets to ensure timestamps are in synchronised. Implementing a system that buffers inputs can be done with a min-priority queue and a system that allows for discrete player actions to be timestamped, stored, and executed at a later time. With that information stored, one can compare the current time minus the current delay to the timestamp of the head object of the queue to know if it should be executed or not. The system performs well, with the only large performance hits coming from when delay quickly lowers and many inputs from the priority queue have to be executed in the same frame. Compared to rollback netcode, the ease of implementation and performance of delay-based netcode are a large strength.

Rollback Netcode

Introduction to Rollback Netcode

Rollback is the second style of netcode we studied in this project; it takes a different approach compared to Delay-based netcode that responds better to variable network performance. It works by updating the game instantly from local players inputs and then, it will retroactively update the game state with remote players input when it arrives, this creates the illusion of a seamless offline experience even with some network delay. This strategy better simulates offline play which makes it more popular for fighting games and platformers that require precise timing.

Core Mechanism

Rollback's main mechanism functions by immediately applying the local user's input and showing this result right away, while the remote player's inputs are predicted so they can also be shown instantly. When the remote players inputs are sent to the local player the game checks to see if what it predicted was correct based on what the remote players inputs are, if they are the same then there is no issue and the game continues. However, if the remote player's inputs are different then predicted the game needs to fix its incorrect predictions, it does this by rolling back to a previous state when the inputs were made and re-simulates the changed inputs several frames forward to the current frame giving an updated accurate current state which is then shown to the local player. When a rollback happens there can sometimes be a small jump or “glitch” in what the player sees when the current state that was based on the prediction changes to the correct state based on the remote players input. An example of how this might look is the remote player was moving forward and it is predicted that they will continue to move forward but their input arrives and they actually stopped moving, this would result in them traveling back to where they were when they stopped pressing the button resulting in them visually jumping back a little bit.

Prediction Strategies

Having accurate predictions of what remote players are doing is important to avoiding unnecessary rollbacks, there are a few different strategies with some being better than others for some types of games. The easiest is to assume your opponent will always remain stationary, but this isnt always the best assumption especially in games where players are constantly moving as this will cause many rollbacks and lots of visual jumping for players. Another prediction method is to predict players inputs will be what they input most recently, since this is a perframe prediction it ends up being right upwards of 90% of the time since players don't often change inputs more than a couple times a second [3]. This prediction strategy works very well for games that have a relatively low number of possible inputs and where players maintain actions for a period of time letting the predictions be accurate for the majority of the time.

Rollback Conditions

Rollbacks happen when the game makes predictions about other players' inputs, usually assuming they will keep doing what they were doing last. When remote players' inputs arrive the games checks if they match the predictions if they don't match then the game rolls back to the frame where the wrong predictions started, applies the correct input, and re-simulates all the frames up to the present to stay synchronized. This whole process happens quickly often without players noticing, However when network latency is high predictions can't be corrected as quickly causing more noticeable rollbacks and visual 'glitches' since rollbacks need to rewind further into the past to update the game and the changes make are large since the wrong prediction has been allowed to continue uncorrected for longer. When network delay is smaller, predictions are either correct or the rollbacks that need to happen are at most a couple frames and are largely unnoticeable.

Performance of System

The rollback-based system was tested under varying network conditions including constant delay, changing delay, and jitter. As with the delay-based system, we used a qualitative approach, recording gameplay under each condition and comparing what each player saw. With the goal of finding what network conditions rollback performs best in as well as which it struggles with.

Constant Delay:

Rollback netcode performs well under constant delay, even when the delay is fairly high. This is because local inputs are applied immediately, and remote inputs are predicted so they happen in real time regardless of the network delay. The main case where the delay is still noticeable is when the remote player changes their movement, which causes a small jump in the player's position when the remote input arrives. But because of the prediction strategy being the last known input the majority of frames don't trigger a rollback. When inputs from peers did not change often (e.g. moving in one direction), there is no visible effect of delay; it is only when input changes that rollbacks effects can be noticed. Rollback produces a fairly smooth gameplay experience with minor visual glitches but no input delay

Changes in Delay:

When network delay increased or decreased mid-game rollback performed better in terms of responsiveness of the local player since local inputs are applied immediately. Since rollback doest rely on past delay information it doesn't suffer in the same way a delay based approach does. However one downside is that if delay fluctuates, it increases the chances that a prediction will be wrong resulting in more frequent rollbacks. These rollbacks can cause more frequent visual stuttering as past frames are re-stimulated more often, these glitches generally affect gameplay less than rapid changes in input delay.

Jitter:

In networks with significant jitter (>20ms variance), rollback outperforms delay-based approaches in maintaining input responsiveness. Since the system never waits to apply local input, the player's actions are immediate even under poor network conditions. However jitter increases the number of incorrect predictions especially when remote players change their input often. This can cause remote players' characters to visually jitter or jump around as rollbacks re-simulate frequently. Our implementation always shows the local players input immediately and they don't get affected by network delay while remote players input tend to jump around a bit more as a result of frequent rollbacks.

Spikes/Dropped Packets:

In rollback-based systems, packet drops don't delay local input but may cause a delayed correction once the missing input arrives or times out. This would look like a sudden large jump as the predicted input is corrected to the actual delayed input, but the jump would be larger since the delay would be increased up to double the normal amount. To mitigate this the system could be configured with rollback limits, such as only rolling back a fixed number of frames or ignoring inputs that arrive too late. We also explored keeping a limited buffer by dropping old states after a certain window, this helps reduce CPU load and memory usage at the cost of a reduced rollback window. In a high packet loss environment rollback still offers a responsive experience but with a higher amount of remote jittering and more noticeable state corrections.

High Effective Delay and Player Experience:

Unlike delay-based netcode, rollback maintains a low-latency feel even in high-latency scenarios, since it never delays local input. This is often preferred by players particularly in fast paced games or games where timings are very specific such as fighting games. The trade off is that visual consistency degrades as latency increases especially if opponents change their inputs frequently, this can be mitigated with visual smoothing and other effects but is still noticeable. Players generally prefer visual inconsistencies over delayed input usually making rollback appealing to players, rollback also avoids freezing the game even when input delays fluctuate making it more resilient to fluctuating network conditions.

Overall:

Rollback netcode performs best in situations where responsiveness is more important than visual precision particularly in real-time, reaction heavy games. It offers smooth player control even when network with high delay, jitter or packet loss. The main drawbacks lie in its complexity of implementation and inconsistent visual effects caused by frequent corrections especially when delay is high causing many rollbacks and visual stutter. While rollback generally performs well on all network conditions it is not as smooth as delay-based when internet delay is low, but offers a more responsive experience compared to delay-based the worse the network conditions get.

Implementation and Computational Performance

To properly implement Rollback netcode there are several considerations that need to be addressed. The main thing is needing to store past game states to allow for rollbacks to happen by rewinding and re-applying inputs when delayed remote inputs arrive. These state snapshots typically include the position, velocity, animation state, and any other relevant information for all entities in the game world. Adding rollback introduces an additional computational burden, the game must not only process the current frame but also occasionally re-simulate several previous frames in quick succession to catch up after a correction. This extra computation also needs to happen in less than one in-game frame, which is often less than 16 milliseconds if the game is at 60 frames per second (fps).

For rollback to even work for a specific game, the game needs to be deterministic. This means that given the same inputs and initial state, the game must always produce the same output. Non-deterministic behaviors, such as physics inconsistencies, random number generation without synchronization, or floating-point inaccuracies across platforms, can lead to permanent desynchronization between players. As such, all game logic involved in rollback must be written with determinism and replayability in mind. Additional considerations include:

- object creation and destruction must be reversible,
- collisions and physics must play out the same for both real and re-simulated frames,
- ensuring players are inputting for the same frame at the same time.

Our implementation of rollback was informed by the design of a rollback platformer project, which helped clarify some of the concepts needed to allow rollback to work effectively [4]. Rollback netcode adds complexity and performance overhead, but when implemented correctly, it enables a responsive and robust multiplayer experience, even under less than ideal network conditions.

Comparison of Netcode Types

Delay-based netcode primarily works by delaying local input until remote input arrives and executing them together on the next frame. Rollback on the other hand applies inputs instantly and predicts remote inputs and corrects prediction when actual inputs arrive if needed. The choice of which netcode type to use and when depends on a variety of factors including development time and resources, the type of game, and the network conditions the game is likely to experience.

Delay-based pros/cons

Pros:

- Simpler implementation: less architectural overhead such as not needing to store past states, and not needing to worry about re-simulating past frames.
- Predictable performance: under stable network conditions the visuals are smoother and more consistent than with rollback.
- Low computational overhead: it requires minimal overhead to just compute what delay to use and check time stamps, compared to high overhead of re-computing past frames.

Cons:

- Input delay: players inputs are delayed to allow for synchronisation, reducing responsiveness.
- Poor Handling of variable latency or jitter: since delay takes some time to accurately predict rapid changes in network delay cause poor performance and extra delay.
- Hard to predict delay with multiple peers: since the delay will either be an average or the maximum delay, neither of which is the best for all players.

Rollback pros/cons

Pros:

- Highly responsive inputs: local inputs are applied immediately making gameplay smooth and reactive, which is closer to an offline experience.
- More resilient to latency and jitter: still player even with high delay or unstable network conditions.
- Better player experience for peer-to-peer games: when there's no central server to standardize delay, rollback can be more consistent.

Cons:

- Much more complex implementation: requires deterministic game logic, state saving, and re-simulation.
- Visual glitches: rollbacks can cause snapping or jittering when corrections are applied.
- High CPU and memory usage: must store many past states and re-simulate past results all in less than an in game frame.
- Not suited for all games: rollback doesn't work well for state heavy or no-deterministic games, such as simulation or strategy games with lots of state information.

Which is Better?

This ultimately depends on the type of game and the resources available for development. Delay-based will likely be sufficient for slower or turn-based games, or those with less resources to develop complex netcode systems. Rollback is preferred for fast real-time games where response time is crucial, peer-to-peer games are also better suited for rollback as it provides a better experience given no central authority. Despite this, development resources or constraints often cause developers to choose delay-based instead due to rollbacks complexity, however libraries such as the GGPO SDK aim to make the use of rollback netcode an easier decision.

Hybrid Solutions

Given these tradeoffs, and the differences in how these methods help reduce the effects of desynchronisation, a system in which both are used simultaneously or switched between may be able to provide the best of both worlds in terms of overall benefit. Due to time constraints, we were not able to test these ideas in full, but they are very important to mention as they are used in some modern games as a “cutting-edge” netcode solution.

Swapping Method

Since delay-based netcode is stable at lower delays, and lacks any risks of needing to rollback the gamestate, a system can be used in which delay-based netcode is used when effective delay is low, and rollback netcode can be used once delay-based netcode is deemed unable to handle the current level of delay without affecting the user experience. This threshold could in theory be a static number (ex. 60ms), but doing so runs the risk of the network constantly jittering above and below it, causing a jump between 60ms of input delay and 0ms with rollback. Instead, one can use a dynamic threshold based on previous delay data to avoid this problem and be able to provide solid performance in a variety of networks. One could even allow users to set their own thresholds, allowing them to decide on a more personal level what point they wish for the switch to occur. This makes the system personalizable, but might be daunting for users who do not understand what rollback or delay-based netcode entails.

True Hybrid Method

But what if instead of swapping between both methods, both methods were used at once. Delay-based netcode to help give the local player a better idea of their state on remote peers and rollback netcode to help provide predicted information on remote player game states. Having to run both systems at once can provide, in theory, the best performance of any option mentioned prior. But under the wrong network conditions, one can risk introducing the worst parts of both systems simultaneously; not only introducing noticeable input-lag, but also frequent rolling-back of the game. The performance of this system hinges greatly on the implementation of both of these systems, better implementations (reliable effective delay calculation and state prediction) will lead to better performance in worse network conditions, which is crucial given poor network conditions will cause both systems to struggle simultaneously.

Conclusion

In this report we have discussed the problem faced by P2P networked online games, what netcode is, and how rollback and delay-based netcode can be implemented to help reduce these effects. We went over how these systems work, what problems they address, the subsystems that make them up and the pros and cons of these systems. Delay-based netcode provides an easy way for developers to give players a better idea of how they look to their opponents, but it falters at high delay amounts. Whereas rollback netcode is a tougher to implement system that has the aim of providing players with accurate information of their opponents that is able to hold up at higher delays. Along with this, hybrid methods in which both systems are used in some capacity were discussed. We found that while rollback based systems tend to hold up better in worse network conditions, the ease of implementation of delay-based systems makes it much more generalizable to games with more complex states. We hope the findings of this paper can help developers make better decisions about what systems are right for their game in a P2P setting and how said solutions can help improve player experiences.

Individual Group Member Contributions

Seth Messier	**Dario Monette**
Project: ● Course Website ● Game Code ○ Player + Projectile Logic ○ Rendering ● Rollback Netcode ● Implementing online multiplayer in the game with the WebRTC system ● Hosting/Deploying the game+server to an external server Report: ● Rollback Netcode ● Comparison of Netcode Systems ● Reviewed and edited other sections	Project ● Signaling/Intermediate Server Programming ● WebRTC connection system + broadcasting system ● Delay-based netcode implementation ● Artificial Delay Tool ● Lobby System and Website Report: ● Introduction ● Problem Definitions ● Application Overview ● Delay-Based Netcode’ ● Hybrid Solutions ● Conclusion ● Reviewed and edited other sections

References

[1] S. Shenker, "Fundamental Design Issues for the Future Internet". IEEE Journal on Selected Areas in Communications, Vol. 13, No. 7, September 1995, p p. 1176-1188.

[2] Chris Courses, “Online Multiplayer JavaScript Game Tutorial - Full Course,” *YouTube*, Aug. 07, 2023. https://www.youtube.com/watch?v=HXquxWtE5vA.

[3] R. Pusch, “Explaining how fighting games use delay-based and rollback netcode,” *Ars Technica*, Oct. 18, 2019. https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/

[4] “Making a GGPO-style rollback networking multiplayer game,” Outof.fish, Jan. 14, 2023. https://www.outof.fish/posts/rollback/