JUL 28, 2020

PEEKING INTO VALORANT'S NETCODE

Hello everyone!

We're Matt deWet, gameplay tech lead on *VALORANT*, and David Straily, project tech lead on *VALORANT* - and we're beyond excited to be here with you all to share some of the technical details behind how we're addressing some common issues in the FPS genre - peeker's advantage, poor hit registration, and simulation divergence.

From the beginning of *VALORANT's* development, we've had one guiding priority for all the decisions we make - what do players want, and how do we provide that in a way that feels satisfying?

First person shooters are what we call a "red ocean" opportunity space. There are dozens of phenomenal games in the genre (many we grew up playing) - any shooter offering made by Riot needed to really differentiate from the others to live up to our expectations. We've made many big bets with *VALORANT* to accomplish this, and one of our biggest is our ongoing investment in competitive integrity.

We take competitive integrity to mean that the outcome of each match is decided only by the planning and execution of the players in the game. The internet, game client performance, Riot's servers, and other players' machines should never get in the way of providing players with a level playing field in each match. Playing *VALORANT* should feel like a natural extension of a player's ability and skill.

This article will first explore the design goals we had in mind when building *VALORANT*'s netcode, and then dive into technical challenges and how we addressed these to align with our design goals. If you'd like to read about other technical topics, please take a gander at our articles on anti-cheat and shader implementations.

A FEW VALORANT DESIGN GOALS

As with many challenges in software development, building a competitive tactical shooter involves making tradeoffs across many (often competing) design goals. Let's briefly cover some of our goals that relate to netcode and competitive integrity, and afterwards we'll take a look at some of the technical challenges we had, and how we resolved them in ways that amplify these goals.

Games are fair.

We want any player who chooses to spend their time mastering *VALORANT* to see their investment pay off. This means preventing cheaters from ruining games and minimizing unfair advantages for players with better hardware or networks.

The challenge: Design client/server communication to limit the attack surface area for cheaters.

Note: *VALORANT* uses a server-authoritative networking model to limit the types of cheats possible. Therefore, our client/server communication is designed around the idea that a server must never trust a client's view of the world.

Movement is smooth and highly responsive.

Smooth, predictable movement is essential for players to be able to find and track enemies in combat. The server is the authority on how everyone moves, but we can't just send your inputs to the server and wait around for it to tell you where you ended up. Instead, your client is always locally predicting the results of your inputs and showing you the likely outcome.

The challenge: Predicting your movement on the client allows for crisp, low-latency response to local input, but it can cause trouble if your local simulation and the server simulation ever disagree. If the server ever disagrees with a client's view, it issues a correction to get the client back into sync. This can cause that player's camera or other players' characters to jump around or rubber-band erratically, making them hard to track. Our goal is to prevent these sorts of disagreements entirely under reasonable networking conditions, and to minimize their impact when they're unavoidable due to network loss or burst latency.

Gunplay feels rewarding.

Building a weapons system that's approachable while allowing for deep mastery is just the first step toward making gunplay feel rewarding. The simulation and networking layer here need to ensure that the results of each gunfight are easy to understand (clear visual feedback) and feel fair to all participants.

The challenge: The server needs to respect all of those clutch shots that you land, and make it clear where your missed shots actually landed.

Give players holding territory an advantage.

A core tenet in the tactical shooter genre is the idea that a player holding territory should, on average, have an advantage over a player taking territory. This is referred to as *holder's advantage*, and is critical for incentivizing some of the key gameplay elements of a tactical shooter: information gathering and denial, coordinated squad movement, methodical clearing of angles, and the use of abilities to gain an upper hand.

The challenge: It takes time for the information that an attacker peeked around a corner to travel over the internet and reach the defending player. By default, this gives the attacker more time to spot their opponent and line up a shot. This is a challenge that any online competitive shooter faces, and is commonly referred to as *peeker's advantage*, which is in direct conflict with our goal of holder's advantage.

Support a wide range of setups.

We want players to be able to compete in *VALORANT* even when their networks or computers leave something to be desired.

The challenge: Provide the best possible experience to each player, given their network & hardware setup. Just as important - we want to prevent network or performance problems that one player

encounters from impacting the experiences of the nine other players in their game. When you're playing *VALORANT*, it should feel like every other player is right next to you, playing on a high-end gaming machine.

TECHNICAL DETAILS

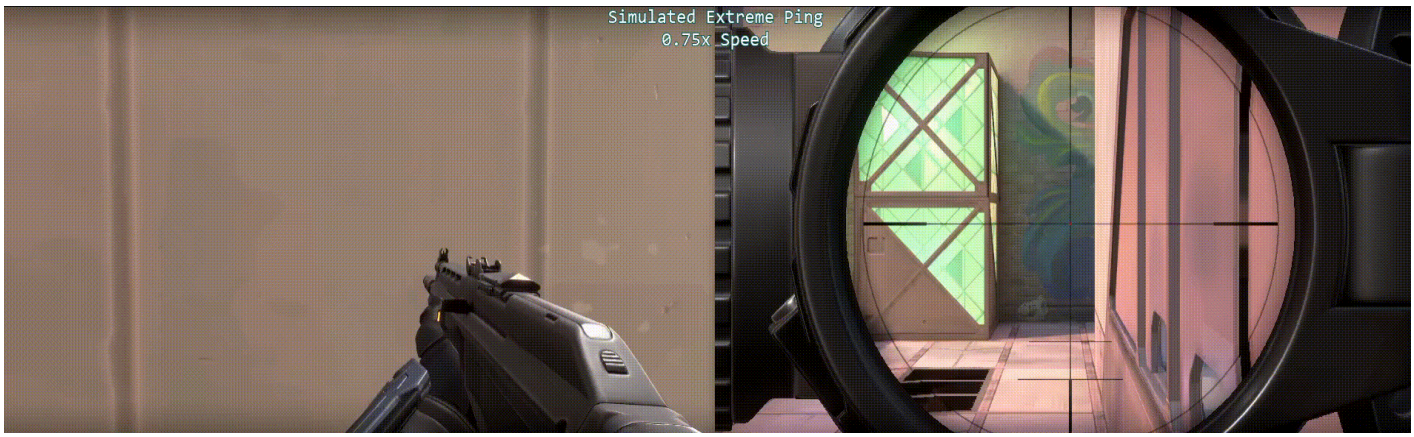
Let's now discuss some of the ways that we approached solving the challenges listed in the sections above.

We'll cover the following four problem spaces: mitigating peeker's advantage, minimizing simulation divergence, resolving combat on the server, and addressing client-side issues.

MITIGATING PEEKER'S ADVANTAGE

Peeker's advantage is an artifact of networked gameplay that's often the focal point of discussions around competitive integrity in tactical shooters. It refers to an advantage that someone peeking around a corner has over an opponent on the other side.

Let's take a look at a clip of peeker's advantage in action (exaggerated for clarity):



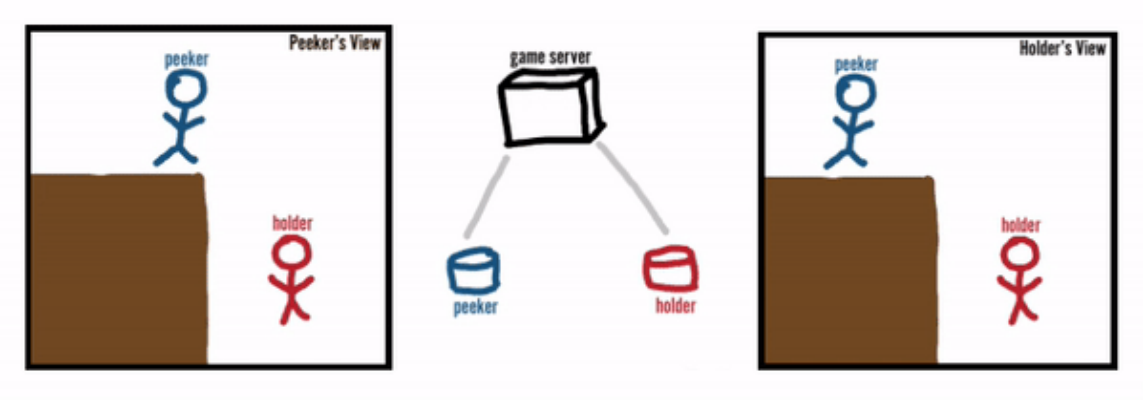
If you look closely at the above clip, you'll see that there are a few frames of time where the player on the left has rounded the corner and spotted their opponent, and their opponent still has no idea that they exist! This network desync is bad for gameplay, and directly counters our design goals.

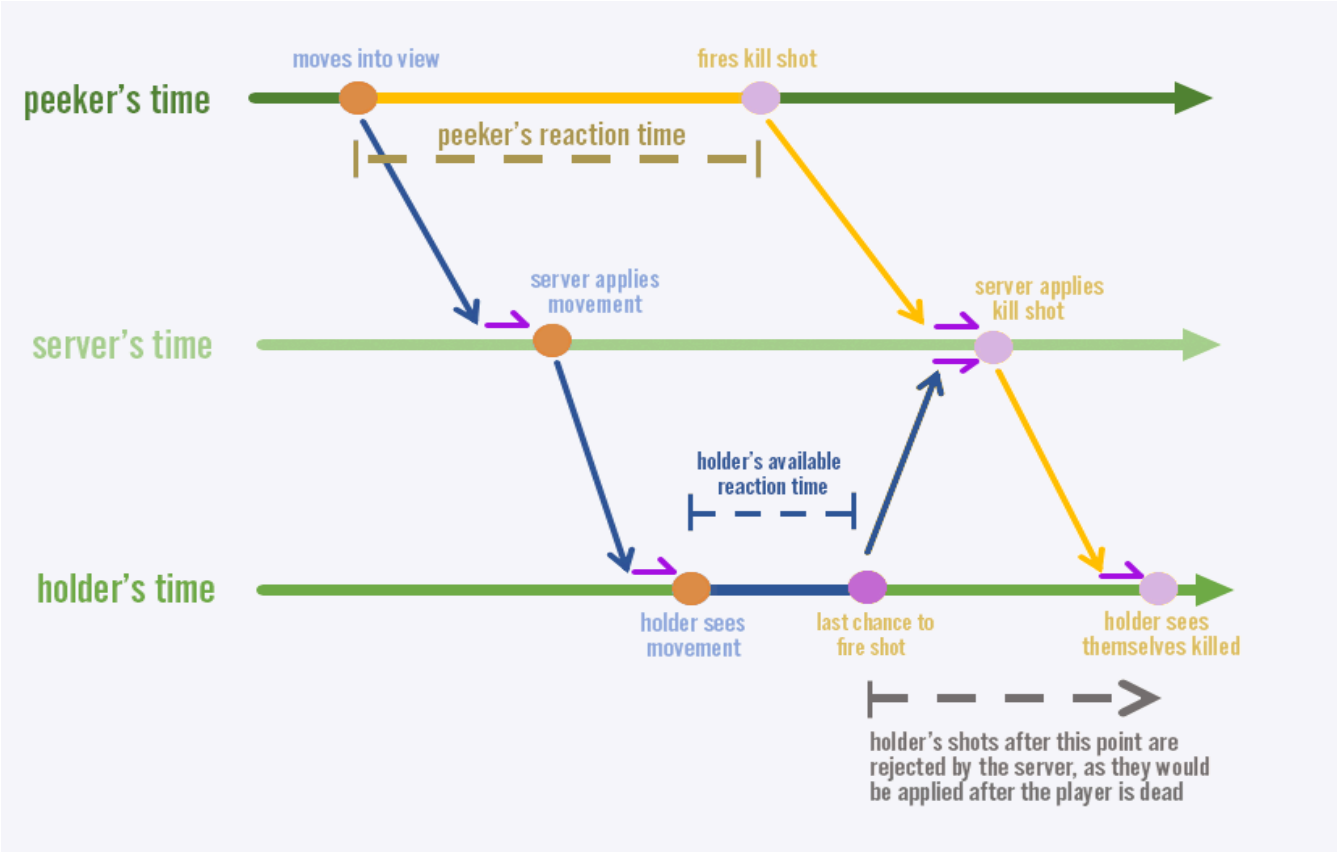
Design goals affected: game fairness, smooth movement, holder's advantage.

GAME FAIRNESS: THE NETWORKING BEHIND PEEKERS ADVANTAGE

The time-to-kill in *VALORANT* is extremely low. Most weapons can kill with a single, well-placed shot. When a player peeks out from behind a corner and comes face-to-face with an enemy, every millisecond counts.

Let's start by looking at what's going on behind the scenes when the peeker rounds the corner, sees their opponent holding the angle, and fires a shot.





The peeking player can see the holding player as soon as their computer processes their input and predictively moves them around the corner.

The holder won't see the peeker's movement until the following events happen:

1. The movement input is transmitted from the peekers client to the game server...
2. where the game server can process it...
3. and transmit the resulting movement to the holder's machine...
4. so the holder's client can apply that movement and draw the peeker in their new position.

Similarly, when the peeker fires the killing shot, the holder won't see it until the information follows that same path. When the server processes a lethal shot, it marks the victim as dead, rejecting any future shots from that victim. For the holding player to win the engagement, they must send up a shot with enough time to travel to and be processed by the server before the lethal shot from the peeker is processed. This creates a short window where the victim doesn't know they've been killed and may fire shots that won't count.

Before we jump to the math, let's talk about those little purple arrows.

SMOOTH MOVEMENT: BUFFERING INCOMING DATA

In a controlled environment, the server and clients might be able to apply movement data as soon as they receive it. Sadly, the internet is a highly unreliable network. Packets often arrive late or sometimes not at all. If the simulation was running against the newest available data, it would frequently find itself waiting on late or missing data and be forced to predict what happened instead. Mispredictions cause the server and client simulations to be out of sync, which lead to visual artifacts where characters pop or slide around in order to resync the simulation.

This sort of artifact goes directly against our design goal of smooth movement, and would make it difficult for players to track a moving target. The purple arrows above represent the common solution to smoothing out an uneven incoming data stream - buffering the incoming data. The buffer can fluctuate slightly in size as the data comes, while providing a smooth stream of data to any downstream systems.

Non-interactive applications like video streaming services can make these buffers very large, to hide relatively large network delays or outages. For real-time games, however, buffering is a delicate balancing act. While buffering incoming movement allows us to smooth out network issues, it delays when the player will see incoming movement and effectively serves as extra network latency.

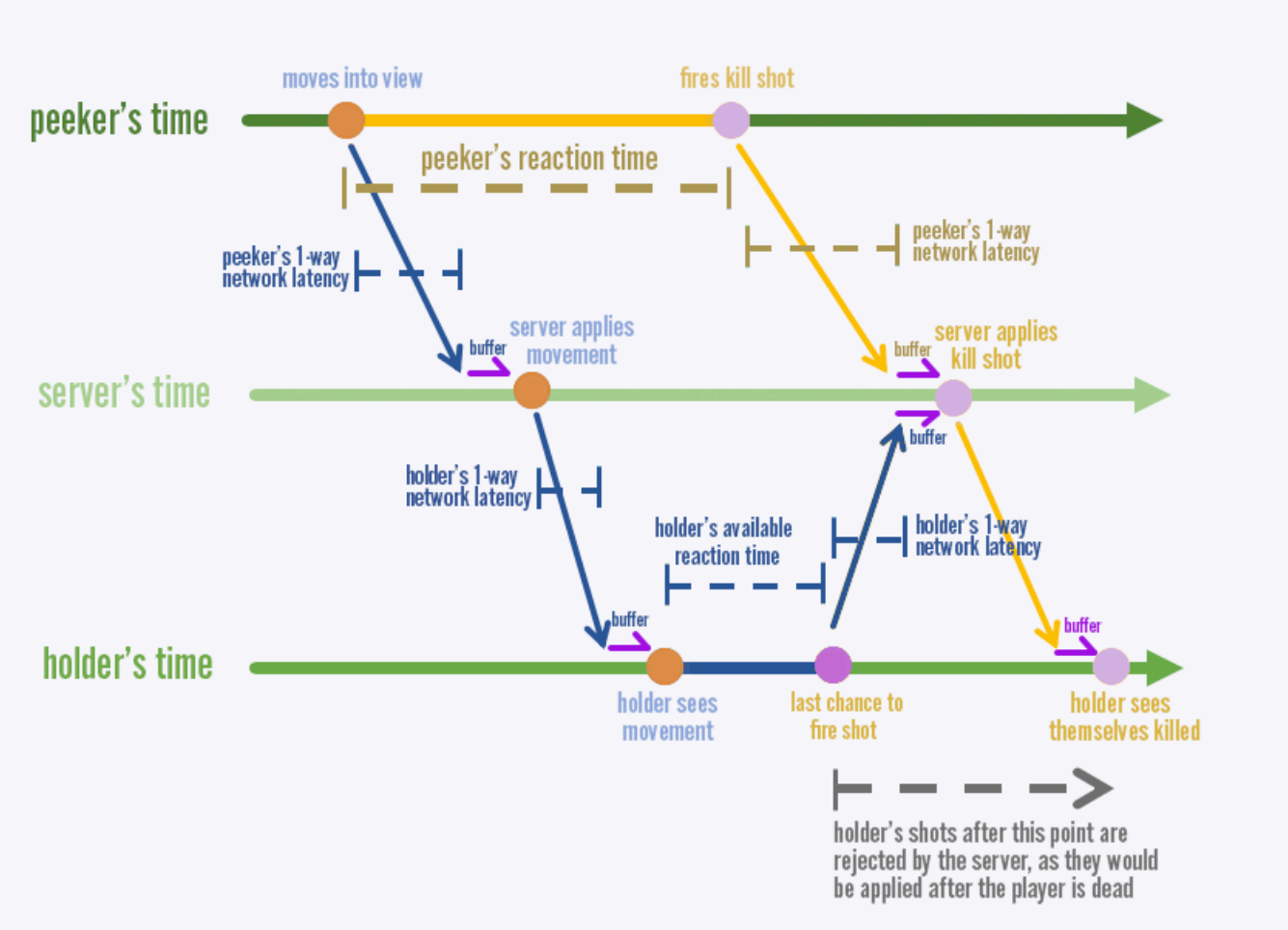
For simplicity, we'll say that "buffering" includes the full time from a move being received from the network to that move being processed and output (rendered to screen for clients or broadcast out to clients for the server). The duration of buffering is directly dependent on simulation tick rate (and render framerate on clients), because:

- Frames of movement data are buffered at tick-granularity
- Moves may arrive mid-frame and need to wait up to a full tick to be queued or processed
- Processed moves may take an additional frame to render on the client

This means that providing high tickrate servers and optimizing client performance help to reduce buffering and let you see a fresher view of the world!

HOLDER'S ADVANTAGE: MEASURING FAIRNESS THROUGH REACTION TIME

Let's get back to the diagram, this time focusing on the factors that contribute to peeker's advantage:



To make these combat engagements as fair as possible, our goal is to minimize the reaction time advantage that the peeker has over the holder. In fact, one way to express peeker's advantage is through the relationship between the peeker's reaction time and the maximum time for the holder to react before they're killed.

For either player to win the engagement, their shot must be processed on the server before the other's. The yellow path in the image shows the peeker moving into view, spotting the opponent, and sending a firing input up to the server. The blue path shows that same movement being sent to the holder and their opportunity to fire a shot before they're killed.

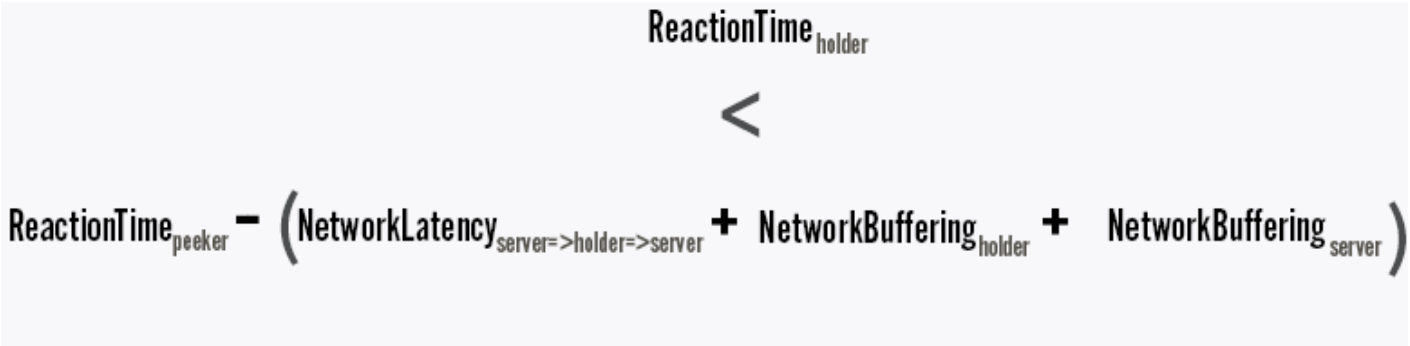
Expressed in equation form, in order for the holder's shot to be processed before they're killed, the following inequality must be true:

$$\text{NetworkLatency}_{\text{peeker=>server}} + \text{NetworkBuffering}_{\text{server}} + \text{NetworkLatency}_{\text{server=>holder}} + \text{NetworkBuffering}_{\text{holder}} + \text{ReactionTime}_{\text{holder}} + \text{NetworkLatency}_{\text{holder=>server}} + \text{NetworkBuffering}_{\text{server}}$$

$$<$$

$$\text{ReactionTime}_{\text{peeker}} + \text{NetworkLatency}_{\text{peeker=>server}} + \text{NetworkBuffering}_{\text{server}}$$

Rearranging terms and solving for ReactionTime(Holder):



$$ReactionTime_{holder} < ReactionTime_{peeker} - \left(NetworkLatency_{server=>holder=>server} + NetworkBuffering_{holder} + NetworkBuffering_{server}\right)$$

In other words, you can define how quickly the holder must react to survive by taking the peeker's reaction time and subtracting the holder's roundtrip network latency to the server and the network buffering on either side.

PLUGGING IN NUMBERS

Now that we have an equation to quantify the reaction time advantage that a peeker gets, let's plug in some values to get a sense for how big this delay actually is.

The buffering term can get complicated, so we'll need to wave our hands a bit here to keep this article to a reasonable size. As mentioned in the previous section, to keep things simple, we'll define the "buffering" term to include all time between a move being received from the network and being output (rendered to screen for clients or being broadcast out to clients for the server).

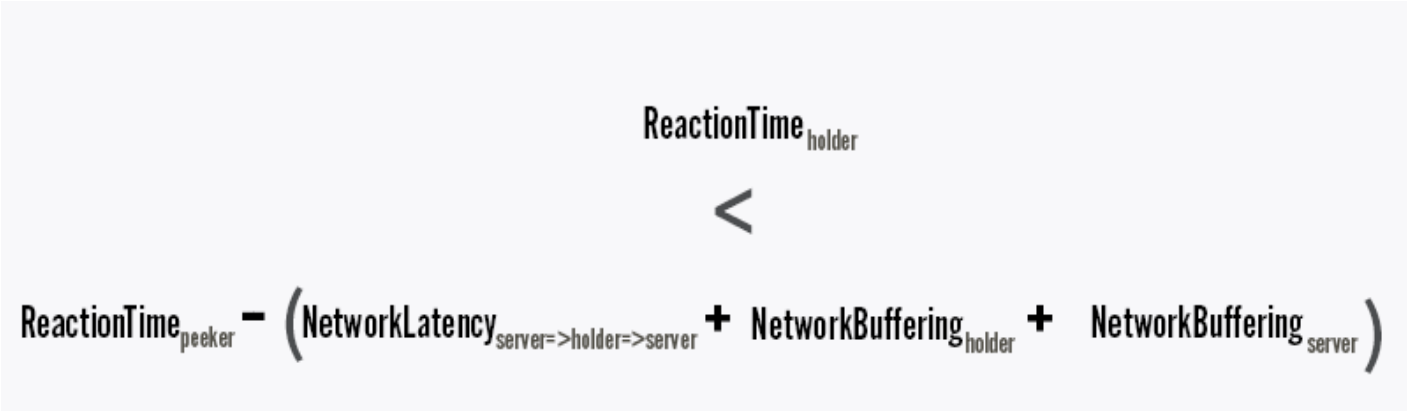
In this simplified model, we'll assume two frames of server buffering, which include:

- **0.5 frames** - the average time that incoming data must wait until the point in the frame where it's put into the queue
- **0.5 frames** - the server target for *actual* network buffering, this is how long incoming data should sit in the queue (on average) before being applied (to smooth out network inconsistencies)
- **1 frame** - a full frame where the move is applied and output (broadcast out to clients)

For client buffering, we'll assume three frames of buffering. This breaks down in the same way as server buffering, except that:

- The client target that we use for network buffering is one full frame, rather than a half frame
- Clients must render the results to the screen, delaying output by GPU & swap chain delay. This can vary from 0-2 frames depending on a lot of factors, but a 0.5 frame delay is a reasonable estimate.

Alright, let's plug those numbers in:



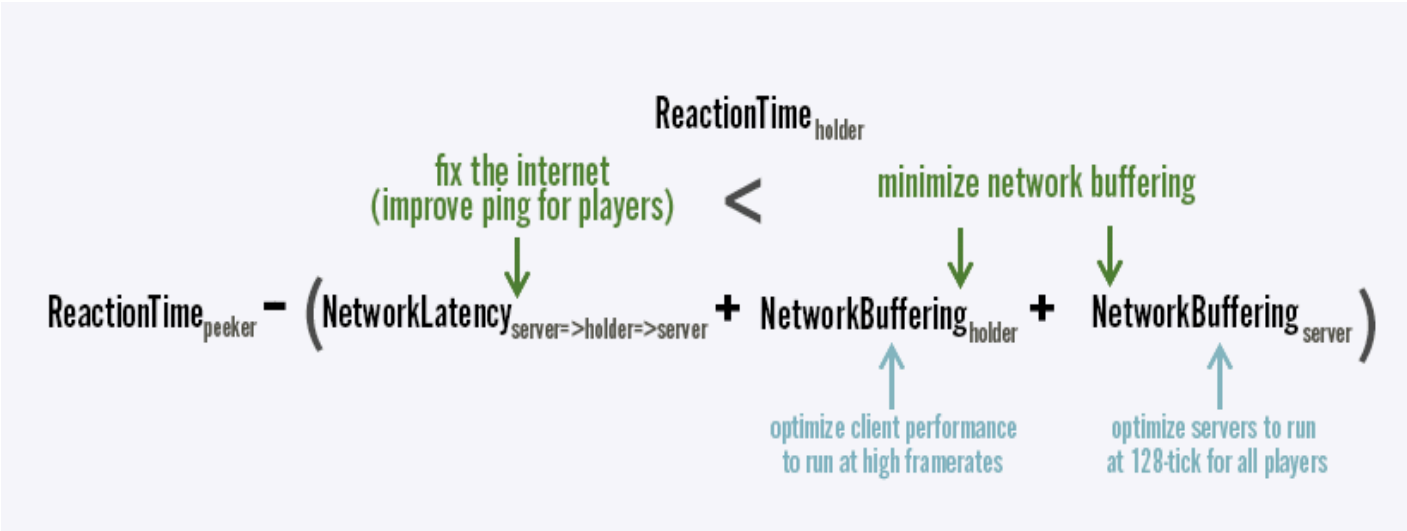
$$ReactionTime_{holder} < ReactionTime_{peeker} - \left(NetworkLatency_{server=>holder=>server} + NetworkBuffering_{holder} + NetworkBuffering_{server}\right)$$

$$ReactionTime_{holder} < ReactionTime_{peeker} - (60ms + [3 * 16.67ms] + [2 * 15.6ms])$$

$$ReactionTime_{holder} < ReactionTime_{peeker} - 141.25ms$$

This means that peekers have ~141ms longer to react than defenders. Considering that human reaction times often fall in the 300ms range, that's a huge advantage!

So then, what can we do as developers to minimize raw peeker's advantage?



In short:

1. We built Riot Direct, our own internet backbone, to minimize network routing delays and processing time across the internet.
2. We're standing up *VALORANT* servers around the world, ultimately aiming to deliver 35ms ping to 70% of our player base.
3. We optimized our servers to provide a smooth 128 server tickrate to all players.

4. We optimized the *VALORANT* game client to run at 60FPS on most machines from this decade and higher framerates for players with high refresh rate monitors.
5. We run clients and servers running with minimal buffering, targeting one buffered frame of movement data for clients and an average of half a frame of movement data on servers.

Filling in the equation for 128 tick servers, 35ms round trip network latency, and a client running at 60FPS:

$$ReactionTime_{holder} < ReactionTime_{peeker} - (35ms + [3 * 16.67ms] + [2 * 7.8ms])$$

$$ReactionTime_{holder} < ReactionTime_{peeker} - 100.6ms$$

Through investing in lowering latency for players and optimizing our servers, we're able to shave off ~40ms (28%) of our baseline peekers advantage.

By optimizing client performance, we can bring this down even further for players with high refresh rate monitors. Players running at a higher framerate will see updates sooner. For example, running at 144 FPS would bring the total advantage of an enemy peeker down to ~71ms (a 49% reduction)!

To help visualize the improvement, let's look at how far a player who starts fully behind cover is able to peek around a corner before their opponent sees the movement. We'll assume that the peeker is running at max speed with a Vandal.

That’s quite a difference, but how does it impact the results of combat?

THE IMPACT TO PLAYER EXPERIENCE

Early in the development of *VALORANT*, we needed to prove to ourselves that it was worth investing in tackling peekers advantage. Would the 40ms improvement we were targeting from Riot Direct and 128-tick servers have a meaningful impact on how the game plays?

To help answer this question, we ran some experiments. We ran several pairs of our highest skilled, evenly matched players through days of peeking scenarios. The holding player secretly picked a location to hold, and the peeking player (at a time of their choosing) peeked out from a known corner to start the fight.

We tested across a matrix of different server tickrates, network latencies, client framerates, and weapons. What we found was this:

- At the highest tier of competitive play, the differences between player reaction times become razor thin. The difference between winning and losing a gunfight in our experiments often came down to 20-50ms.
- Even though the playtests were blind (players weren’t told what conditions each round was running on), skilled players were able to accurately identify small changes (~10ms) to peekers advantage. Differences of 20ms felt very impactful to these players.
- For evenly matched players, a delta of 10ms of peekers advantage made the difference between a 90% winrate for the player holding an angle with an Operator and a 90% winrate for their opponent peeking with a rifle.

In other words - at the competitive level of play, very small advantages in reaction time can meaningfully change the outcome of combat.

APPROACHING ZERO

Minimizing the raw reaction time advantage of the peeking player is critical for reaching our goal of consistent holder’s advantage. Riot is investing heavily into Riot Direct and global infrastructure to help bring down latency numbers for our players around the globe. At the same time, the *VALORANT* development team will continue investing in further optimizations to client and server performance, even as new content and features are added to the game.

Aspirationally, we want to converge towards **ZERO** raw advantage for peekers, but in practice, that’s an unreachable target. Fortunately, in the tactical shooter genre, there are a number of design levers that help us achieve a true holder’s advantage. Combat in *VALORANT* is designed to give players defending a position a slight edge in intuitive ways.

1. Maps design plays a large role here. As an example, attackers typically have to peek from a single entry point to a site, while the holder can choose to watch from multiple angles. On average, it will take the peeker longer to find the holder and place their crosshair.
2. When the peeking agent moves around a corner, their shoulders appear before their head (and more importantly, their point of view). This buys the holder a few extra milliseconds to react.
3. Weapon fire is inaccurate while a player is moving, forcing the peeker to either come to a stop before firing or fire while inaccurate (and hope for the best). It takes some time to come to a complete stop, during which the holder can react.
4. Hit tagging in *VALORANT* briefly slows the movement of players who are hit with a non-lethal shot. This makes it easier for holders to secure a kill on a peeking player who is caught out in the open.

This isn’t an exhaustive list, but hopefully it illustrates some of the ways in which design and technology in *VALORANT* come together to swing the advantage back toward the holder.

Let’s move on to another interesting challenge.

MINIMIZING SIMULATION DIVERGENCE

Some divergence between the server’s simulation and a client’s predicted simulation is unavoidable. A few dropped packets or two players trying to move into the same space, and suddenly your client’s view of the world has drifted away from the server’s authoritative simulation.

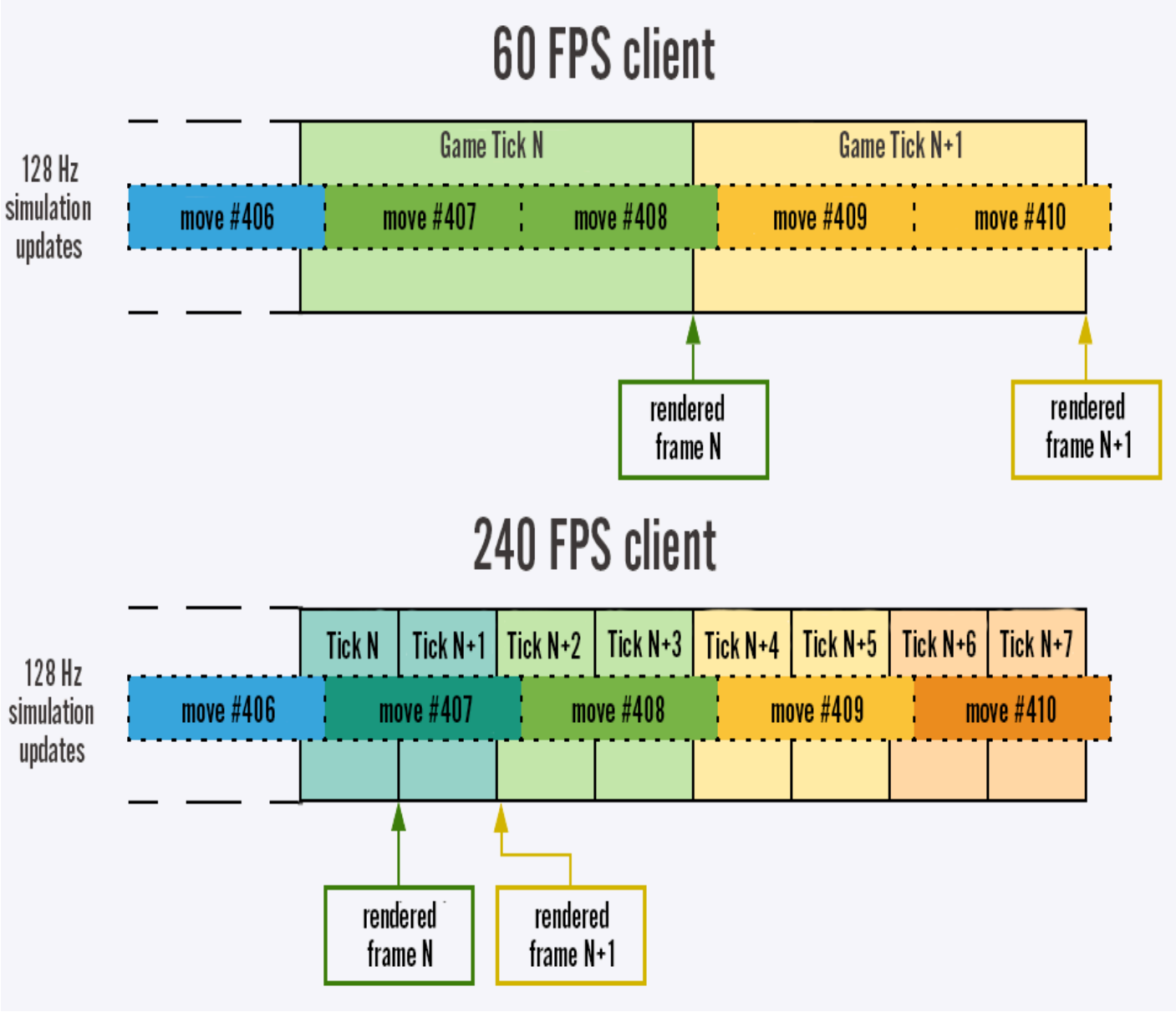
Our goal as developers is to limit the occurrence and severity of simulation divergence where possible, and to minimize the impact to the player experience caused by bringing the two back into line.

Design goals affected: smooth movement, support a wide range of setups

SMOOTH MOVEMENT: FIXED TIMESTEPS

The first step here is to make sure that differences in client and server framerate don’t cause simulation divergence. Running two 5ms physics updates back to back yields a different result than running a single 10ms update. This is due to the way forces are numerically integrated (accumulated and applied) during each update.

Importantly, this means that a client running a 30hz physics simulation will slowly drift from a server running its physics at 128hz and would need to be frequently corrected. To prevent this, we decouple our simulation updates from game ticks (your render framerate). Regardless of render framerate, clients and servers always update movement, physics, and other related systems with a fixed timestep: exactly 128 times per second.

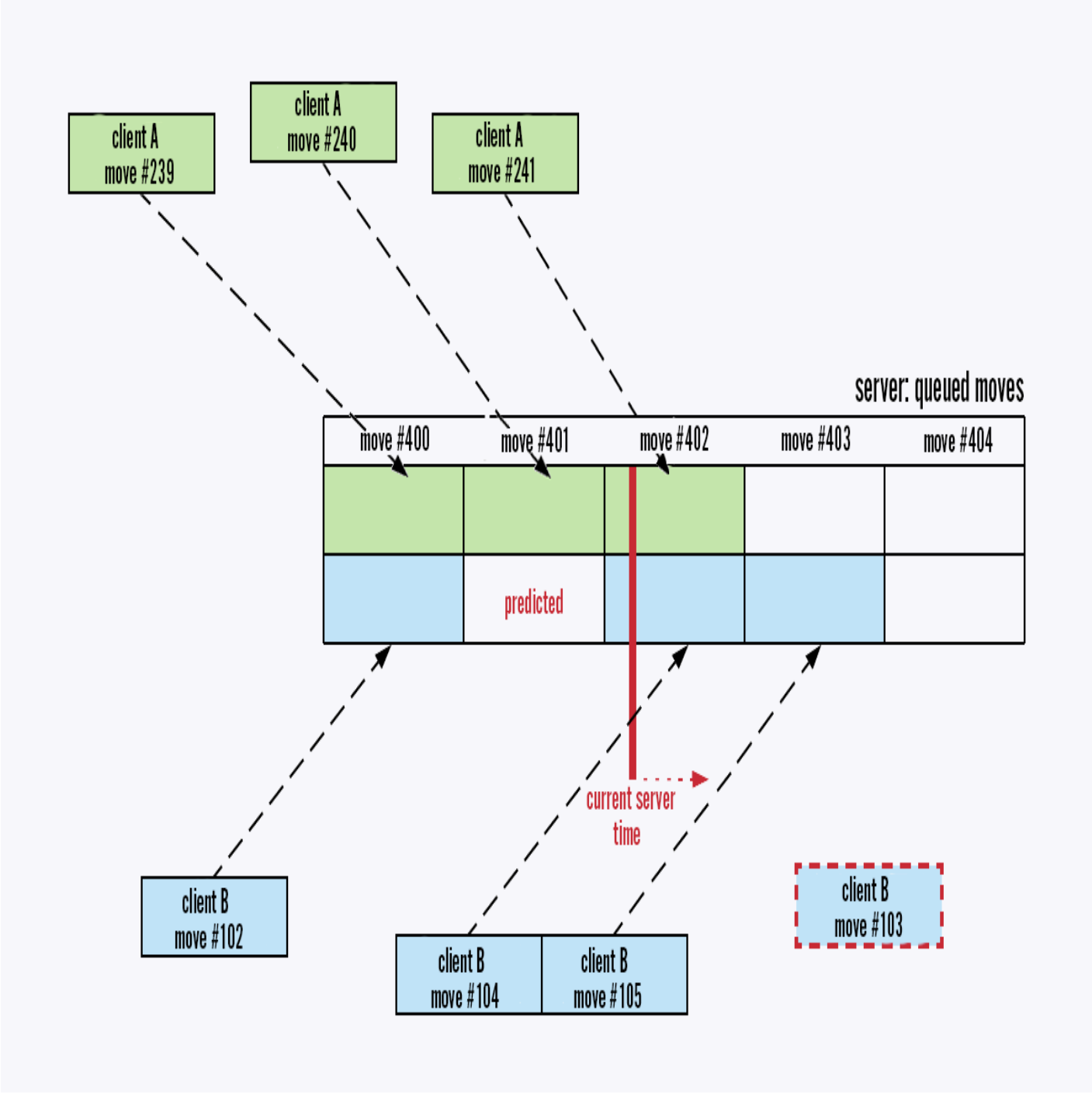


Shown above, a client running at 60 FPS will simulate multiple movement ticks per frame, while a higher frameright client might blend a single simulation update across multiple frames. We simulate slightly into the future (e.g. move #408) as needed to make sure we know exactly where the client will be at frame boundaries. We then linearly interpolate the state of the world within a move update to draw things exactly where they should be when the frame is rendered.

Now, with fixed move timesteps, the client and server can make an apples-to-apples comparison of their simulation results. When a client tells the game server “I just simulated move #408,” the game server knows *exactly* which move the client is talking about.

MOVE QUEUEING & SCHEDULING

Breaking up movement into consistently sized chunks allows us to easily pass it around and reason about it on different machines. Each client sends its input and movement results to the server, which runs its own version of the simulation and sends a correction down to the player if it disagrees with their results.



When the server starts receiving data from a client, it establishes a basis between that client's timeline and its own (client A's update #239 maps to server update #400). As each update arrives from a client, the server slots it into the corresponding slot in its queue.

Remember the network buffering we referred to in the peeker's advantage discussion? Here's where it comes in. The server sets up its basis (and adjusts as needed) to maintain a healthy queue of upcoming moves to execute. As mentioned above, we keep this queue as short as possible to minimize latency but long enough to smooth over the uneven rate at which the updates arrive.

As the server advances its own simulation, it executes the queued moves that each client sent, and transmits the resulting simulation state back to all clients. Sometimes, however, as shown for move #401 for client B above, the server hasn't received a client update when it's needed. In these cases, the server will predict what the client would have done. Usually, we guess that they continued to hold down whatever keys were being held in the last received update, since only a few milliseconds have passed.

But hey, sometimes we get it wrong. Occasional client/server disagreements are unavoidable.

SUPPORTING A WIDE RANGE OF SETUPS: RESOLVING DISAGREEMENTS

What do we do when the server and client get out of sync?

When one client disagrees, our top priority is to minimize the impact to the other nine players. The server commits its prediction as truth, and that client is told to adjust their simulation state back to match the server. This usually means instantly adjusting the positions or state of mispredicted characters back to where they should be. These corrections are rare, small in magnitude, and only seen by the player who encountered the underlying network issues & misprediction. The other nine players continue to see smooth movement.

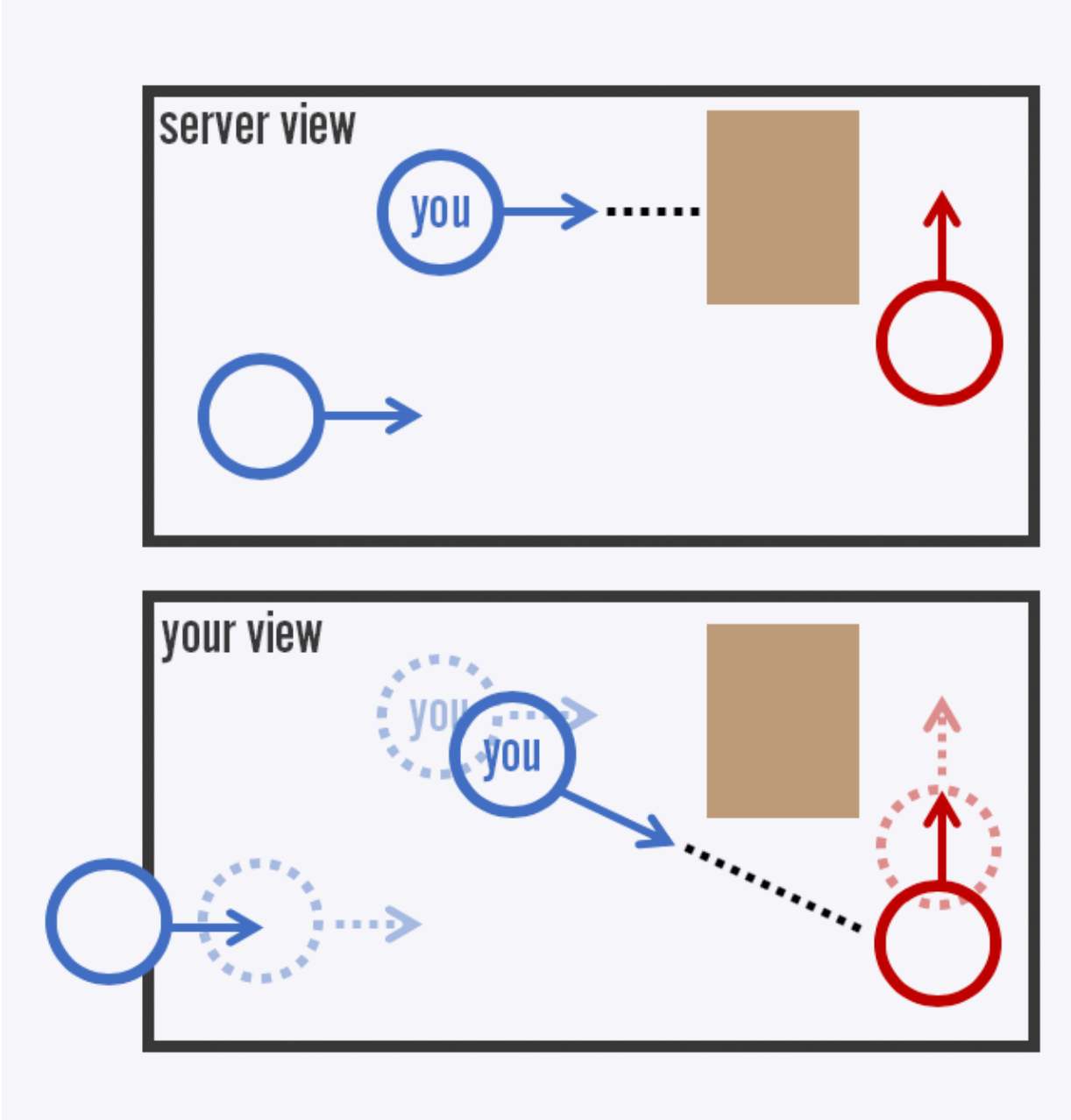
RESOLVING COMBAT ON THE SERVER

In real-time networked games, the world you see is always slightly out of date. This impacts many of the decisions we have to make to design a game like *VALORANT*, but in this article let's focus on what this means for combat.

Design goals affected: gunplay feels rewarding.

GUNPLAY FEELS REWARDING: CONFIRMING HIT REGISTRATION

It takes time for the server to send you information about where other players are. By the time it reaches your screen, they've already moved to a new location. When you fire at a moving target, you're firing at their past location. Moreover, they'll have moved even further by the time your input makes its way back to the server for processing.



Due to network delay, you'll always see yourself slightly ahead of where the server thinks you are, and you'll see other players slightly behind their current server positions. When the server hears that you've fired a shot, it needs to rewind the world state to what you were looking at when you pulled the trigger in order to accurately determine whether your shot landed.

Games that don't solve this problem force players to lead their shots on moving targets by a variable amount (based on their one-way network latency to the server). That's unintuitive for players and counters our goal of rewarding gunplay.

Fortunately, as we described in the previous section, the client and server know how their movement timelines are synchronized. This makes it possible for clients to send up exactly what time in the simulation they were looking at when firing a shot.

In reasonable networking conditions, this allows for near-perfect hit registration. That is, the client's prediction almost always agrees with the server about where a shot landed. Nailing this is critical for making gunplay feel crisp and rewarding.

It's also important to put some limits on how far the server is willing to rewind the simulation for hit registration. Without limits, a player with 500ms latency could kill you a half second after you'd moved behind cover. We tune these limits to prevent abuse, while allowing the vast majority of players in each region to play with near-perfect hit registration.

ADDRESSING CLIENT-SIDE ISSUES

We've said that we want to avoid letting one player with a poor network setup or hardware impact the experiences of other players, but that doesn't mean we're leaving these players high and dry!

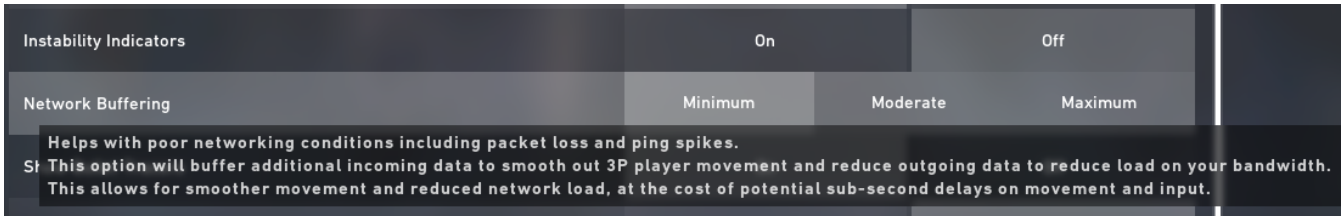
Design goals affected: support a wide range of setups.

SUPPORT A WIDE RANGE OF SETUPS: PERFORMANCE MONITORING

We can't solve players' hardware issues or network problems from our side, but we can help them to identify and address those issues on their own. To that end, we've built performance monitoring tools directly into the game, along with warning indicators that appear when a network or performance issue is detected:

Our hope is that arming players with more information about what's causing in-game issues will make it easier for them to help themselves (e.g. switching to a wired connection to reduce packet loss).

We also provide players with an option to change the amount of client-side network buffering to make tradeoffs between smoother movement and lower latency depending on what their local network conditions can support.



LOOKING FORWARD

We hope this article has been an interesting *peek* at some of the tech that goes into building a competitive tactical shooter.

We're happy with where the game has landed, but we're just getting started. In the near term, we're continuing to invest in improving game performance and standing up more servers around the world to deliver our 35ms latency promise to more players. We're also making improvements to clarify visual feedback around hit registration, so that players can more easily understand where their shots are landing in the heat of combat. Our entire development team is committed to improving *VALORANT* with our players and pushing the bounds of competitive integrity for years to come.

Thanks for reading, and glhf out there! Feel free to post comments below.

Posted by Matt deWet and David Straily