

MULTIPLAYER → E-BOOK

// HOW TO DEAL WITH LATENCY IN YOUR MULTIPLAYER GAME


Tricks and patterns to understand and reduce network latency

INTRO_

Everyone hates laggy gameplay when playing online, but it shouldn't be seen as an unavoidable part of multiplayer gaming. There is actually quite a lot that you can do to help your players have the smoothest experience possible.

However, between distance between server and clients, packet hops, a server's **tick and update rate**, and a plethora of other issues, developers can get buried in technical issues and your players may suddenly find out they were clocked in the head by something they thought was half the map away.

While it is never easy to put together a multiplayer game with a strong and secure online experience, it is absolutely doable and this guide on network latency management will help you lock down the fundamentals.



WHAT IS NETWORK LATENCY?

Online games have issues that single player or LAN-only games don't have to worry about, such as jitter, round-trip time (RTT), or packet loss.

Any kind of delay between sending, receiving, and queuing information between clients and servers can cause an issue for gameplay. For a full list of definitions and problems, check out [this guide here](#).

To successfully address latency problems, you need to consider the priority and relationship between the following elements:

1. Security
2. Reactivity
3. Accuracy and consistency

No solution is perfect, and every way to approach latency issues has strengths and weaknesses. The key is to find the way that works best for your game and this guide will help you understand how to make that decision.

CONTENTS_

Security and authority	5
Authority models and latency	6
Solving latency issues in server authoritative games	11
Conclusion	15
Glossary of terms	16

SECURITY AND AUTHORITY

Authority defines who has the right to make final gameplay decisions over objects in the client-server relationship. The authority model you select has implications for how you manage network latency in your game.

There are two authority types in multiplayer games, and each has its own relationship with security, responsiveness, and accuracy and consistency.

SUMMARY	
Server authority	More secure. Less reactive. No sync issues.
Client authority	Less secure. More reactive. Possible sync issues.

Any code that exists on the client's side can be tampered with, and players can forge false network messages sent to the server.

If you want to know how this might end up looking in your game, consider the example of a game where you cannot kill imps from a certain distance.

If you specify in your client logic that you can't kill these imp more than 10 meters away, but the "kill imp" message is a server [RPC](#) (remote procedure call) that doesn't check distance server-side, players can forge that network message to bypass your client side logic.

Unfortunately, some people will always try to mess with your game so you should always keep in mind you can never fully trust clients. For the sake of those poor imps and all others playing your game, your server needs logic to validate player actions coming from clients.



AUTHORITY MODELS AND LATENCY

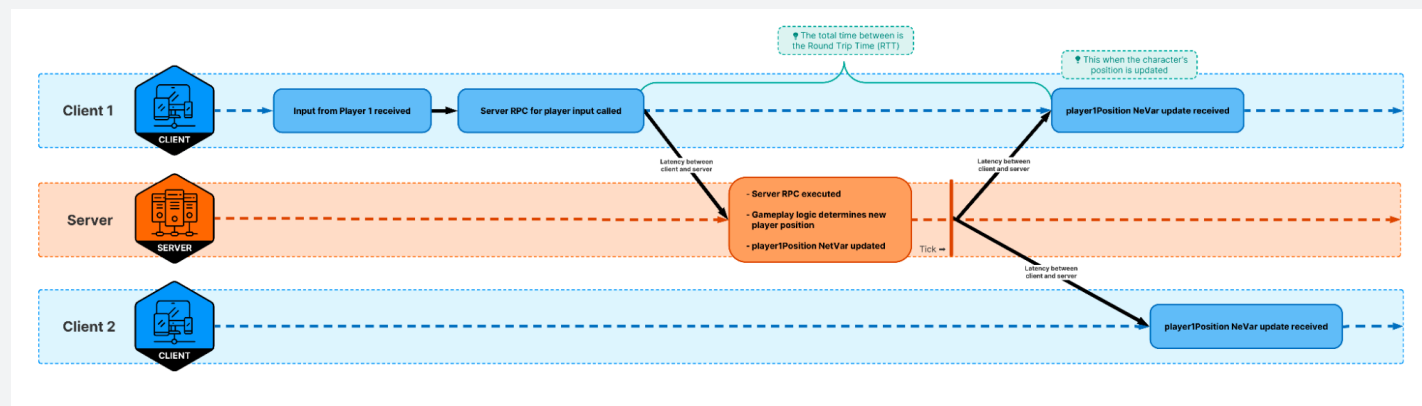
As mentioned above, the authority model you select has implications for the network latency in your game. Let's review the two authority types and their relationship with security, reactivity, and consistency.

SERVER AUTHORITATIVE GAMES

A server authoritative game has all its final gameplay decisions executed by the server.

EXAMPLE

Character position with server authority



In this example, the server gets to make the final gameplay decisions.

✓ 1. GOOD FOR SECURITY

In a server authoritative model, critical data such as your character health or position can be server authoritative to ensure that cheaters can't mess with it. In that instance, the server will have the final say on that data's value. You don't want players being able to set their own health – or, even worse – other player's health at will.

[Netcode for GameObjects](#) is server authoritative, which means all writes to `NetworkVariables` are only allowed from the server. However, when accepting RPCs coming from clients, you need to make sure to add validation code, since those RPCs are coming from never to be trusted sources.

✓ 2. GOOD FOR CONSISTENCY

An advantage of server authoritative games is your world's consistency. Since all gameplay decisions (such as a player opening a door or a bot shooting a player) are made by the same node on the network (the server), you can be sure these decisions are made at the same time.

A client authoritative game would have decisions made on client A and other decisions on client B, with both being separated by [RTT](#) ms of internet lag. Player A killing player B while player B was already hiding behind cover would cause consistency issues.

Having all this gameplay logic on one single node (the server) makes these kinds of considerations irrelevant, since everything happens in the same execution context.

✗ 3. NOT GOOD FOR REACTIVITY

An issue with server authority is that you end up waiting for your server to tell you to update your world. This means that if you send an input to the server and wait for the server to tell you your position change, you'll need to wait for a full RTT before you see the effect.

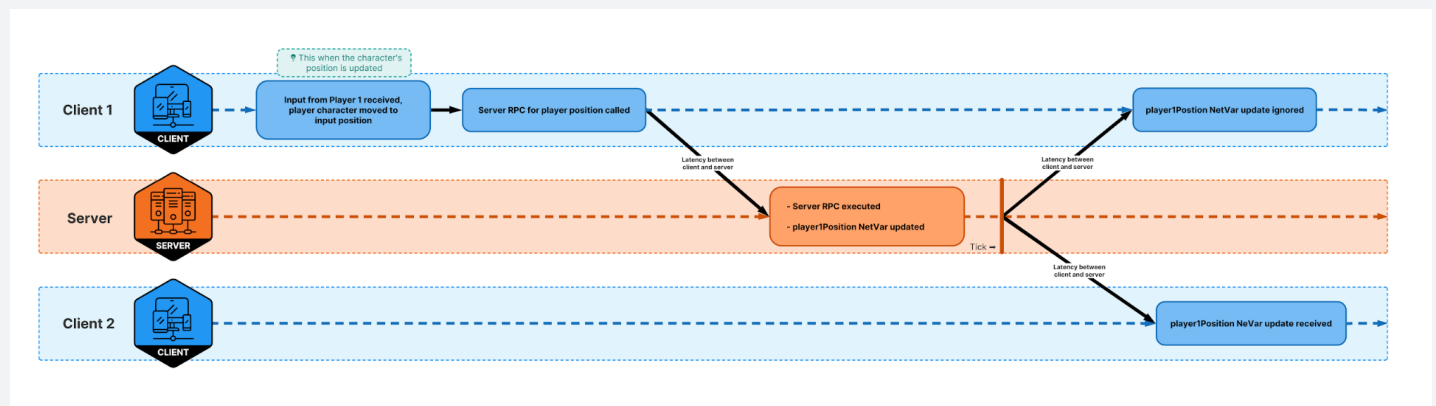
However, stay tuned as there are patterns you can use to solve this issue while still remaining server authoritative that will be covered further down.

CLIENT AUTHORITATIVE GAMES

In a client authoritative (or client driven) game, you still have a server that's used to share world state, but clients will own and impose their own reality.

EXAMPLE

Character position with client authority



In this example, the client gets to make the final gameplay decisions.

✓ 1. GOOD FOR REACTIVITY

A client authoritative model can often be used when you trust your users or their devices, and it's a useful model for reactivity. Since the client itself is making all the important gameplay decisions, it can display the result of user inputs as soon as they happen instead of waiting a few hundred milliseconds.

For example, you could have a client tell the server "I killed player X" and have the server simulate that action to return the result. This way, your client could show the death animation for your enemy as soon as you clicked, since the death would already be confirmed and owned by your client. The server would only relay that information back to other users.

❌ 2. NOT GOOD FOR CONSISTENCY

Client authoritative games present possible sync issues. If you let your client make an authoritative decision using outdated information, you'll run into **desyncs**, overlapping physics objects, and other such issues.

For example, if player movement is client side, different players may be in different versions of the world based on each client's processing and refresh speed.

That means player A may move freely on their client for a few frames behind Player B's client in which they have stunned Player A. These two game realities are out of sync and will result in Player A appearing to be suddenly stunned when Player A's client jumps back in sync with the other client's.

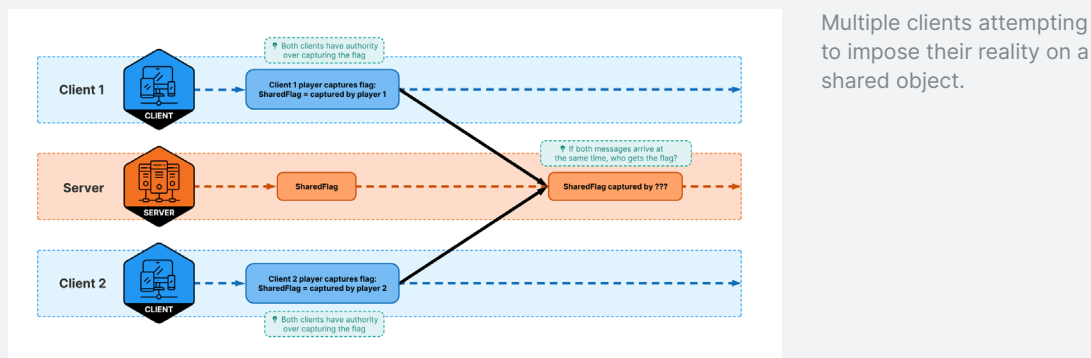
EXAMPLE

Owner authority versus all clients authority

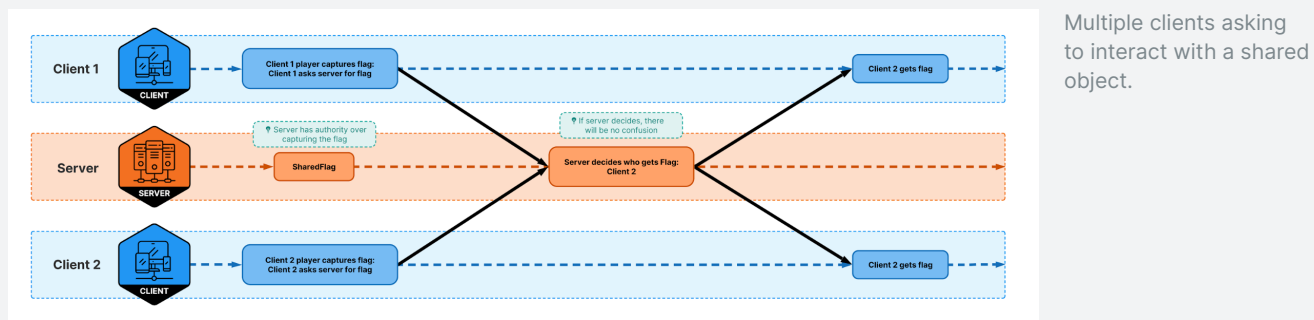
Giving multiple clients the ability to affect the same shared object can get messy.

To avoid this, it's recommended to use client owner authority, which would allow only the owner of an object to interact with it. Since ownership is controlled server side in Netcode, there's no possibility of two clients running into a race condition. To allow two clients to affect the same object, you'd need to ask the server for ownership, wait for it, then execute the client authoritative logic you want.

Capture the flag part one – client authority issue



Capture the flag part two – server authority fix



❌ 3. NOT GOOD FOR SECURITY

Client authority is a pretty dangerous door to leave open on your server, since any malicious player could forge messages to say “kill player X” and win the game.

When you don't think there's any reason for your players to cheat – such as it's a cooperative game – the client authority model can be a great way to have reactivity without some of the complexity added with techniques like input prediction.

Another way of solving this issue in a client authoritative game is using soft validation server side. Instead of completely doing a simulation server side, the server will only do basic validation. It could, for example, do range checks to make sure a player isn't teleporting to places it shouldn't. This would usually be acceptable in a **PvE** game. However, any **PvP** will usually require server authority.


ANOTHER OPTION

A pattern we've seen that helps when you're not sure about using client or server authority is to implement your game behavior not by server/client, but by authoritative/non-authoritative.

By abstracting this to authority instead of isServer/isClient, your code can easily be swapped to client or server authority without too many refactors.

```
// before
if (isServer)
{
    // take an authoritative decision
    // ...
}
if (isClient) // each of these need to be swapped if switching authority
{
    // ...
}

// after
bool HasAuthority => isServer; // can be set for your whole class or even project
// ...
if (HasAuthority)
{
    // take an authoritative decision
    // ...
}
if (!HasAuthority)
{
    // ...
}
```



SOLVING LATENCY ISSUES IN SERVER AUTHORITATIVE GAMES

In multiplayer game development, best practice is to adopt a server authoritative model for consistency and security of your gameplay. Client authoritative models open your game to cheating and more latency issues.

Below, we'll cover four key game design strategies to manage latency in server authoritative games:

SUMMARY

Client authority	Less secure. More reactive. Possible sync issues.
Action anticipation	More secure. Somewhat reactive. Possible visual sync issues.
Prediction	More secure. More reactive. Correction on sync issues. Complex and tentacular.
Server side rewind	More secure. More accurate by favoring the attacker. Shot behind a wall issue.

1. ALLOW LOW IMPACT CLIENT AUTHORITY

When designing your feature, use server authority as the default and then identify which feature needs reactivity and doesn't have a big impact on security or world consistency.

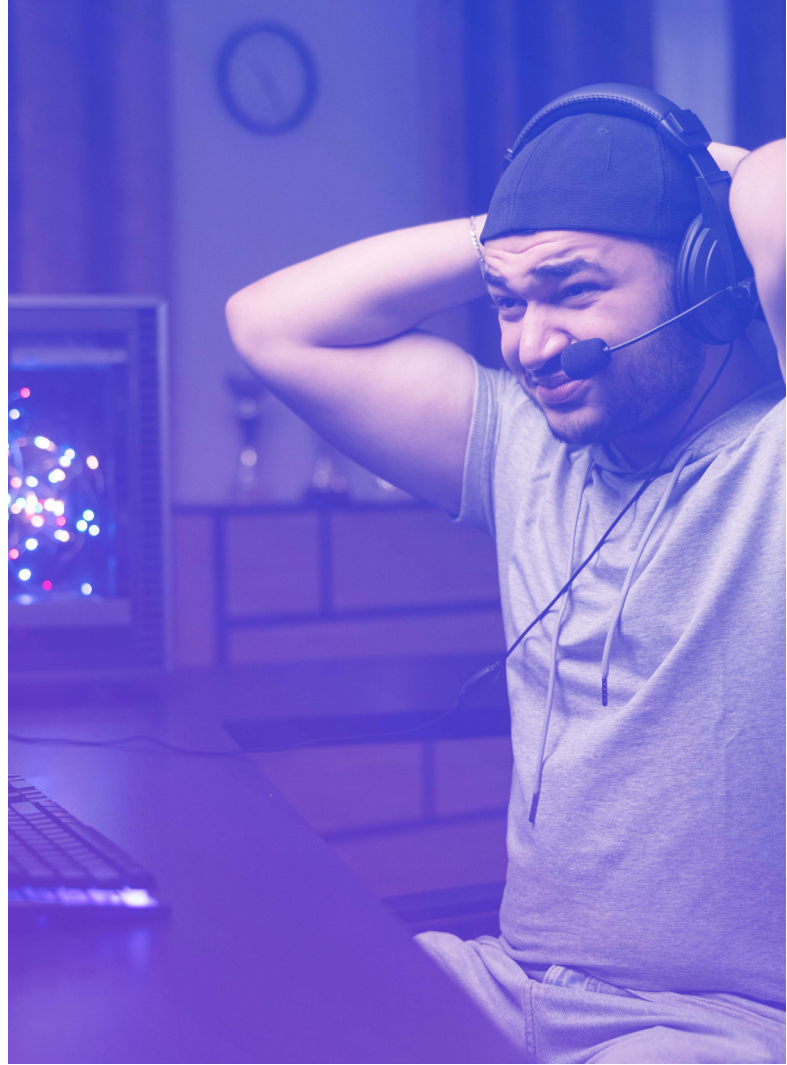
User inputs are a good example. Since users already own their inputs (I own my key press, the server can't tell me "no you haven't pressed the W key"), user input data can easily be client driven.

In FPS games, your look direction can easily be client driven without much impact. The client will send to the server its look direction instead of mouse movements. Having a correction on where you look would feel weird and security for this has its own challenges.



TIP

If the server can correct the action, use server authority.



2. CLIENT SIDE PREDICTION

Prediction is a very common way of making an educated guess as to what the server will send you. Your game can stay server authoritative, but instead of waiting a full RTT for your action results, your client can simulate and run gameplay code of what it thinks will happen as soon as your players trigger inputs.

For example, instead of waiting a full RTT for the server to tell me where I moved, I can directly update my movements according to my inputs. This is very close to client authority, except with this technique you can be corrected.

The world (and especially the internet) is messy. A client could guess wrong. An event produced by another player could come and mess with your own local guess or your physics simulation could be non-deterministic.

This is where "reconciliation" (or "correction") comes into play. The client keeps a history of the positions it predicted, and, being server authoritative, the client still receives (outdated by X ms of latency) positions coming from the server.

The client will validate whether the positions it predicted in the past fits with the old positions coming from the server. The client can then detect discrepancies and "correct" its position according to the server's authoritative position. This way, clients can stay server authoritative while still being reactive.

3. ACTION ANTICIPATION

There's multiple reasons for not having server authoritative gameplay code run both client side (with prediction) and server side.

For example, your simulation could be not deterministic enough to trust that the same action client side would happen on the same server side. If I throw a grenade client side, I want to make sure the grenade's trajectory is the same server side.

This often happens with world objects with a longer life duration, with greater chances of desyncing. In this case, the safest approach (well, safe for a grenade) would be a server authoritative grenade, to make sure everyone has the same trajectory.

But how do you make sure the throw feels responsive and that your client doesn't have to wait for a full RTT before seeing anything react to their input?

For a lot of games, when triggering an action, you'll see an animation/VFX/sound trigger before the action is actually executed. A trick often used for lag hiding is to trigger a non-gameplay impacting animation/sound/VFX on player input (immediately), but still wait for the server authoritative gameplay elements to drive the rest of the action.

If the server has a different state (your action was canceled server side for some reason), the worst that happens client side is that you've played a quick but useless animation and it's easy to just let the animation finish or cancel it. This is referenced as action casting or action anticipation. You're "casting" your action client side while waiting for the server to send the gameplay information you need.

For your grenade, a client side "arm throw" animation could run, but the client would wait for the grenade to be spawned by the server. With normal latencies, this usually feels responsive. With higher abnormal latencies, you could run into the arm animating and no grenade appearing yet, but it would still feel responsive to users. It might feel strange, but at least it would feel responsive and less frustrating.

This is also useful for any action that needs to interact with the world. An ability that makes you invulnerable would need to be on the same time as other server events. If I predict my invulnerability, but a sniper headshots me before my input has reached the server, I'll see my invulnerable animation, but will still get killed. This is obviously pretty frustrating for users.

Instead, I could play a "becoming invulnerable" animation with the character playing an animation, wait for the server to tell me "you're invulnerable now" and then display my invulnerable status. This way, if a sniper shoots me, the client will receive both the sniper shot and the invulnerability messages on the same timeline, without any desync.

4. SERVER SIDE REWIND (AKA LAG COMPENSATION)

Server rewind is a security check on a client driven feature to make sure we stay server authoritative.

A common use case is snipers. If I aim at an enemy, I'm actually aiming at a ghost representation of that enemy which is $RTT/2$ ms late. If I click its head, the input sent to the server will take another $RTT/2$ ms to get to the server. That's a full RTT to miss my shot and is very frustrating.

The solution for this is to use server rewind by "favoring the attacker". This is because it's way more frustrating for an attacker to always miss their shots than for a target to get shot behind a wall once in a while.

The client sends along with its input a message telling the server "I have hit my target at time T ". The server, when receiving this at time $T+RTT/2$, will rewind its simulation at time $T-RTT$, validate the shot and correct the world at the latest time (i.e. kill the target). This allows for the player to feel like the world is consistent (my shots are hitting what they are supposed to hit) while still remaining secure and server authoritative.

Note: The server rewind of the game's state is done all in the same frame – this is invisible to players. This is a server side check that allows validating a client telling you what to do.



CONCLUSION

Building a multiplayer game is a challenging endeavor, but also an exciting one. Whether you're building the next battle royale smash hit, or a cozy online co-op, understanding the nuances of latency and how to manage it is essential.

Check out Unity's Netcode for GameObjects solution and documentation to get started with your next multiplayer project today.

[EXPLORE NETCODE](#)

GLOSSARY OF TERMS

TERM	DEFINITION
Network latency	<p>The amount of time between a cause and its visible effect when sending information over a network.</p> <p>Example: The time between you pressing the button to shoot and your in-game gun firing.</p>
Remote procedure call (RPC)	<p>RPCs are ways to call methods on objects that are not in the same executable or hosted on the same machine.</p> <p>Example: A client can invoke a server RPC on a NetworkObject. The RPC will be placed in the local queue and then sent to the server, where it will be executed on the server version of the same NetworkObject.</p>
Round-trip time (RTT)	<p>RTT is the time it takes for a packet to be transmitted from one machine to another and back.</p>
Player vs environment (PvE)	<p>A game type where players compete against the environment or situation, and not directly against other players.</p>
Player vs player (PvP)	<p>A multiplayer game type where players compete against other players directly.</p>
Desync	<p>When the client and server are not synchronized. When this happens, player movements and actions are not represented on the game server so the game state is not accurate for the player.</p>



unity.com/gaming-services