Our fast-action 2D shooter putting bandwidth to good use.

Netcode at Super Bit Machine: Multiplayer First

Nicola Geretti Follow 5 min read · Aug 2, 2021

28 1

First time I read about the Quake network model, it blew my mind.

Netcode is an all-encompassing term to define the logic managing your multiplayer… experience. It's a lot.

I used to be a progamer in the early 2000s, mainly competing in Unreal Tournament, but did my tour of duty with Quake3. I knew what good

netcode felt like, and knew how to play to its strengths, as a player. No idea what it took to actually make it.

ARMAJET was really my first foray into real netcode. I wanted to build a shooter with nostalgia for Gen X and early millennials, while showing Gen Z what their mobile phone was capable of.

What language to use? What protocol to use? What networking model to use?

If it's your first time working on a multiplayer game, or your 10th, you're probably asking the same questions. And there are lots of different answers.

This is the first article in a series that I look forward to publishing, covering our approach to Multiplayer and how we achieved our goals with ARMAJET.

Here's the full set of articles if you want to jump right ahead:

- **Part 2: Data Channels (Snapshots and RPCs)**
- **Part 3: Player Movement (Input and Physics)**
- **Part 4: Projectiles (Lag Compensation and Prediction)**
- **Part 5: Infrastructure (from Matchmaking to Play)**

Our Goals

ARMAJET was set out to prove that what John Carmack came up with in the 90s was still the best networking model to adapt to a modern shooter

designed to run on a phone from 2010 (the iPhone4, at the time).

I didn't write ARMAJET's core networking libraries. They're based off of Glenn Fiedler's fantastic netcode.io / reliable.io open source work. He's also a big Quake model fan. He contributed to the networking stack for Titanfall and Apex Legends, and I was fortunate enough to have him consult with us and show us what it took to deliver solid multiplayer that scales.

Our objectives were simple:

- Write a multiplayer-first game. You're a client programmer? Now you're a server one, too.
- Quake3 ran well on a wired 56k. Let's make this run well on a 3G.
- Our client is built in Unity, so use C# for everything.

Writing Multiplayer first

The 90s were all about RTS and single player shooters - internet was nascent.

In the 2000s we saw quite a few studios hiring outside dev teams to "add" multiplayer to their single player shooter games. Domain knowledge was still niche, but multiplayer required less expensive content (to some degree) compared to lengthy dialogue-filled, cutscene-requiring campaigns, and with with far higher reusability. It's what a growing audience wanted and devs started designing for multiplayer first, and for single player second.

Now, in the 20s (yup, we can say that) single-player FPS campaigns are being dropped left and right in favor of multiplayer-only, live-ops driven experiences. And, of course, the metaverse, which we're all going to live in soon.

So, what does it mean to write multiplayer first?

- All core gameplay logic is server logic.
- Want offline? Start up your server, and play through a loopback interface.
- Want dedicated? Build server-only logic/dependencies and deploy.
- Want peer to peer? Start up your server, and punch your NAT.

Discussing P2P really warrants a separate discussion, and with server costs decreasing with cloud, mobile network limitations, battery concerns, and the insurmountable issues of latency, network instability, host migration, NAT/firewalls, it's no wonder dedicated servers are the way to go.

The stack

Neon is the name we've given the stack we've build on top of netcode/reliable.io. This is what the full stack looks like, from bottom to top:

- Netcode.io
- Reliable.io
- Neon

- ARMAJET Game Server
- ARMAJET Unity Client

Netcode.io deal with establishing connections over UDP with *full-encryption* with signed packets. Its *minimal header size* really keep things packed to a minimum, providing protection against *man-in-the-middle attacks* and packet replay.

Reliable.io is an application layer that, in short, recreates the *reliability layer* of TCP, by *guaranteeing packet ordering* and *managing fragmentation*. This is used for our chat and a few other latency tolerant features.

Neon is the next layer that we've built in-house, which is responsible for *serializing the game state* and producing *delta snapshots* to be sent to each client connected. We pack down things to the *bit* where possible, and this is what allows our game to run a 4v4 deathmatch on a *64kbit connection* (8kb/s download, 3–4kb/s upload).

ARMAJET Game Server is the *gameplay logic* which contains all our custom physics, player and projectile entity simulations. It has a data layer that's accessible by the Unity client so that a developer can create entities and configure them in editor, serializing the data to a non-Unity dependent configuration.

Unity Client handles *rendering* on the client hardware, *capturing player input* (keyboard, mouse, controller and touch) to be sent to the game server at regular intervals.

How we ship it

The game server binary is essentially all of the above, minus the Unity client, with *zero Unity library dependencies*. This means when we build the server, we run a C# compiler on *full source code*, and spit out a headless binary we can deploy. To give you a sense of how thin that is, the whole game server with content and configurations is less than 5Mb.

By now, we automate it all, so the right commit produces a new binary and automatically deploys to the right environment, with zero downtime, including live, in minutes. We're pretty proud of that!

The client we ship for PC, Mac, iPhone and Android contains all of the above. All server code (minus some configurations) is *included*, so we can *run the server locally* when we need to. We don't have a single player campaign in ARMAJET, but we do have a practice mode you can play offline against bots.

Had we built the game any differently, we would've had to construct a new way to handle player input and rendering changes to the screen, which would've been extremely expensive to produce and maintain in the long run.

In the next article, we'll discuss the actual communication system used to send messages back and forth between client and server, and how it all works.

Netcode Series Part 2: Data Channels (Snapshots and RPCs)

Nicola Geretti Follow 7 min read · Aug 17, 2021

10

The in-game network monitor widget is hard to read, but never lies.

Ok, so we have a client running a phone, a PC, a console, it doesn't matter. What matters is that you're talking with a dedicated server through a connectionless communication channel, and it's time to make things move.

The focus of this article is identifying what data is core to the gameplay experience, what's not, and the process of deciding how the information

gets across.

When implementing any feature, you need to ask yourself the **2 golden questions**:

> ***Is the data latency important?— Is the data essential to game state?***

Speed über alles

Printed on countless t-shirts, only funny to a few.

UDP is the devil you know, and the one you need to get things done **fast**. We won't deep-dive into the protocol, but suffice it to say it's used for the most demanding real-time multiplayer games and audio/video communications.

If you're interested in knowing more about UDP and building a protocol on top of it, Glenn Fiedler has you covered once again.

If you don't consider network congestion, UDP is faster than TCP and with a smaller footprint. To be that fast, you need to shed some weight:

- **Connectionless communication**. There's no built-in handshake or packet ACK, which means you're not really "connected". If you know the port, you can start sending packets, no strings attached.
- **No packet resend on loss**. Your data may arrive, or it may not. Cross your fingers! 🤞 You can decide what's important and resend it, and discard what is not.
- **Unordered packets**. If your data arrives, it may be too late or too... soon?

So, TCP is like a bumper to bumper highway. There's no escape until all of the cars have reached destination. In UDP, you got your flying cars that may stray off the main path but are whizzing by each other.

It's really important to understand what the game needs, and for which functionality. Not having to deal with packet congestion is great, but you need to build a reliability layer.

With ARMAJET, we use 2 channels to build all gameplay features on:

- **Unreliable, Unordered channel**. For core gameplay and non-essential data. The game state travels on top of this as well as some events.
- **Reliable, Ordered channel**. For latency-tolerant events and commands. Chat and RPC-style calls are done here.

Unreliable, Unordered: Core Gameplay Snapshots

All core gameplay is built on the unreliable, unordered channel. Before jumping onto what that means, it's important to first define what **Core**

Gameplay data is.

> Core gameplay data is any data necessary to define the state of the game at any point in time.

The easiest way to think about it is, let's say you connect to a game already in progress. You need to know who's playing, the score, if any projectiles are flying by, etc.. Basically, the data required to build the scene to display.

Core gameplay answers a resounding **Yes** to both our golden questions:

> *Is the data latency important?* ***Yes*** *— Is the data essential to game state?* ***Yes***

All entities such as Game mode, Players, Projectiles, etc. make up the **Gamestate**. Entities can spawn, carry changing properties and despawn. Entities carry a limited amount of pre-determined properties called the **Net State**, which is the contract needed to run our **bit-packing** (yes, bits, not bytes!).

Net states are written out as **protobuffer definition files**, a format we created to quickly spell out variables and their serialization format. They're then automatically parsed and exported into C# files with utility functions. This is what a paired down projectile net state looks like:

```
class ProjectileEntityNetState : PropEntityNetState
{
   var InstigatorID : ushort;
   var VelX : float<0.001>;
   var VelY : float<0.001>;
   var IsWallStuck : bool;
   var FireTime : double;
```

```
    var WeaponID : ushort;
}
```

These support inheritance, so the `PropEntityNetState` also carries `Position` and `Angle` properties. All the entity net states put together **are** the Gamestate.

Here's where it gets interesting.

The Gamestate is saved on the server at every simulation tick. Tickrates in popular shooters are known to be as low as 20hz (*Overwatch*) and as high as 128hz (*Valorant*). Every tick stores the Gamestate into a circular buffer of **Snapshots**. Delta Snapshots are the key to effective data compression.

Delta snapshots are messages that are custom-tailored to each player, and they works like so:

- The server sends snapshots periodically to all clients connected. The first snapshot is the **Gamestate in its entirety**, including all entity net states (that's data pertaining to game mode, player entity, projectile entity, etc.).
- Since we can't rely on the protocol ordering packets for us, each snapshot has a sequence number. If the client receives a snapshot that has an older sequence number than the last received, it is **discarded**, as it contains old data we don't need (like a previous player's position).
- When the client receives a snapshot, it also **acknowledges the receipt** to the server by sending the sequence number.
- The server now **knows what the client knows**. It can generate custom-tailored snapshots based on the last acknowledged sequence

number, sending the client **only the data that has changed.** This is what's called a delta snapshot.

- If there is packet loss or jitter, there will be a delay in snapshot acknowledgement. This means the server will send increasingly larger snapshots, including more and more changed data that was never received by the client.

What's important to note is that no matter which delta client snapshot is the latest, a **full game state can be reconstructed**, meaning core gameplay state is maintained and the simulation can continue running.

You can find out more about the technical implementation of Delta Snapshots on Fabien Sanglard's deep-dive of Quake3's source code here.

I should note that Snapshots aren't suitable for all games. They're great for **non-deterministic physics-based games that require frequent synchronization on a limited amount of entities**. If you have a fully deterministic game (meaning the same inputs will produce the exact same result each time) such as a fighting game, input rollback systems are a better, more efficient way to go, with no need for positional deltas. If you're making an RTS with hundreds or thousands of entities, you'll need a different network model altogether.

Unreliable, Unordered: Events

These are the events describing volatile data that don't break the game state if they don't reach the client on time… or at all.

> *Is the data latency important? **Yes** — Is the data essential to game state? **No***

The railgun hit effect and confirmation. Not a big deal if the event packet is lost.

In ARMAJET, when a player fires a projectile that hits an enemy, a few things happen on the client side:

- The projectile explodes.
- The player's health is reduced.
- Damage numbers appear on top of the enemy.
- Triggers sounds and visual hit effects, using angle and hit offsets.
- If killed, a log shows the shooter's name, the weapon icon, and the victim's name.

The player's health change, as well as the projectile explosion are all part of the entity net state property changes, informed by delta snapshots.

Projectile **hits**, however, **are not entities**. They don't spawn, carry changing properties, or despawn. They are **discrete events**. Snapshots can tell the client that the enemy health was reduced by 30hp, but they can't inform if it was caused by a single 30hp hit, or 3x10hp hits.

Events travel on the same unreliable channel as Snapshots, but are **not repeated or stored**. In the rare occasion when a client misses the packet describing the hit, the opponent's health reduction will still reflect the hit(s) and the victim's demise, reliably. The client can still reconstruct the damage hit from the snapshots, it just won't know the exact projectile that caused it on the server.

It's important for the developer to create a rendering engine that can **tolerate missing data and delays**, so when they do ultimately happen, the player isn't faced with a buggy, glitchy-looking experience. Or even worse, completely different outcomes (Worms replays, anyone?).

Reliable, ordered

This reliability layer replicates the core features of TCP, without some of the overhead, or the need to use another protocol/port. We use this for any classic RPC-style commands.

> *Is the data latency important?* ***No*** *— Is the data essential to game state?* ***No***

Examples of this are:

- Chat
- Remote cheats
- Selecting Loadouts/Spawning
- Changing teams

Client and Server are guaranteed to get these in the right order, eventually.

Today we went over how gameplay data is split up into channels, and how to decide how the data needs to be transferred to maintain reliability and keep bandwidth use reasonable.

In the next article, we'll discuss how player movement works in a server authoritative environment, and how much there is under the hood to keep players looking smooth.

Netcode Series Part 3: Player Movement (Input and Physics)

Nicola Geretti Follow 5 min read · Aug 24, 2021

57

Your player exists in infinite parallel… simulations.

Player movement is the most important aspect of physics in an action game. Speed, acceleration, friction, and each impulse applied defines the feel for the game.

With ARMAJET, we spent a ton of time defining what makes player movement responsive, yet challenging enough to maneuver in the right

conditions. And a lot of that has to do with networking in a server-authoritative game.

Input Prediction

All clients **send a ring buffer of compressed player inputs** to the server at regular intervals. Some games send these as fast as the framerate allows.

A ring buffer means that the client doesn't just send the latest inputs pressed, but the **last several inputs**, delta compressed, so if (when) there is packet loss, the server can reconstruct tthe input queue and avoid data gaps where it doesn't know what the player did.

Because the **game runs on the server** and sends regular updates to all clients, there is an inherent delay from when the player presses a button, when the player actually moves, and when all other clients receive player's inputs to render animations.

There isn't much we can do about the delay in rendering other player's animations, but for our own player, we **can predict** what's about to happen.

Input prediction bridges the gap by running the game simulation instantly for the local player. When the client receives Gamestate updates from the server, the **snapshot includes the player's own inputs**. If the local and the server simulation match, everyone's happy. If they don't, you get what's called a de-sync, and the client smoothly lerps (linearly interpolates) to the server's simulated position.

Small de-syncs happen frequently. They're normal and expected, especially in a non-deterministic physics simulation. On the client, they trigger context synchronization, and one technique to run corrections smoothly is called **rollback**.

Client's Context Synchronization (Rollback)

The engine takes care of context synchronizations constantly. It corrects small offsets smoothly, while larger de-syncs caused by network interruptions or high congestion can trigger the infamous **rubber-banding**.

Rubber banding happens when positions are incorrectly predicted, and the simulation needs to **roll back the position** because the server played a different set of inputs than the expected one. The simulation on the server therefore does not match the player's local one.

So, for instance, if the player is stopped, and starts running, but the server doesn't receive inputs showing that the player started running, it will be stopped on the server until the new inputs are received. When the client re-synchronizes, it'll be warped back, where it **actually** is (remember, the server is the authority!).

In most cases though, synchronizations happen without the player noticing, and they manifest themselves in almost **imperceptible speedups or slowdowns**. This is how it all unfolds:

- Client sends inputs up to the server and instantly simulates the movement. We'll call the local simulation the **primary context**.

- Server plays the inputs and simulates the movement, then sends back down to the client a snapshot containing the **latest player input played**.
- If the client detects the server has **played a different input** than the local one, or there is a **significant positional divergence**, a de-sync is triggered.
- During the de-sync, the client continues running the local simulation on the primary context, but also starts simulating a **secondary context**. This context is the result of the **last known correct player state** (right before the point of divergence), **plus all local inputs** simulated up to the last. The result is the final, correct position to the best of the client's knowledge.
- Now, we can transition from the primary context to the secondary context over a short amount of time, **all while still simulating both**.

This approach allows for a smooth transition, constantly adjusting for unexpected changes in position, velocity or inputs.

Many different types of games can apply rollback networking successfully. In recent years, fighting games such as MK11 and Smash have jumped on the bandwagon and adapted GGPO, a popular rollback SDK for peer-to-peer games, much to the fighting community's satisfaction.

Client's Input extrapolation and buffering

So we covered how we move our own player and synchronize it with the server. What about everyone else?

Snapshots deliver all the necessary positional deltas. We move other players to the positions, play animations based on inputs pressed, and **interpolate over time** to make them look smooth. Because our physics are not fully deterministic, we don't simulate other player movement with inputs and real physics, which saves a lot of CPU time on the client (remember, ARMAJET was built to run at 60fps on an iPhone4!).

Regardless, issues arise when there are delays in snapshot processing and the client is left with no data to process.

So what do we do? Stop the player and resume when we have the data? Here are 2 ways to combat that:

- We buffer snapshot and delay player's rendering. The delay is called **Interpolation back time**. A high value increases jitter tolerance on unreliable connections, but it also tells the client an ugly truth: the enemies are way past where they seem to be. Try hitting someone that's rendered 200ms in the past when boosting off a jumper. This effect is what forces players to "lead the shot" in order to hit an enemy in certain shooters.
- As soon as there's data missing, we start a physics simulation, repeating the last player inputs for a period, similar to the local player's simulation. This is called **Input extrapolation.** Allowing extrapolation means the client is now guessing where the enemy is headed. Over the course of a couple of frames, it's likely that the opponent has not changed their course, predicting correctly. When doing it for longer, it's most certainly headed towards de-syncs and the eventual enemy's positional warping.

Our solution? A little bit of both. We keep a static amount of interpolation back time, something that requires a **light amount of projectile lag**

compensation (more on this on the next article), and run input extrapolation on the more rare occasions where downstream connections are being spotty.

The result is latency tolerance of 150–200ms worth of jitter in most circumstances without any discernible visual glitch.

Server's Input extrapolation and buffering

The same logic is applied to the Server. The server holds an input buffer, which delays input processing to allow for constant, smooth, uninterrupted simulation.

Unlike the client's static Interpolation back time, we have implemented an **adaptive input buffer**, that changes based on the client's connection quality. In short, if the client's connection is fairly stable, there is little to no buffering, making firing and moving run almost immediately once received. As instability kicks in, the buffer increases, allowing the client to recover at the cost of some processing latency.

When the buffer runs out, extrapolation is applied for a short period, repeating the last player's inputs just like the client does.

While there's much more under the hood, these are the main techniques used to maintain smooth player positioning keeping in mind packet loss, latency and jitter.

In the next article, we'll go over how projectiles work and what lag compensation is, an important technique to make the clients hit their targets where they see them on a variety of network conditions.

Netcode Series Part 4: Projectiles (Lag Compensation and Prediction)

Nicola Geretti

Follow

5 min read · Aug 31, 2021

69 2

A whole lot of projectiles in need of hit detection.

Great, so now you know how players move and how they're synchronized between client(s) and server. Projectiles, like players, are entities that can spawn, change their properties, and despawn.

The only difference is that they're fully deterministic, and that makes a big difference. It means given an origin point and a vector direction, every single projectile will behave exactly the same.

Because ARMAJET is a 2.5D platformer where you can literally dodge projectiles, the game needs to consistently, and clearly, visually represent projectiles in all latency situations. Let's dive right in.

Projectile Data

Projectiles are relatively light entities. Their main properties, carried by the net state, are:

- Instigator ID (the player that fired)
- Weapon ID
- Firing time
- Position Vector
- Direction+Power Vector

Unlike players, which have their input/position constantly communicated to all clients, projectiles have their **payload sent only when spawning in**. When the projectile despawns, it means they hit static geometry, they expired, or they hit another player, which will trigger a separate damage event.

It's common for large scale games (the Battlefield series or any Battle Royale game out there) to carry **update masks** that reduce the amount of updates to pack onto snapshots, based on entity type or distance from the player. Another way of **culling data** is to calculate the frustum viewport

on the serverside as well as the distance between the player and the projectiles, to gauge whether the player can even see or hear the projectiles based on their trajectory.

In ARMAJET, we transfer less than 20 bytes once per projectile spawn, so culling this data isn't a major concern.

Projectile Lag Compensation

Lag comp, for short, comprises a series of techniques that enable your projectiles to hit where you expect them to.

In the previous article, we covered player physics synchronization, and the fact that the client sees enemies in the past due to latency and interpolation. So if a player were to aim at a moving enemy where they are, their projectile would most certainly miss the target.

The are a multitude of ways, depending on the type of game, to remedy this issue. In our case, we implement a mix of client and server lag compensation techniques to significantly reduce player rage quits.

Client and Servers are constantly measuring the latency between them. **RTT**, or **Round Trip Time**, is the measure we use to determine the latency between client and server. It's not determined by classic ICMP pinging as that doesn't take into consideration application-layer delays due to packet processing, tickrate, etc., so half of the RTT gives us a good picture of how long it takes for a message to be sent, received and unpacked for processing.

This is what happens, every time you fire your projectile in ARMAJET:

- Client presses the fire button,
- Client locally predicts the firing of the projectile, based on known rules of the game (Is there ammo? Is it reloading? Is the player stunned? Etc.),
- On high latency situations, the **client artificially delays firing prediction**, to communicate latency to the user and **reduce the burden on server lag comp**. More on this later.
- When the server receives the input, it is put in a **queue** for simulation.
- During each projectile simulation tick, all entities are **rewinded** to a certain amount of time, the projectiles are moved, collision logic is run, and all entities are then **placed back where they were**. All of this happens in a fraction of a ms. This rewind time is equal to:

```
rewindTime = inputQueueDelay + interpolationBackTime +
clientLatency - clientFiringDelay;
```

- `interpolationBackTime` and `clientLatency` comprise the delay the client suffers from displaying the correct opponent position.
- `clientFiringDelay` also helps reduce the rewind time by placing a **delay on the client's local simulation** itself, at the expense of **responsiveness,** which is why the higher the delay on the client, the less lag comp is required.
- `rewindTime` has a **maximum value cap**, so players with high latency can't hit enemies with a significant delay advantage.

This last part is very important because lag comp isn't free. On the receiving end (the enemy's client viewport), the paradoxical "**hit behind the corner**" effect is possible, where the enemy has turned a corner, and

yet are still vulnerable to hitting, because lag comp rewinds their server simulation in the past to assist with the instigator's latency. This is why shifting some of the delay onto the client with high latency helps keep the game fair for everyone.

From my experience as a pro-player (and the overwhelming customer feedback), it's far more frustrating to not get your projectile to hit where you're firing as opposed to getting damaged while fleeing. And if you don't understand why you got killed, this is one of the reasons why *Call of Duty* invented the killcam: to unmistakably show what **the server saw** when you got headshotted by the guy you just called a cheater.

There are other tricks related to physics that we implement under the hood, particularly effective when there's **high jitter and latency spikes**, but they go beyond the scope of this article. If you're interested in more about this, leave a comment or shoot me a message.

Projectile Prediction

Because projectiles are fully deterministic, and their trajectory is determined by their origin position, velocity vector, and properties of the weapon they were fired from, we can safely **predict their full lifetime**.

When colliding with dynamic entities, however, such as other players or even other projectiles, predictions can start drifting from the real simulation, such as in cases where projectiles:

- Stick to other players (arrows)
- Push other players (explosives)
- Are pushed or triggered by other projectiles (chain explosions)

The explosives with knockback are the most evil of them all. If you simulate their explosion, because the local client detected colliding with an enemy, you then also have the option of simulating **pushing the enemy away**. If your simulation is right, the effect is smooth and instantaneous, but rewinding when you're wrong will look terrible, teleporting the enemy. In ARMAJET, we use a blend, where the hit is locally simulated, but the push is only applied when we receive server confirmation, which maintains the snapshot positional interpolation contract.

Matt Delbosc has a very interesting GDC talk about peer-to-peer vehicle replication to deal with vehicle collisions in Watch Dogs 2.

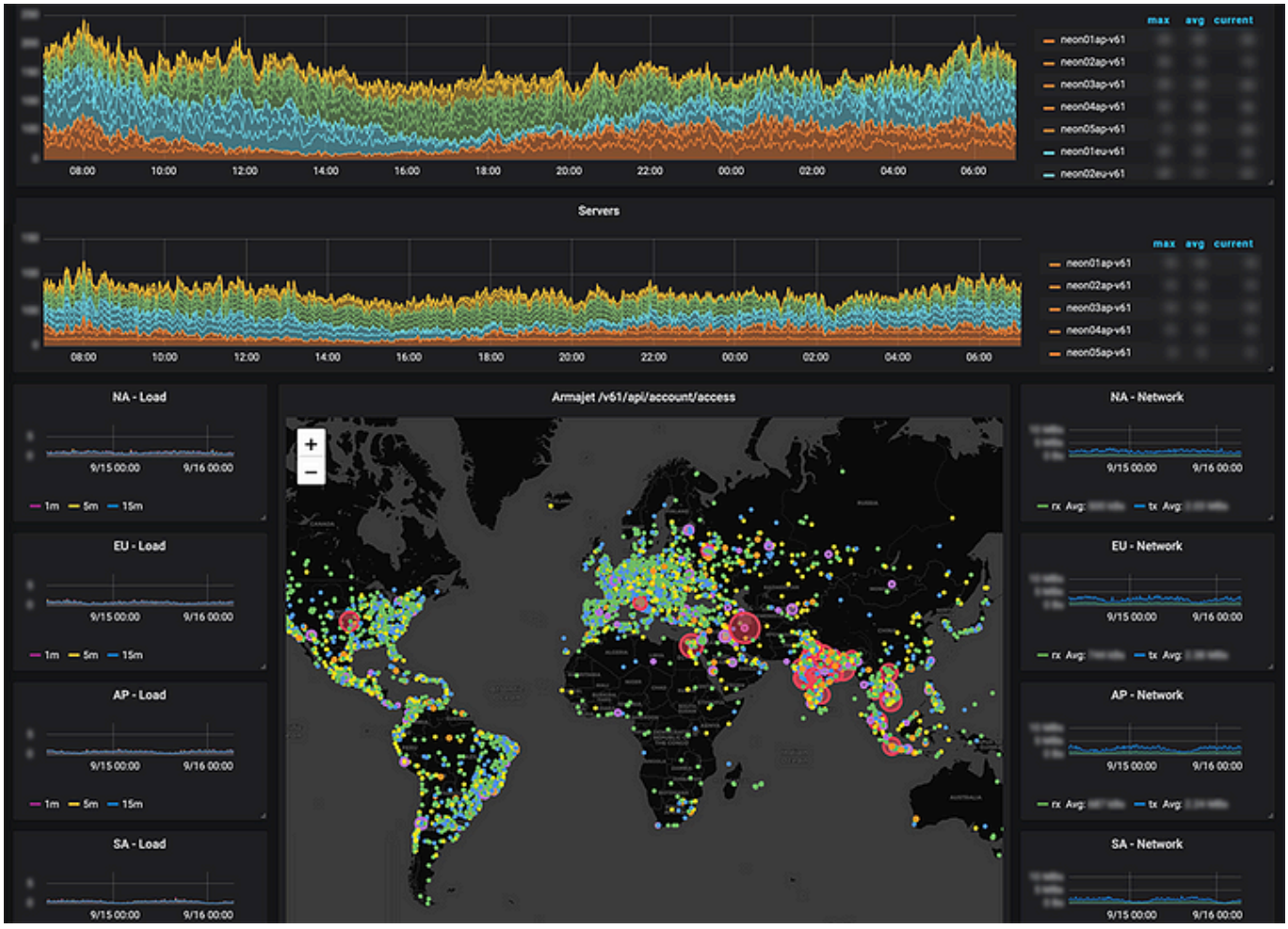
Today we looked at the data payload of a projectile and how client and server simulate projectiles in the interest of keeping the game fair, while assisting players in high latency scenarios.

In the next article, we'll go over our infrastructure, what it takes to quickly and securely transport a player from matchmaking to the server in the right region, and how we scale it all.

Netcode Series Part 5: Infrastructure (from Matchmaking to Play)

Nicola Geretti Follow 5 min read · Sep 16, 2021

2



One of many custom dashboards to tell us how each region is doing.

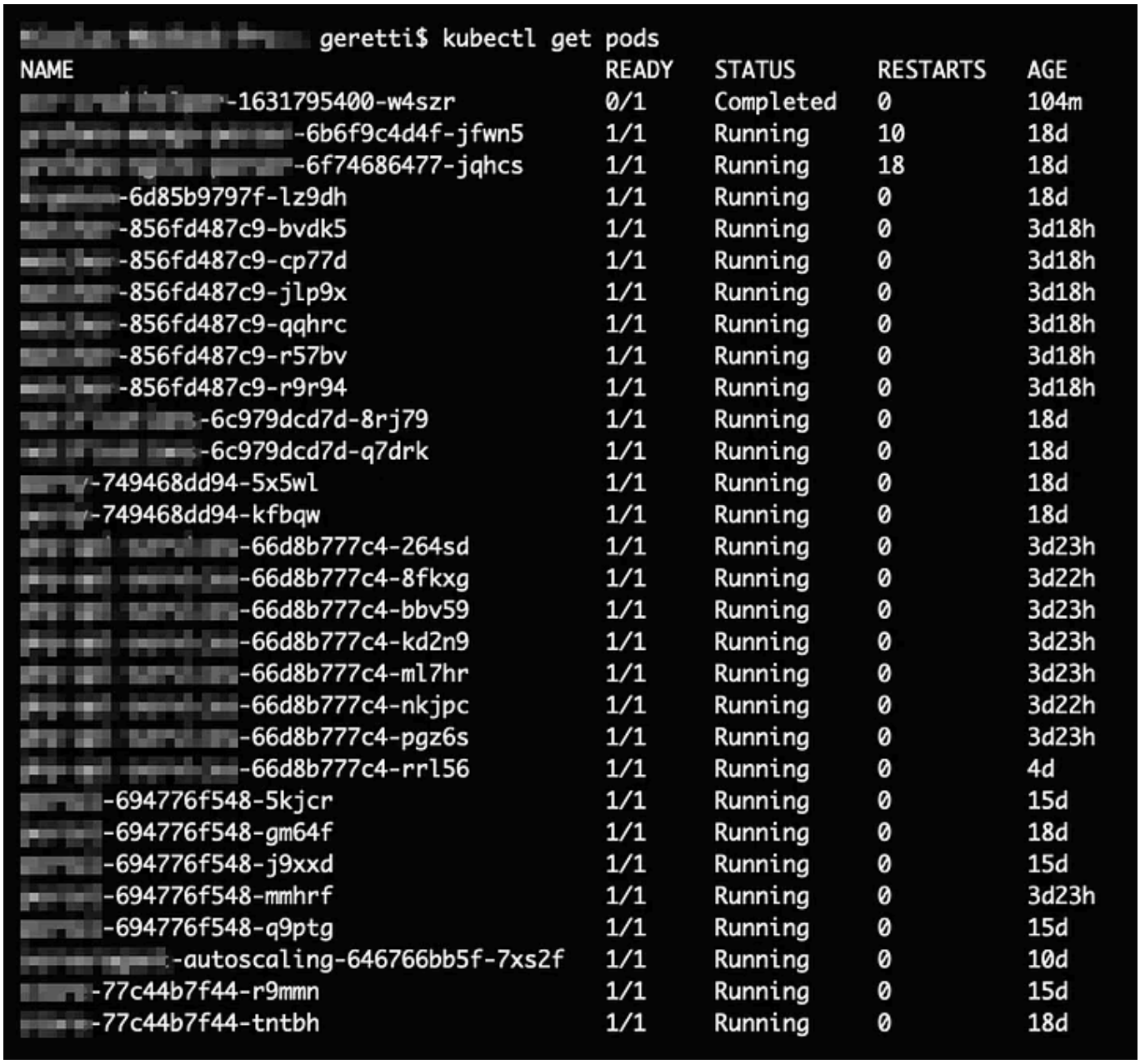
Getting clients to play on a server quickly and effectively is a big feat. That's certainly what, as a game developer, you should be focusing on.

In an increasingly social gaming sphere, however, you're going to need infrastructure and services to support scaling game servers, account management, matchmaking, party systems, etc.

There's tons of services out there you can use out of the box to support your efforts. Microsoft, Amazon, Unity, Epic all are building software to service game developers with their needs. Mainly, of course, to attract developers with the promise of an easy and fast integration, and an ultimate ecosystem tie-down. If you're new to services, I recommend starting there unless you have highly specialized needs.

My background comes from creating game services from scratch and running them on bare metal and the cloud, so at Super Bit Machine, we tailored our solution to the needs of our first game, ARMAJET.

Services and orchestration



```
[redacted] geretti$ kubectl get pods
NAME                                    READY   STATUS      RESTARTS   AGE
[redacted]-1631795400-w4szr             0/1     Completed   0          104m
[redacted]-6b6f9c4d4f-jfwn5             1/1     Running     10         18d
[redacted]-6f74686477-jqhcs             1/1     Running     18         18d
[redacted]-6d85b9797f-lz9dh             1/1     Running     0          18d
[redacted]-856fd487c9-bvdk5             1/1     Running     0          3d18h
[redacted]-856fd487c9-cp77d             1/1     Running     0          3d18h
[redacted]-856fd487c9-jlp9x             1/1     Running     0          3d18h
[redacted]-856fd487c9-qqhrc             1/1     Running     0          3d18h
[redacted]-856fd487c9-r57bv             1/1     Running     0          3d18h
[redacted]-856fd487c9-r9r94             1/1     Running     0          3d18h
[redacted]-6c979dcd7d-8rj79             1/1     Running     0          18d
[redacted]-6c979dcd7d-q7drk             1/1     Running     0          18d
[redacted]-749468dd94-5x5wl             1/1     Running     0          18d
[redacted]-749468dd94-kfbqw             1/1     Running     0          18d
[redacted]-66d8b777c4-264sd             1/1     Running     0          3d23h
[redacted]-66d8b777c4-8fkxg             1/1     Running     0          3d22h
[redacted]-66d8b777c4-bbv59             1/1     Running     0          3d23h
[redacted]-66d8b777c4-kd2n9             1/1     Running     0          3d23h
[redacted]-66d8b777c4-ml7hr             1/1     Running     0          3d23h
[redacted]-66d8b777c4-nkjpc             1/1     Running     0          3d22h
[redacted]-66d8b777c4-pgz6s             1/1     Running     0          3d23h
[redacted]-66d8b777c4-rrl56             1/1     Running     0          4d
[redacted]-694776f548-5kjcr             1/1     Running     0          15d
[redacted]-694776f548-gm64f             1/1     Running     0          18d
[redacted]-694776f548-j9xxd             1/1     Running     0          15d
[redacted]-694776f548-mmhrf             1/1     Running     0          3d23h
[redacted]-694776f548-q9ptg             1/1     Running     0          15d
[redacted]-autoscaling-646766bb5f-7xs2f 1/1     Running     0          10d
[redacted]-77c44b7f44-r9mmn             1/1     Running     0          15d
[redacted]-77c44b7f44-tntbh             1/1     Running     0          18d
```

List of active pods in a specific environment, autoscaling to our needs.

Our services are APIs that are developed, deployed, managed, and virtually separate from one another, with no cross-dependencies. While that's *mostly* true, the game still ultimately needs all client-serving applications to function.

There are plenty of articles on the web arguing in favor of microservices or monoliths. We have a mix of the two to take advantage of fast iteration on new projects, while not having to split up our main legacy backend.

- **Platform**: serves all game data and authentication for players. This is the first point of contact for the game and the largest service. It's the only one allowed to make permanent changes to account status, progression, etc.
- **Matcher**: accepts requests from players who want to start playing, and runs queries against the requested criteria. When a match is established, it requests a new server instance.
- **Portal**: entry-point for all servers requested. Manages scaling regions and routes requests from clients who want to play, as well as Matcher.
- **Social**: it provides functionality to enable real-time push messaging, party grouping for players, chat, etc.

All services are built by our **Jenkins CI** pipeline, spit out a **Docker image** which is then uploaded to the corresponding environment and deployed via **Kubernetes**. This goes from local bare metal for internal development environments, all the way to production cloud (AWS, Rackspace, etc.).

As an application developer, this is all transparent. Once a **commit to a specific repository branch and tag**, the app gets built and deployed to the target environment in anywhere between 1 to 5 minutes depending on the size of the service. The `rollout restart deployment` Kubernetes command is configured to launch the new versions and shut down old ones without any service interruptions.

Kubernetes has resource allotments configured on a service basis, so as it horizontally scales depending on traffic, we're keeping our costs to a minimum.

Platform

Our platform service owns all the data and is the only service that has read/write properties on it. If other services need to access player data, this is their way in.

This is a classic REST API, with connection to relational (SQL-like) and Redis type datastores. Safe, reliable, not reinventing the wheel here.

Platform serves:

- Player authentication, via Device ID, Platform ID (Steam/GameCenter/GooglePlay/etc.) and email (Superbit ID)
- Player data retrieval (inventory, stats)
- Game data retrieval (list of items, their stats, etc.)
- Game's meta progression
- Game Server's endpoints for player auth and score submission

Matcher

Our Matchmaking service connects players together, runs calculations and groups players together resolving the final rules. Once the matchmaking session is deemed completed, Matcher **requests a Server instance** from Portal, which returns a qualified server ready to accept players.

Matcher is an API made for polling. It allows websocket push connections, but for most cases, polling it to receive updates is sufficiently fast even for a mobile in-and-out game.

It connects and queries a Mongo datastore to **manage active matchmaking tickets** and find the best matches.

It takes into account:

- Party size
- Criteria selected (game modes, number of players, etc.)
- Device type (PC, Mobile)
- Input system
- Region Latencies (pings)
- Skill Rating
- Win/loss history

All these variables are used to do multi-dimensional searches to find the best match, and **autotune based on how many players are online**.

There will be another article going in depth about our matchmaking and how it works at scale.

Portal

It's the entry point into the Game Server. Portal accepts requests for new servers from Clients and Matcher directly. It launches new servers, manages all active server instances, and routes players.

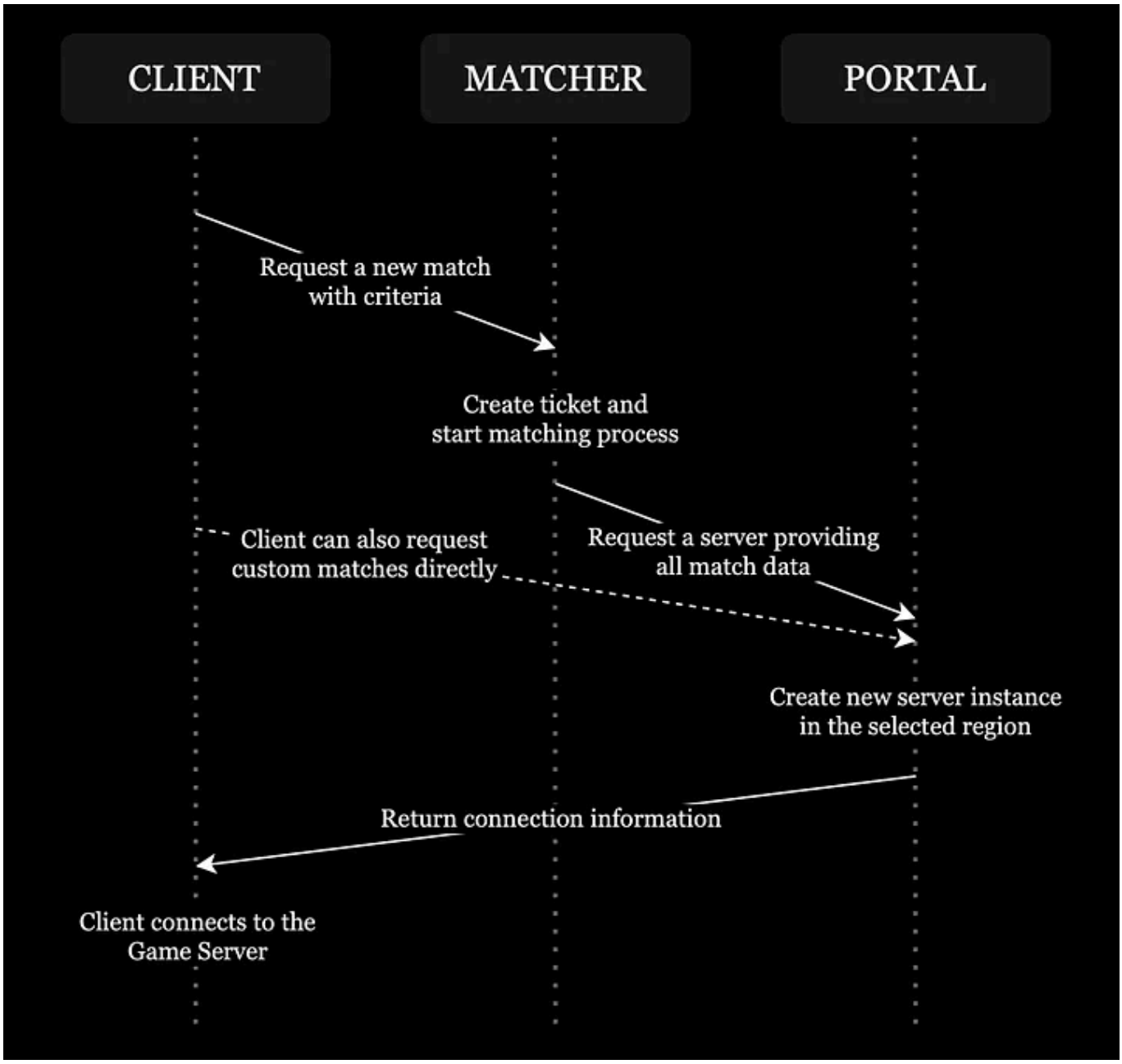
After launching the server, Portal returns a host so the clients can connect over UDP via IPv4 or IPv6.

Like Matcher, this is also a lightweight API, and it's the only one that actively manages **multi-region game servers**.

Portal **receives heartbeats from all active servers**, alongwith the active players, rules and state of the games. As long as a game server has a region code associated with and is deployed within our security group, it's recognized as a new potential destination and registered in an active server region.

I'm pretty proud of this one because we can decide to whip up a new supported region and deploy it on AWS, and all client and services will now be open to the new destination automatically. We've done that a few times with our Indian/Australian/Korean regions as we were testing quality of play and comparing latencies from our main APAC servers in Singapore.

The full connection flow looks like this:



From the moment they click play, to when they actually play.

Social

This isn't an essential service, but it enables partying up as well as other components that enable group experiences and social graph. These are:

- Party system
- Chat

- Social Real-time messaging system (friending, etc.)

Social is an API with Redis Pub/Sub as backbone. It only manages transient data and is responsible for all real-time connectivity outside of the game server itself. Websockets are used here, with classic polling fallback.

Today we looked at most of the services that are required to bring a player from a **game homescreen to a game session**. You can build all of this as microservices or a monolithic backend, you can use off the shelf software or plug into the many SaaS on the market today.

On the next article, we'll dive into the nitty gritty of our cross-platform matchmaking system and how it adjusts to any amount of CCUs to provide the best experience.

Netcode Series Part 6: Matchmaking Cross-Platform and Cross-Region

Nicola Geretti Follow 6 min read · Sep 22, 2021

1 1

Tuning for perfect balance… or just good enough for most.

> “This game has great matchmaking!” — said nobody, ever.

Let's be honest, matchmaking is the one component of online gaming that never stops being criticized, even more so than netcode. And for good reason. Even when it functions just as you expect it, it will still not be good enough for some players.

The shuffle function on your favorite music player isn't really random. People want a random song picked from their playlist, but they don't *really* want true random. If you just listened to this one track out of 10, you probably don't want to hear it next.

> The best algorithms understand how users got to where they are, and try to deliver the best outcome to keep them going.

Matchmaking is no different, except your problem domain has more dimensions, and it has to account for the collective experience of users with different needs.

Do you even need a matchmaker?

Ok, hear me out: most multiplayer games need a matchmaker, but there are cases where your efforts are best focused on better onboarding and gameplay. Some games may not even require true matchmaking:

- Party-based or couch co-op games
- Drop-in, drop-out kind of games, where game sessions are flexible

- Games where progression or skill gaps don’t have strong impact on the collective experience

When we first released ARMAJET, our mobile 4v4 PvP shooter, in closed Alpha, we knew we wanted to grow the game starting with a private invite system. We knew it would not provide enough liquidity to justify a true and fair matchmaking system.

So we didn’t make one. We created a system that allowed players to instantly join pre-generated rooms with AI, and people could **drop-in and drop-out** as they pleased. They could also **park the room** and continue playing with newly discovered friends match after match.

The result was that our audience really praised how **fast and easy** it was to join a game with no wait, and the **social experience was incredibly strong**, leading to very high retention rates among our testers.

Had we created a sophisticated matchmaking system during early access, it would’ve gated much of the experience in the interest of fairness, which wasn’t a priority at the time.

Instead, we made sure to **allow anyone to create custom matches** and invite whoever they pleased, which filled the need for experimentation, fairness and discovery of what the game could really become.

But ARMAJET is a twitch-based shooter where latency and skill-matching are central to competition, so we knew we would ultimately need the real deal for the open beta.

The matchmaking variables

This is applicable to so many things…

As the audience grows, so does the appetite for competition. Since most of our growth was on mobile, it was extremely important to **balance the experience for wait times**, especially since most matches ran for less than 5 minutes.

Today, our matchmaker takes into account quite a few variables. These are primary ones:

- Party size
- Criteria selected (game modes, number of players, etc.)
- Device type (desktop, mobile)
- Input system (controller, mouse+keyboard, touch)

- Region latencies (pings)
- Skill rating
- Win/loss history

Once a request for a match is made, the Matcher API creates a **new ticket** in a document store for tracking, and spins up a new process. These processes tick each second to **run queries against other tickets**, and group tickets together based on general configurations and the criteria of the request being made.

New, independent processes **allow matchmaking to scale** horizontally to as many pods as needed, while running separate, context-aware heuristics that are only pertinent to that one ticket. If the user decides to cancel matchmaking, the process is killed and the ticket removed. If a match is considered made, the process locks the eligible tickets and requisitions a new server.

At a small-medium scale, the system would work file just as a single process staggering queries, but a glitch in the system could invalidate large amounts of matches, which is less likely to happen in the independent process scenario.

When tickets are compared, some variables are weighted more than others, and we generate a **score** to determine the affinity, like a dating service.

As logging events fires off, we are able to construct a **top down view** of what the status of the matchmaking process looks like, building graphs to inform of how and why players are being grouped together. Without some form of visual feedback, it would be very challenging to determine if the matching algorithm functioned as expected. It also gives us insight into

how much a small configuration change may affect matching, enabling quick iteration.

Cross-platform, cross-region

Some players hate waiting, others would rather wait for a better match.

Some players hate bots, others enjoy slaughtering predictable AI.

Some players hate steamrolling noobs, others live for it.

I could go on and on, but the main variable here it **time**. It's not a mystery that the more time you have to run matchmaking, the better a match will be. But there are limits, and especially on mobile, players have no interest in waiting 30 seconds for a match that lasts less than 5 minutes.

Depending on the active CCUs across the globe, our matchmaker adjusts itself to provide the best experience possible, with time being the core measure. With low traffic, we may prioritize sending players to servers sprinkling in some AI. Or if their latency is low enough, we may have 2 **regions cross-play each other** in the interest of getting more real players together to have fun.

Because we treat regions just like any other variable, it's trivial to adjust thresholds based on current conditions and allow the matchmaking to do its magic. The hard part is finding the fluctuations that don't break the game.

This is not ideal for all games, and we tested this at length. We've designed ARMAJET's gameplay design, input systems parity and latency tolerance

with as much care as we could, to allow players on any device or location to play fair.

We've been able to match EU and NA with success, but this tends to happen less as of late. Same goes for matching PC with mobile. We try and keep platforms and regions playing with their corresponding peers, but when the matching isn't ideal at certain times of day, especially for a growing indie game, **cross-platform/cross-regional play** has been an incredibly useful tool to **maintain social activity** and make the game feel alive.

Players are really smart and they have plenty of choices for online games, so keeping them engaged is worth tweaking the matchmaker for.

Evolving Matchmaking

The way we see matchmaking evolving is by learning from its mistakes (drop *Machine Learning* keyword here).

We already use win/loss history as a means of **artificially inflating or deflating skill rating**. Consecutive losses will more likely land you playing on a powerful team or against one that's probably going to lose, with consecutive wins doing the opposite, and it's been an industry standard for many years across AAA games.

The next step for us is to **analyze historical churn rates** (players that stop playing the game) and predict churn based on the variables sent in to the matchmaker. The ultimate goal is to provide the best experience possible, which isn't one that's constant, but one that has its **ebbs and flows** based on what a player persona (data profile, if you will) is best

suited for, **without compromising the fairness** of the matchmaking process and personal progression.

It's not an easy task, but one that every game developer should strive for. Players want to feel the rush of winning, losing to someone more skilled, learning from their mistakes, and accelerating once again. That's how you master a game — that's what feels good.

Matchmaking is a big topic that we just scratched the surface of. Whether you build a system in-house or rely on the many off the shelf solutions (see the infrastructure article for some examples), the goal is to customize it to your needs, which vary depending on the stage of your game, the platform that's being played on, and the audience you are targeting.

Don't worry, your players will tell you if you've done a good job. Or at least they won't complain as much if you did.