

GDC

Stop Killing Our Servers!

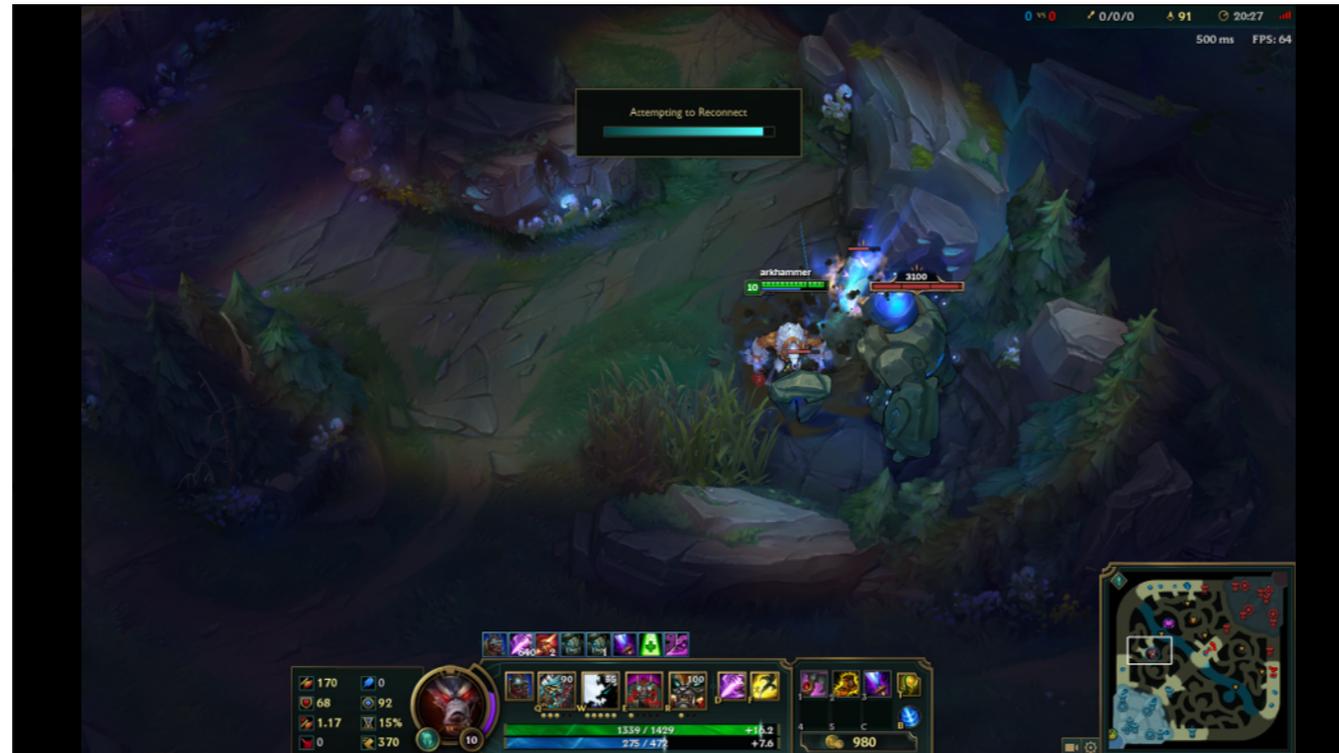
Sela Davis
Senior Software Engineer, VREAL

Jennie Lees
Senior Software Engineer, Riot Games

GAME DEVELOPERS CONFERENCE | FEB 27-MAR 3, 2017 | EXPO: MAR 1-3, 2017 #GDC17

UBM

JENNIE, THEN SELA, THEN BACK TO JENNIE FOR INTRO
Introductions
Noisemakers, Evals, Q&A



JENNIE

How do you deal with servers going down and creating a terrible player experience?
This is my least favourite screen, and the reason I turn up to work every day.



These are the four principles that we've determined are critical to the development of a client that talks to a server. These should be thought about early and often. In this deck, we'll discuss eight areas within these four principles and provide some stories to see how they can be useful in action.

Understand your data model



SELA

Build your data model in a way that scales gracefully. Batch transactions. Understand when your requests cause the server to read or write from your data store, and what should be stored on the client vs. the server. Consider the impact of concurrent clients. Know what needs to be secured on the server vs. the client.

Batching

Achievement: 1000 Enemies Killed

```
if (enemyKilled)
{
    server.UpdateProgress(1000_ENEMIES, 1);
}
```

There is a lot of overhead across multiple requests, especially on the server side. Batch when you can (do you really need to send each of these 1000 up, or could you send updates when your combo ends).

This client in particular is performing the action constantly. Imagine the effect if the player is firing a gatling gun.

Other things to think about: HTTP connections – may have a limit on the client, slowing down what you can do. Your server may be able to batch-fetch data for you, reducing the number of servers needed overall.

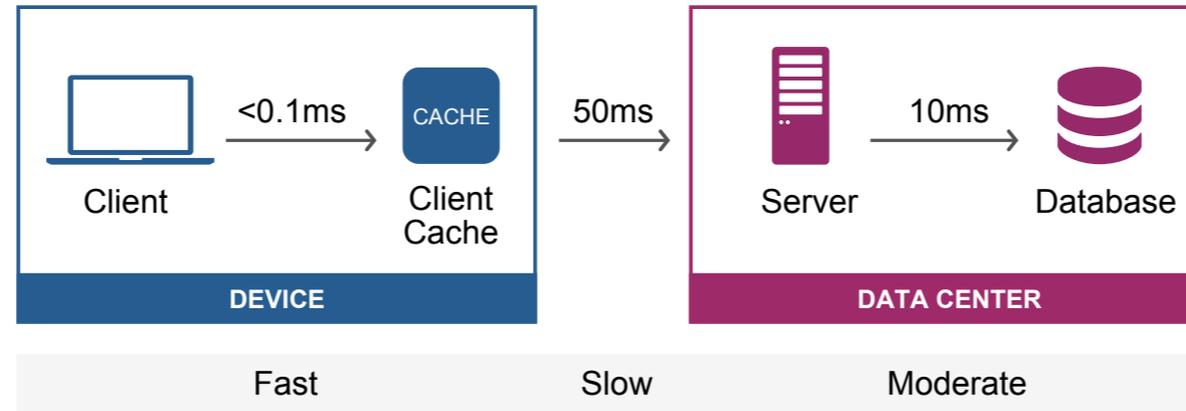
```
while (enemiesAreBeingKilled)
{
    enemiesKilled++;
}

if (enemiesKilled > 0 && !achAwarded)
{
    server.UpdateProgress(1000_ENEMIES,
enemiesKilled);
}
```

This is a much kinder client. Now we have batching and we don't send a network request if we don't need to. This client should, in theory, send no more than 1000 requests - and should never send large bursts of requests.

Any further optimizations are left as an exercise for the viewer.

Data Storage



These are rough, napkin-math numbers. Based on what your server is doing, the way you are retrieving data, etc, this may be faster or slower. Remember that a single frame is 16ms if you are targeting 60fps.

Understand how expensive a call is on the server. Are they fetching data from a cache, from a database, or from a remote service? How fast is that?

Cache it locally if it makes sense to do so rather than requesting it frequently. Some things change rarely, and in some cases infrequently-updated things can just be updated at login if you have authentication.

In some cases, it is okay to show out of date data for other users, but not for the player.

Storytime: Too Many Writes



Smash Squad data model. The server team became aware of a large amount of writes on the database during soft launch. Operations that seemed trivial on the client were actually expensive (dozens of writes per request) on the server. Average number of writes per second per user was 3. Client engineers manipulated the data frequently, because they didn't know the effects, and this was realized too late to change before launch. More hardware was needed.

Image credit: smashsquad.com

Design for concurrency



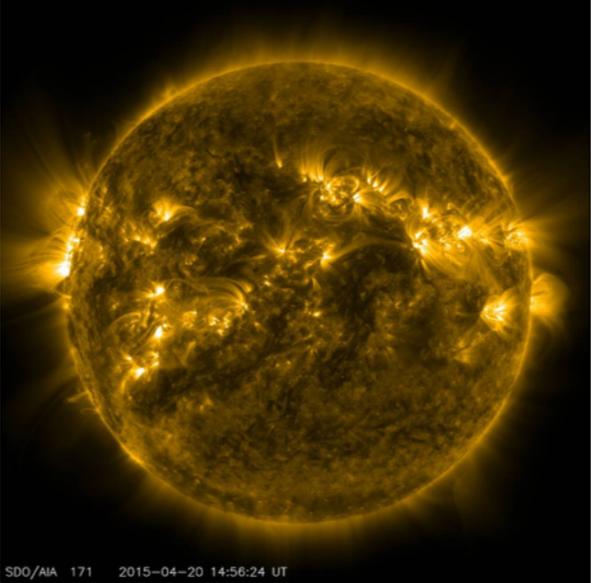
JENNIE

Concurrent requests can impact your data model, but can also impact the work your server can do. Understand the limits of your servers and build the client calling patterns appropriately. Requests can be queued both on the client and the server.

GDC GAME DEVELOPERS CONFERENCE® | FEB 27-MAR 3, 2017 | EXPO: MAR 1-3, 2017 #GDC17

Hot Spots

- Resource contention
- Where are your hot spots?



SDO/AIA 171 2015-04-20 14:56:24 UT

UBM

How does your server deal with resource contention?

Be careful about generating hot spots – for example, if every user online is attempting to write to the same data field on the server, there will either be optimistic concurrency concerns or queuing. Even if everything goes smoothly, that particular server may become overloaded.

Going back to data model - how does it handle source of truth? CAP mention?

Image credit: NASA. http://www.nasa.gov/sites/default/files/styles/full_width_feature/public/thumbnails/image/20150420_active_regions_171.jpg

Queueing

- Fire and forget requests
- Lose control, gain flexibility
- When a resource queue fills up, can your client handle it?



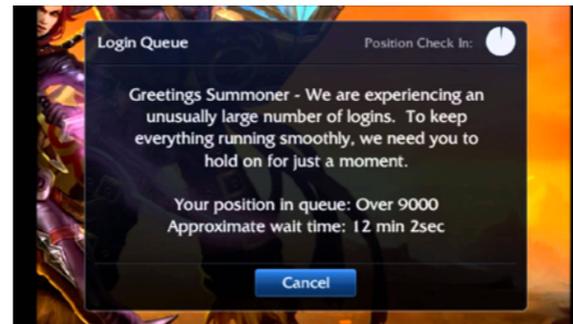
Image credit: German Federal Archives, originally found at https://commons.wikimedia.org/wiki/File:Bundesarchiv_B_145_Bild-F079012-0030,_Berlin,_Michael_Jackson-Konzert,_Wartende.jpg, under <https://creativecommons.org/licenses/by-sa/3.0/de/deed.en>

Naturally, queuing slows down requests

Individual resource may have their own queues while other requests are fine – or the entire system may begin to queue up

Queueing can cause cascading failures that take down servers, or even cause incomplete requests to occur and incomplete data to be persisted.

Storytime: Login Queues



League of Legends login and message queues.

When we have a server crash we have a buffered login queue to ensure that we don't spike the auth service. Because the first thing a player does when things aren't working right is exit the game and try to log back in. By putting a login queue in front of our entry system, we allow players to enter the backend at a predictable rate, avoiding hot spots in our accounts database. Of course there are downsides too - the login queue spiking is often the first sign of trouble, and it can take some time to diagnose the real issue.

Image credit https://www.youtube.com/watch?v=YCL9_MppGrU

Embrace asynchronicity



SELA

Don't poll to keep data up to date for constant-time access. Instead, acknowledge latency and background tasks. Make calls on-demand before you need the data - better to throw it away than to poll constantly. Understand how and when to cache data on the client.

Do. Not. Poll.

- Keeping data fresh is awesome
- ...but not so awesome for your servers
- Make requests before you need them



Don't do it! This adds up to constant load and is rough on your servers.

Rather than poll, find alternatives. Make requests before you need them. You may not be able to predict when you will get the data back, but in most cases you know the data will be needed "soon".

There are lots of ways to be informed when a request completes. Javascript popularized callbacks and promises. C# has events.

Caching

- Constant time access!
- Do you need the data to be 100% fresh at all times? (No.)



Rather than requesting frequently, cache data locally unless it needs to be immediately up to date. Once cached, you can access it in constant time.

While your user's remote data should probably always be up to date, other users' data can be slightly stale and players won't notice or care.

When caching, remember that you can use the old value from your local cache and make the request to update it for next time. At that point, it will be fresh on the next fetch.

Promotional image from Amazon (<https://www.amazon.com/Fresh-Prince-Bel-Air-Complete-Variou/dp/B00B5AAWD4>)

“Cache your data and keep it fresh!”

C# Pseudocode: cache that refreshes itself in the background

```
int? cachedOrEmptyValue = null;
DateTime nextFetch = DateTime.MinValue
public int GetCachedValue() {
    if (cachedOrEmptyValue.HasValue()) {
        int resultValue = cachedOrEmptyValue;
        // check included here for clarity
        if (DateTime.UtcNow > nextFetch) {

            // update on a background thread and update fetch time on success
            UpdateCachedValueAsync();
            return resultValue;
        }

        // wait on the result of the call if the value was never cached. time out as necessary.
        return UpdateCachedValueAsync().Result;
    }
}
```

Storytime: The saddest graph :(



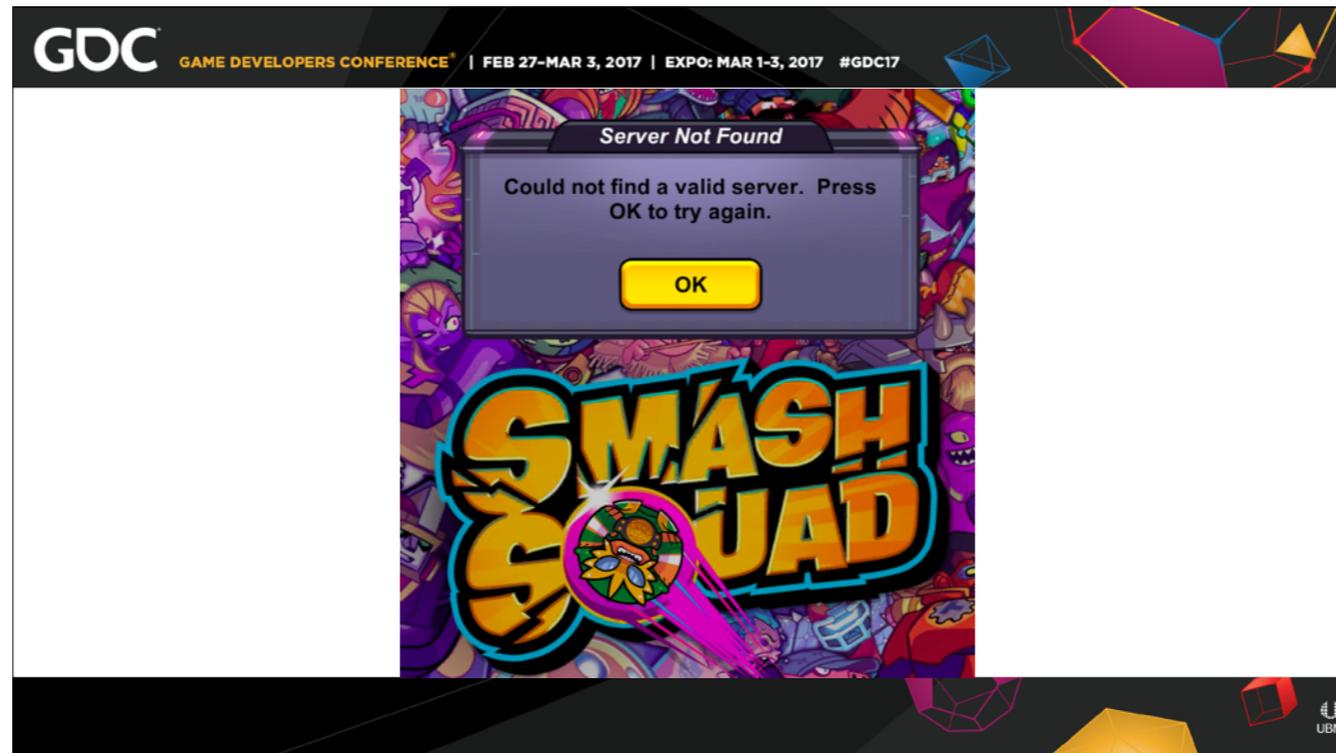
This graph is the saddest graph. It represents a service's steady state that was suddenly changed when a game came online at a midnight launch. This client was polling to keep data fresh in memory, which majorly increased the load on the servers. Even worse - the data appeared to be polled when unneeded so that it was accessible in constant time in the game. Again, more hardware was necessary, when this could have been realized before the launch.

Things will break!



JENNIE

Networks are frequently inaccessible. Data might not make it back. Data might take a long time to retrieve. Fail early and fail often. Don't crash on failure. Expect it, test for it, and manage both partial and complete failure in your systems. Provide a great, degraded experience when failure occurs.



Networks go down all the time!

Be prepared to deal with slow requests.

Fail early when you can.

Don't hammer the servers to retry! Back off.

Different services have different request SLAs - talk to your service team to understand.

Even if things are accessible, they may be slow. Slow as in "multiple seconds". Game developers often think in frames, but you should think in terms of "at some point".

Fail early if you can. Depending on the operation, you can often bail within a few seconds rather than waiting 60-120. Consult with your server team on reasonable values – they should be able to discuss the average case and worst case they expect.

For example, the Xbox 360 account recovery process can legitimately take 120 seconds. But fetching a single string from S3 should not take more than a second.

Even if you never have an incident in the data center, our friends at Lizard Squad / etc can cause similar problems! Example: DDOS on DNS, 21 Oct 2016

Don't wait for real failures

- These scenarios are 100% testable!
- Add fake latency to mock network calls
- Test with intentional server failures
- Include failure in your design and acceptance criteria

You may want to add fake latency into your fake, local network calls to realize the behavior before it is too late. Likewise, you want to intentionally fail requests from the server in testing scenarios to track down these issues early.

GDC GAME DEVELOPERS CONFERENCE | FEB 27-MAR 3, 2017 | EXPO: MAR 1-3, 2017 #GDC17

Unreliable Data

- What happens if you only get partial data - or it's corrupted?
- What happens if you get nothing back?



UBM

Do what is best for the user experience. “The server is inaccessible” is often not best.

If you can provide partial information to the user without breaking the experience, a degraded experience is often preferable to a broken one. The Netflix client is a great example. <http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html>, among others

So you can still watch Buffy even if the recommendation engine isn't working.

Storytime: Microservices



JENNIE

The League of Legends backend is made up of separate microservices, so we don't have to kick players out of games if a non-crucial part of the system goes down, like the Store. By having the ability to check the status of all our backend components, we allow the Network Operations Center to disable specific features and make a call on whether the game itself has to be disabled.

Image source: Riot Games (Brasil Porofolio)

Load test



SELA

Load testing is important, even from the client. As the client, you have more insight into the mix of traffic than anybody else. It's not just the server team's job.

Ahead of the game

- Run load tests early and often
- Use different user scenarios to build realistic traffic
- Build scenarios per feature, not across the board

Run load tests early and often. Use your different user scenarios to try and build as realistic a mix of synthetic traffic as you can. Build scenarios where a feature becomes suddenly more popular.

Simulate the worst case, but build for the expected.

Rush-hour network traffic

- Peak times, low times, etc
- Consider traffic at:
 - Tutorials
 - Single-player, per level
 - Multiplayer
 - After a major disconnect



Consider how different your traffic looks at different times. For example, tutorials are usually only performed at the beginning of the game. If you have a popular title with a midnight launch, you will have a spike of players performing tutorial actions all at once. If your tutorial is expensive on the server, you risk overloading your servers.

Image source: https://commons.wikimedia.org/wiki/File:Miami_traffic_jam_I-95_North_rush_hour.jpg. No changes. License: <https://creativecommons.org/licenses/by-sa/4.0/deed.en>

Simulate the worst case

- Random distribution can be unfavorable
- If you can handle rough times, you can handle optimal times

Remember to use the right mix! If you evenly distribute all of your requests, you may find issues where you don't have any - or may hide issues that only appear under load. If you can handle your worst case scenarios under accurate load, you will likely be fine at launch (unless your traffic estimates are wrong).

Storytime: Can we crash it?



In mid-2016 I (Sela) came across one of the most challenging issues of my career: managed heap corruption in a process with two managed languages (Java and C#) as well as unmanaged code. We were also near launch, and needed to resolve this ASAP. I worked with a client engineer who was working on load testing, and we put together a reasonable set of operations that simulated realistic user input (as opposed to sending single requests repeatedly, which caused hot spots in our system). Load testing helped us out in a number of ways: we were able to finally reproduce the issue, and we were able to tune numbers to start to narrow down the cause. Because it was only happening once every couple days in production, it let us test theories and new server builds quickly and without deployments. It also helped us to identify other areas where we might have problems in the future and create more realistic synthetic traffic.

Image credit: <http://gunshowcomic.com/648>

Profile, measure, repeat



JENNIE Profiling

Understand how your client performs in different user scenarios. For instance, if a certain feature causes extra strain on the server because it causes more requests to occur, you may not want to build a live event around that feature without additional server capacity.

Measure

Track how long requests take from end to end (client -> server -> client). Track worst cases and average cases. Understand how your game performs in poor network conditions.

Mobile: low signal

Non-mobile: Slow internet, busy internet, etc. What if somebody else in the house is streaming Netflix?

Get used to looking at graphs early so you can understand them when you look in an emergency.

Profiling

- How does client behaviour impact the server?
- How does server behaviour impact the client?
- Get used to looking at graphs!

Why do you even care? -> Understand how your client performs in different user scenarios. For instance, if a certain feature causes extra strain on the server because it causes more requests to occur, you may not want to build a live event around that feature without additional server capacity.

Understand client behavior in different scenarios

Understand the impact of client behavior on the server

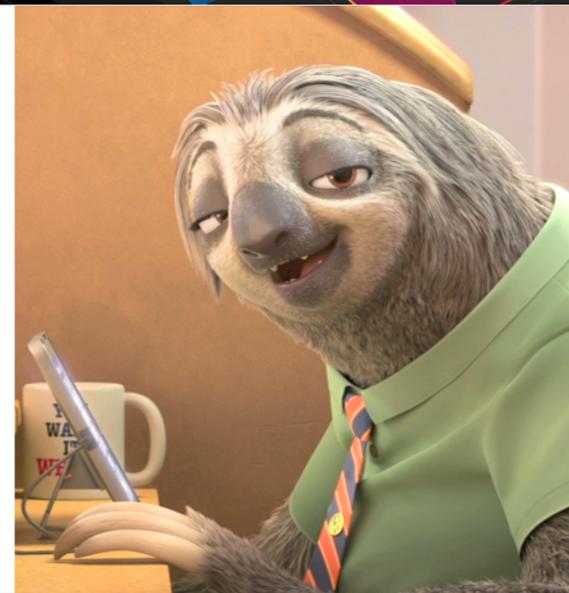
Helpful when designing live events!

Understand the impact of server behavior on the client

Get used to looking at graphs

Measuring

- Track requests end-to-end
- Know the worst case (p95%+)
- Know the average case



Measure

Track how long requests take from end to end (client -> server -> client). Track worst cases and average cases. Understand how your game performs in poor network conditions.

Measuring

- Understand performance in poor network conditions
- Poor or dropped cell signal
- Provider outage
- Roommates streaming Netflix



You won't always get great network coverage. In 2010 Sela lived in the center of San Francisco - arguably one of the most connected cities - and had frequent outages and interrupted connection. Failure will happen.

Image credit: <https://www.flickr.com/photos/dbrulz/945284001>. License: <https://creativecommons.org/licenses/by-sa/2.0/>

Storytime: Detective work



JENNIE

A non-games example: by monitoring the response times for a search API, we found that clients were making very unusual calls and managed to stop them before they brought the entire search database to a locking halt. As we were a startup at the time, we didn't have elaborate monitoring or alerting, and the way we caught this issue was humans looking at the graphs during standup...

Build for debugging



SELA

“Build for debugging.” Log requests (and results) on the client. Build specific metrics and telemetry into your client/server interactions. Build dashboards to track both your client and server metrics, and build team habits out of looking at the data. Use the server team’s metrics to identify client defects.

GDC GAME DEVELOPERS CONFERENCE® | FEB 27-MAR 3, 2017 | EXPO: MAR 1-3, 2017 #GDC17

Logging

- Log requests and results on the client
- Include information to correlate with servers



413
Request Entity Too Large

UBM

Log requests and results on the client in addition to the server. Include enough information to track this request on the server.

Remember that your server team's logs may be handling thousands of lines per second, so you need to be immensely searchable.

Work with the server / logging team to add rich metadata to your logs!

Your requests and results can be compared against what the server thinks it sent back.

Tracking error codes can be immensely useful, especially when the request is invalid due to the payload.

Example: 413: request too big. You might get this while testing a new feature out. It is a client problem, not a server problem, but may need resolution on the server.

Image credit: <https://www.flickr.com/photos/girliemac/6508023747>. No changes. License: <https://creativecommons.org/licenses/by/2.0/>

Logging

- Expose logs to client and server teams
 - Upload to 3rd parties or your server
- Ensure this data is searchable so server teams have client context
- Use crash reporting tools

Consider how to expose this data to both client and server teams. Does this data get uploaded to your servers in realtime? Do you have tools that can pull the relevant data out of the client? How does this interact with the game packets?

Consider how searchable your data is. If the server team asks you about requestId 12345678-1234-1234-1234-123456789012, can you find it easily? They probably can if you ask them for the same.

Use crash reporting tools. Make sure the data gets aggregated in a way that makes it easy to spot problems, and include in crash reports enough information to pull up relevant server information. Perhaps a session id in addition to request ids so you can replay the user's game session. (Example failure this may catch: a json deserialization failure, if the server gave invalid json)

Storytime: 200 NOT OK



AKA “the server lied”. The server can have bugs too.

A client game spoke to a service that didn't really explain failure well. The client team noticed odd behavior and results and reached out to the server team. The server team was able to track a request to the database, where it turned out there was a communication failure. The server handed back a 200 and partial results instead of a 500. Results that were not received were indistinguishable from “no data”, and the client game happily used that data and persisted its results to the server. Both the server and the client needed to change to fix this issue.

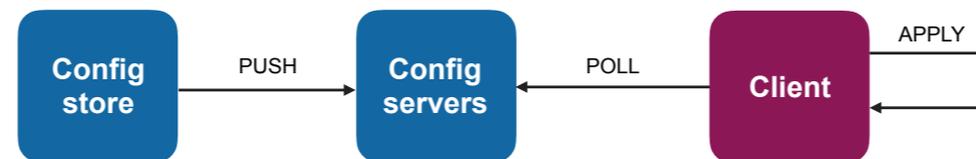
Design to deploy



JENNIE

Dynamic Configuration

- Clients need configuration too!
- Update and tune in real-time



Clients need configuration too.

This is not just “config received from the server”; this is “how do I communicate with my server”.

Allow real time updating of variables from server, i.e. a feature flag on steroids; can tune gameplay behaviour and make realtime balance changes without requiring deploys.

What are my endpoints? What environment?

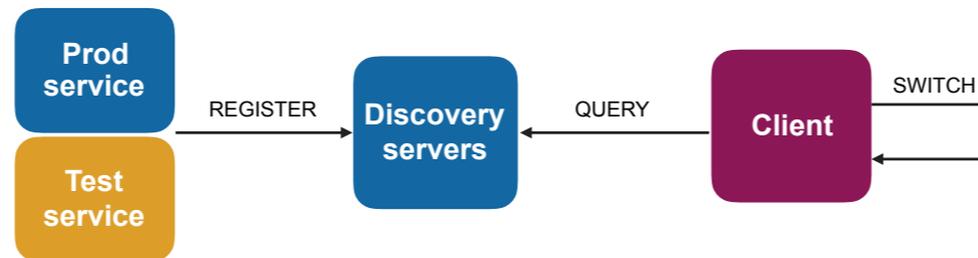
Allow real-time config update from servers.

Feature flags on steroids.

Tune anything you want without requiring deployments, like sales or design changes.

Service discovery

- Allows dynamic switching on the server side
- Switch between test and prod in the same client



Example: consul.io

Use discovery to find services, instead of hard coding - easy to deploy, turn things on and off at will, understand what version you are talking to.

This can be used on the server side behind load balancers and in larger architectures is more secure this way.

Switch between test and prod, no matter what build it is. It can be helpful to have a debug build running in prod – especially if you can just point a test device rather than having to get a new build.

zookeeper, etcd, consul, handrolled system designed for games probably

Feature flags save the day

- “Chicken switches” or “kill bits”
- Dark launch ahead of time
- Disable features live



aka “chicken switches” or “kill bits”.

Allow features to be disabled remotely and to work independently.

If a feature doesn’t scale, you can save your servers. If a feature fails, you can save your clients.

Allow you to deploy the latest code everywhere and choose when you turn something on, rather than risk service downtime when you want to launch.

API Versioning

- Switch all clients over to a new version of the API at once
- Gradual rollouts
- Graceful rollbacks



API Versioning

Talk to different versions of an API to switch all clients over at a preconfigured time. Or switch a percentage over at a time until everybody is migrated.

Storytime: Replays Dark Launch



Dark launching replays for LoL - image Riot Games

Replays is a feature that LoL players have wanted for a long time, so we couldn't screw it up. We also wanted to get the feature to them in a timely manner, and ensure that we didn't have huge failures on launch to disappoint our players and ruin our reputation ;)

We dark launched replays by having the server side components rolled out weeks in advance and gradually turning on traffic from different regions. The client integration went out in a patch 2 weeks before launch and we used dynamic configuration (aka micropatching) to turn it on. Our backend was distributed (there are hundreds of replay services running) so we used load balancing + service discovery to ensure the clients always had a server they could talk to.

Understand your data model

Design for concurrency

Embrace asynchronicity

Things will break!

Load test

Profile, measure, repeat

Build for debugging

Design to deploy



SELA

[Wrap up slide]

We discussed four principles and gave eight topics to consider, but if you take any one thing away: *talk to your server team*. You are all on the same side.

Thank you!

@jennielees

@sela_davis

