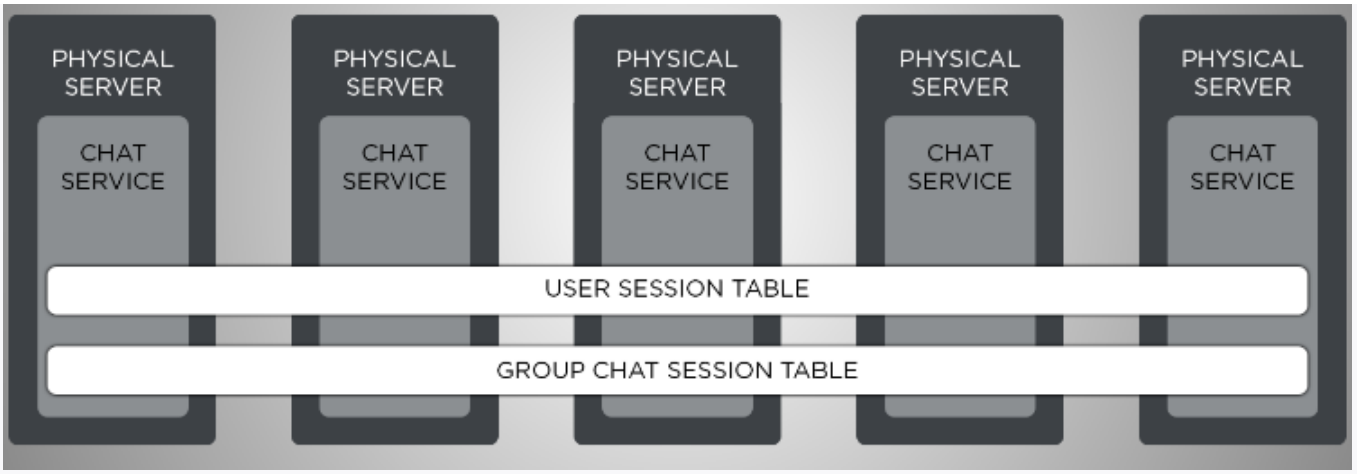


CHAT SERVICE ARCHITECTURE: SERVERS

League of Legends players collectively send millions of messages every day. They're asking friends to duo-queue, suggesting a team comp on the champ select screen, and thanking opponents for a good game. On July 21st of this year (I picked a day at random), players forged 1.7 million new friendships in the game—that's a lot of love! And each time players send a message they trigger a number of operations on the back-end technology that powers Riot chat.

In the previous episode of this series on chat, I discussed the protocol we chose to communicate between client and server: XMPP (Extensible Messaging and Presence Protocol). Today I'll dive into the mechanisms in place on the server-side and the architecture of the infrastructure, and I'll discuss the work we've done to ensure that our servers are scalable and robust. Like the last article, I hope it'll be interesting to anyone building out chat features to a distributed client base.

SERVER HARDWARE

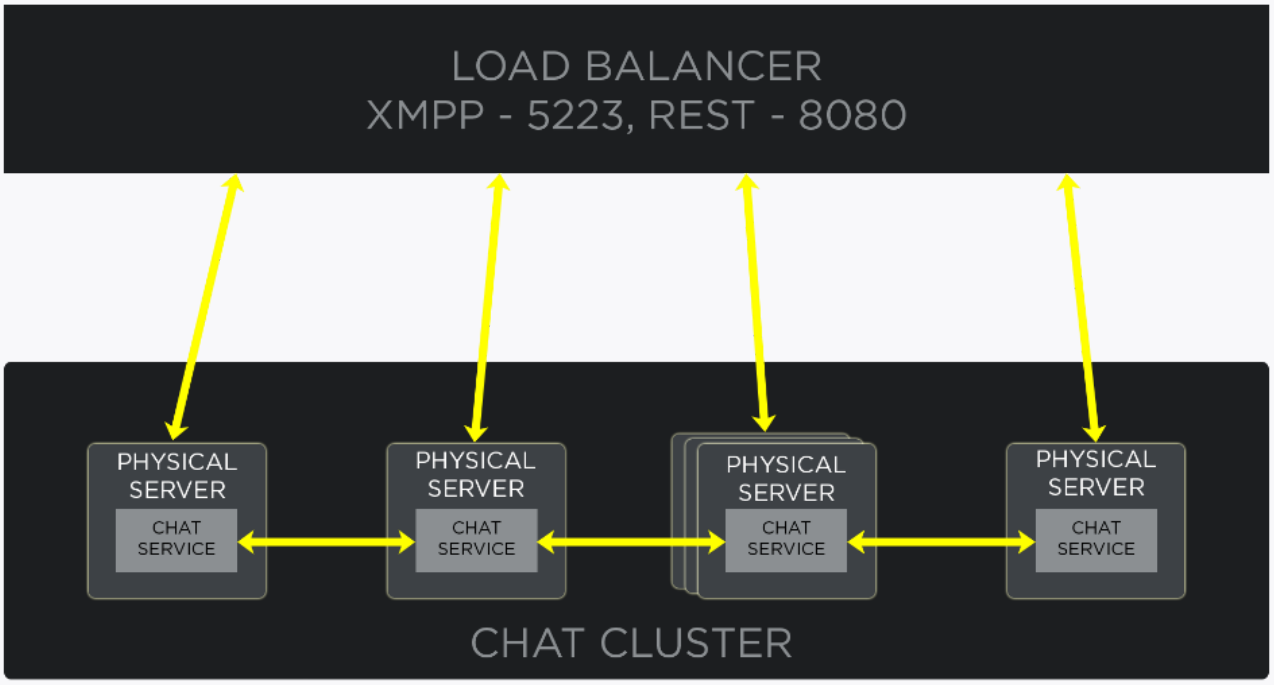
Chat's physical servers maintain a number of responsibilities to ensure that the service remains available to players. These servers manage individual players' chat sessions, and also apply all necessary stability and security validations like privacy settings, traffic rate limiting, metrics collection, and logging.

We deploy chat services on a per-region basis (we call them 'shards'), meaning each *League of Legends* region has its own chat cluster that provides features only to players assigned to that shard. As a result players can't communicate between shards, and chat servers can't use data from other regions. For example, North America servers can't directly communicate with Europe West servers.

Each shard maintains a single chat cluster comprised of a number of heterogeneous physical servers running the same server software. Hardware specifications vary between regions for a number of reasons such as capacity requirements, age of equipment, and hardware availability: our newest servers run on modern 24-core CPUs with 196 GB of memory and solid-state drives, while older ones use CPUs with 24 GB of memory and spinning disks.

Within a chat cluster, every node is fully autonomous and replaceable, allowing for easier maintenance while increasing the system's overall fault tolerance. The number of nodes in a cluster ranges from six to twelve machines. Although we could run chat on fewer machines in every region, we prefer to maintain comfortable headroom for fault tolerance and to accommodate future growth. If upgrades require a shutdown, for example, we can shut down half of the nodes in a single cluster without interrupting service to players.

The following is a simplified diagram showing our chat cluster architecture:



In the past we've had numerous incidents related to hardware or network failures that forced us to shut down individual servers for a few days. However, since the system lacks a single point of failure, the service continued without disruption. Also, we implemented logic in the client that helped players connected to the shut-down servers automatically regain their connections.

IMPLEMENTATION

We write Riot chat servers primarily in Erlang (check out this video if you're curious about the language), although we use C for bindings to certain lower-level operations such as XML parsing, SSL handling, and string manipulation. 10% of our chat server codebase is written in C while 90% is pure Erlang (ignoring external libraries and the Erlang VM itself).

Before developing the components of the server written in C, we spent a lot of time profiling and optimizing the existing Erlang codebase. We tried to find potential concurrency and efficiency bottlenecks using a full battery of existing tools, operating at different abstraction levels:

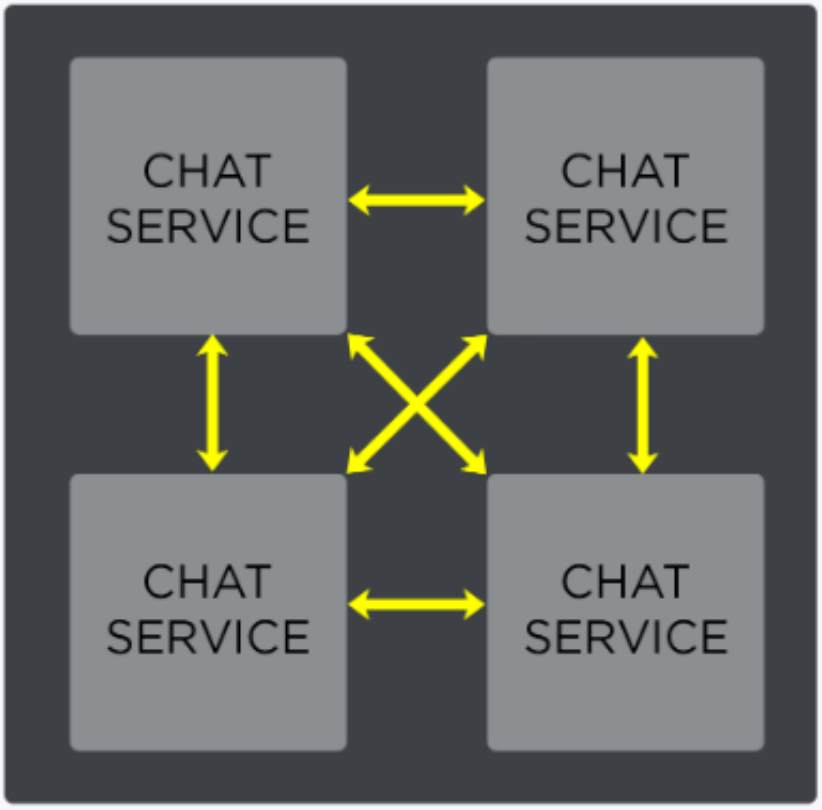
- For simple call counts we use cprof. Despite cprof's extremely simple architecture it provides the ability to validate that we're making the correct calls at the correct times.
- For a more detailed analysis we run fprof. Unfortunately, fprof has a much bigger impact on the

tested server and can't be used during a full load test—however, it gives much better insight into how and when calls are made. This includes the time spent by a function during just its own execution ("own time"), the total time spent including all called functions ("accumulated time"), and the grouping of results by process. All of this helps to identify the most CPU intensive parts of the system.

- When detecting concurrency bottlenecks, percept and lcnt are your friends. These tools helped us tremendously with identifying opportunities to parallelize data processing in order to utilize all available cores.
- On the OS level we fall back to traditional analysis tools such as mpstat, vmstat, iostat, perf, and a number of /proc filesystem files.

We found that most of the precious CPU cycles and memory allocations happened while processing text and data streams. Since Erlang terms are by definition immutable, performing any sort of intensive string manipulation would be costly no matter how much we optimize our code. Instead, we rewrote the heaviest parts of the string manipulation system in C and compiled parts of the standard library with HiPE—performance improved by over 60% for CPU utilization. For memory usage, we observed a drop from over 150kB allocated per player session, to 25kB for each session process.

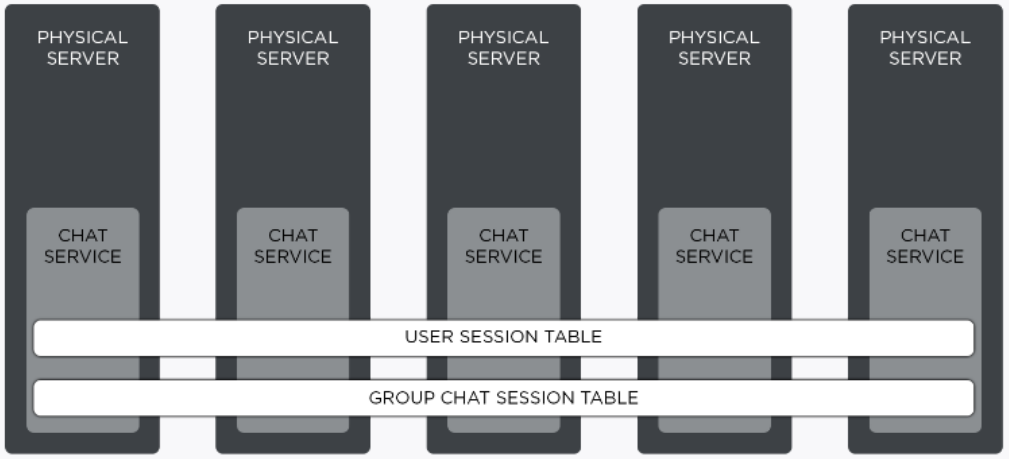
We try to follow best practices and Erlang/OTP design principles, so Riot chat servers form a fully connected cluster where each Erlang VM maintains a single persistent TCP connection to each other chat server in the cluster. The server uses these connections, which speak the Erlang distributed protocol, for internal communication. Thus far, despite exhaustive testing, we have been unable to find any bottlenecks related to this communication model.



I mentioned earlier that each individual server in a chat cluster is a fully independent unit—it can operate without any other cluster members, and we can service it at any time. In

order to provide for sufficient scalability, we designed the system such that nodes share as little data as possible. As a result, servers in a chat cluster share only a few internal key-value tables that are fully replicated in-memory and communicated via a highly optimized Erlang-distributed store called Mnesia. (If you'd like to learn more about Mnesia, I urge you to check out material on erlang.org and learnyousomeerlang.com.) These tables map player or group chat identifiers (called JIDs in XMPP) to Erlang handler processes that maintain context-specific data.

For individual player chat, this data might include the connection socket, friends list, and rate limiters; for group chats, room roster and chat history. Every time there's a need to route any kind of information between player or group chat processes, the server consults these tables in order to resolve their session handlers using JIDs. Since each server holds full replicas of session tables, all reads can be executed locally, reducing the overall routing latency.



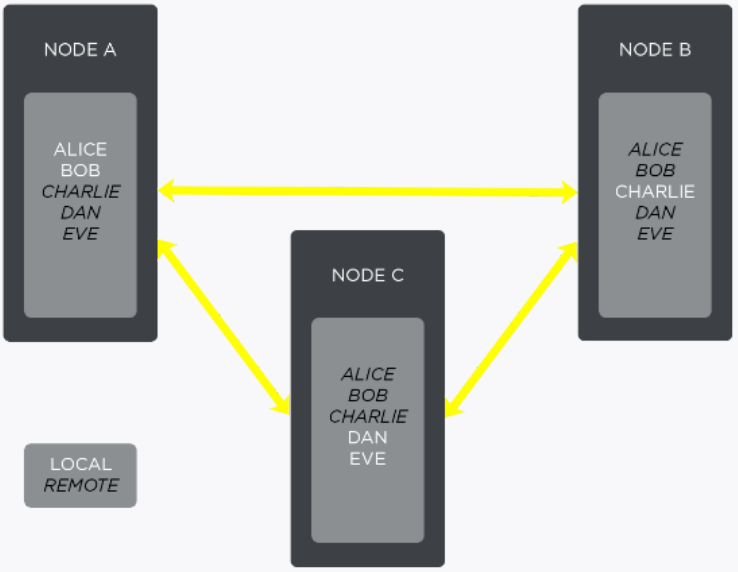
The downside of this approach is that each update applied to terms stored in these tables must be replicated to all other connected chat servers. To our knowledge these tables are the only blocker from being able to linearly scale a chat cluster—however, our load tests show that we can grow chat clusters to at least 30 servers without seeing a performance degradation.

In order to keep players' session data and group chat data in sync in the event of network partitions or server problems, each chat server runs an Erlang process subscribed to the VM's internal notifications about cluster topology changes. Whenever an event is generated concerning a chat server going down, the other nodes will remove from their local tables all session entries that were running on the offline node. As soon as that node returns online (usually either after a restart or when the network connectivity restores) it will push its own local state to the other cluster members and download the state of those other members itself. Through this model chat servers play an authoritative role for their

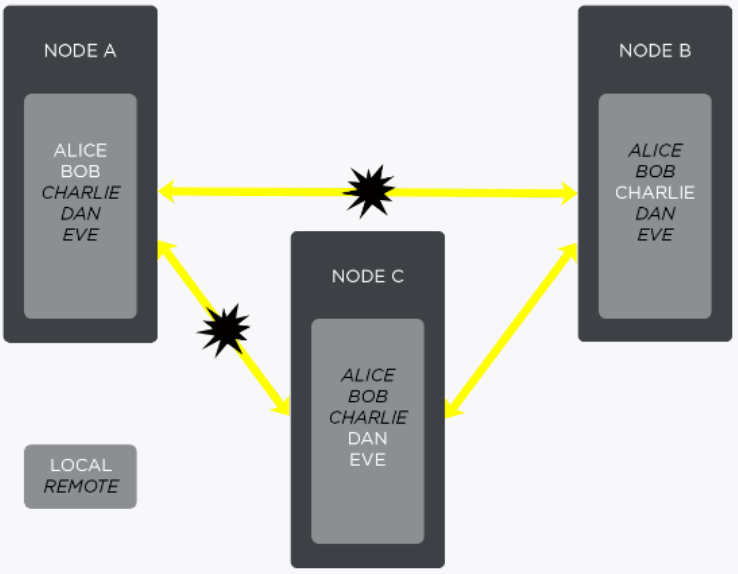
own data, and trust data coming from other chat servers.

To illustrate this design, let's consider the following example:

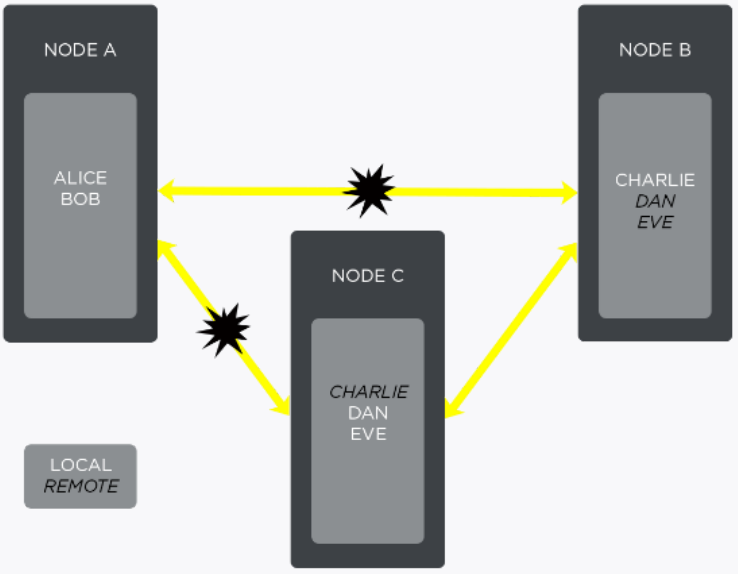
- We have a cluster consisting of 3 chat servers: node A (with players Alice and Bob connected), node B (with player Charlie connected), and node C (with players Dan and Eve connected). All players can talk to each other right now:



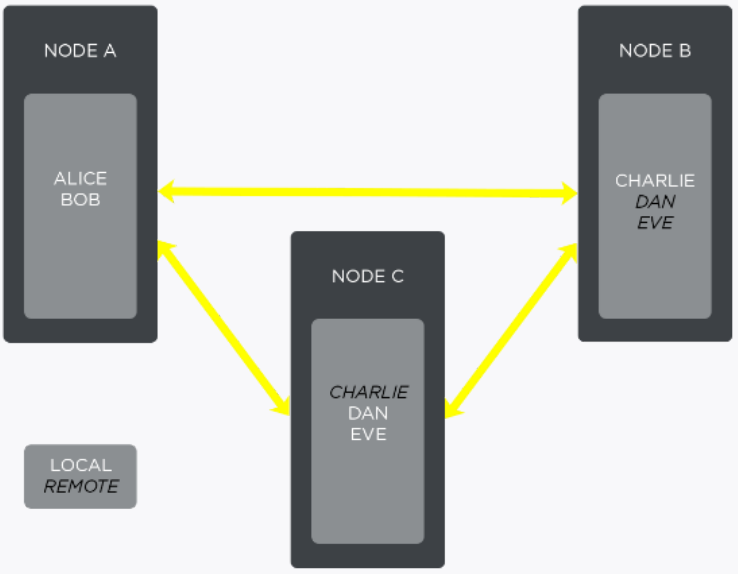
- A network failure cuts off node A from the other servers in the cluster. The next time another machine attempts to communicate with node A, it'll encounter a closed TCP socket and a raised event signaling a change in the cluster topology. This is a classical example of a netsplit. In this case Alice and Bob reside on an island and can't talk to Charlie, Dan, or Eve. Charlie, Dan, and Eve can still talk to each other:



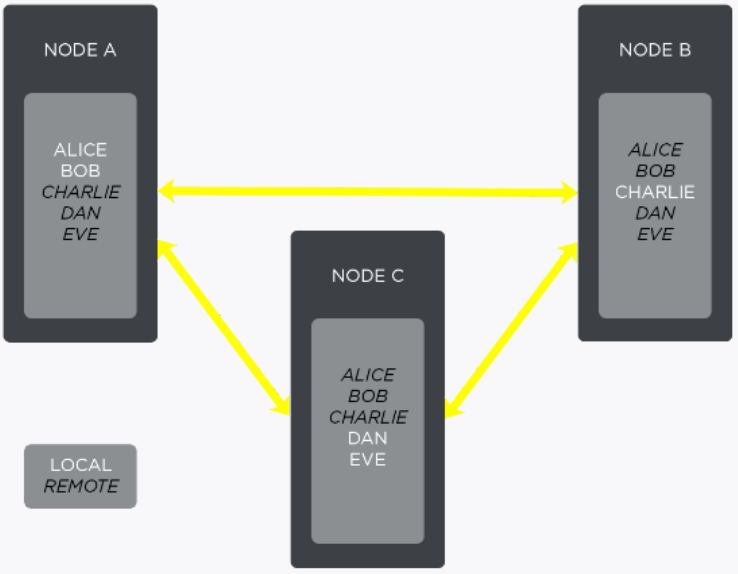
- Now, nodes B and C cannot access player sessions residing on node A. When the Erlang VM delivers the cluster topology change event to subscribed handlers, nodes B and C will drop those player session references. Although Alice and Bob are still connected to the service and can talk to one another, they can't communicate with Charlie, Dan, or Eve:



- After network connectivity is restored, node A reestablishes a TCP connection with nodes B and C. Unfortunately node A does not know of node B and C yet, so Alice and Bob still can't reach Charlie, Dan, or Eve:



- Lastly, the Erlang VM fires the cluster topology change event again, this time, signaling the joining of a server. Node A downloads the session data of nodes B and C to its local Mnesia tables. Nodes B and C do likewise with the data from node A. The cluster is now fully replicated again, allowing for full communication between Alice, Bob, Charlie, Dan, and Eve:



This design allows us to build robust and self-healing infrastructure, removing the need for immediate manual intervention in case of connectivity failures. As a result, the service can automatically recover from network issues and restore its functionality to players.

DATAFLOW

Given that server implementation, let's examine the software processes that provide chat to players. Whenever a player's client connects to the chat service, that client opens a persistent, encrypted (using AES256-SHA cipher) TCP connection to its public XMPP endpoint. The cluster load balancer then selects one of the individual chat servers on the back-end and assigns the player session to that particular box. Currently we use a regular round-robin load-balancing strategy to distribute load evenly across all available servers.

As soon as the connection arrives, the chat server creates a new dedicated Erlang process to handle that player's session. This process (called c2s for "client to server") maintains the player's TCP socket, XML parser instance, list of the player's friends and ignored contacts, last presence data, summoner name, rate limit configuration, and other important details used by the server to provide the player experience. Upon connection, our system immediately requires authentication that validates the player's identity. Here we leverage standard XMPP authentication mechanisms that are compatible with third-party clients.

For further illustration, let's consider Alice and Bob again. They are both Bronze II players with high hopes of getting to Master tier one day. They practice together day and night, honing their skills on Summoner's Rift.

Alice would like to ping Bob and arrange a ranked duo game so that they can finally be promoted to Bronze I. Here's what happens behind the scenes once Alice is successfully authenticated:

- Alice's AIR client sends an XMPP message (carrying "*do you wanna build a duo bot death squad?"* payload) to the chat server over the secure TCP connection maintained in Alice's c2s process.
- Alice's c2s process receives the message, decrypts it, and parses the XML.
- After parsing, the process applies a number of validations on the message including rate limit compliance, spoof checking, and membership tests of the ignore list and friends list.
- After passing these validations, Alice's c2s process looks up Bob's session handler in the internal routing table to validate his availability.
- If Bob is currently unavailable, the message sits in the persistent data store for delivery until he next logs into the game. Depending on the shard this store is either MySQL (for legacy environments) or Riak (for new shards). Data stores will be the topic of my next blog post on chat.
- If, however, Bob is available, the server sends the message to his c2s process using standard Erlang message passing mechanisms.
- When Bob's c2s process receives the message, it applies a number of validation tests similar to those of the sender.
- The process then serializes the message into XML and sends it to Bob's TCP socket.
- Finally, Bob's AIR client receives the message and displays it in the appropriate chat window.

Obviously, upon receiving that question Bob agrees to join forces with Alice, and they carry their team to Bronze I in no time.

INTERNAL INTERFACES

Besides providing chat features directly to players, Riot chat servers also expose a battery of private REST interfaces for internal consumption. Typically, the service uses these REST endpoints to access the social graph (the network of players connected by virtue of being friends).

Examples include:

- For content gifting, such as skins, the service behind the in-game store verifies that the friendship between them is sufficiently mature—this helps to combat gifting from compromised accounts.
- The Team Builder feature uses the *LoL* friendship social graph to build a list of suggested friends-of-friends when creating a team.
- For the leagues service, the system queries the friends list endpoint in order to determine the best league in which to seed a new player, as it prefers placing a player in a league with friends.

As a real-world example, consider the request to get a list of friends of the summoner with ID 13131:

```
> GET /friends/summoner/13131 HTTP/1.1
> …
< HTTP/1.1 200 OK
< server: Cowboy
< connection: keep-alive

[
    {
        "ask": "none",
        "ask_message": "",
        "created_at": "2015-06-30 10:52:26",
        "group": "Work",
        "nick": "Riot Teemo",
        "note": "top laner!",
        "subscription": "both",
        "summoner_id": 112233
    },
  {
        "ask": "none",
        "ask_message": "",
        "created_at": "2015-06-25 11:25:07",
        "group": "Family",
        "nick": "Pentakill Morg",
        "note": "Mom",
        "subscription": "both",
        "summoner_id": 223344
```

```
    },
    {
        "ask": "none",
        "ask_message": "",
        "created_at": "2015-06-17 17:57:17",
        "group": "Work",
        "nick": "Jax Jax Jax",
        "note": "plays only Jax?",
        "subscription": "both",
        "summoner_id": 334455
    },
    …
]
```

In order to support these requests, chat servers run Cowboy, an embedded web server, that handles the HTTP requests coming in from other internal services. For easier integration every endpoint self-documents via Swagger and returns JSON objects to fulfill requests.

Although we're currently using a centralized internal load balancer to dispatch these requests, in the future we'd prefer to move to a model that leverages automatic discovery and client-side load balancing mechanisms. This would allow us to dynamically resize our clusters without network reconfigurations, and make setting up new shards (internal for testing, or external for players) much easier, as services auto-configure themselves.

SUMMARY

In a single day, Riot chat servers often route a billion events (presences, messages, and IQ stanzas) and process millions of REST queries. While the system certainly isn't perfect, the stable and scalable nature of the infrastructure keeps chat available to players—while hardware issues take down individual servers from time to time, the self-healing nature of the system has provided five 9's of uptime in all Riot regions this year. Although I've only scratched the surface here, I hope it sheds some light on how chat functions on the server-side.

If you have any questions or comments I would be really excited to read them. The next and final article in this series introducing Riot chat will focus on the databases we employ for persisting data on the service side. See you then!

Posted by Michal Ptaszek

Join the discussion…

yenic • 21 days ago

My languages of choice are Python(2, of course) and Erlang. Most of my fellows seem obsessed with multicore programming, which IMO is only useful for latency bound situations such as an OS or game engine. For everything else, multinode wins handily. I'm pleased to see yet-another one of my favorite applications building on the fantastic BEAM VM.

1 • Reply • Share ›

michalptaszek Rioter ➦ yenic • 21 days ago

Yup, we are pretty huge fans of BEAM VM at Riot as well ;)

1 • Reply • Share ›

Davi Alves • 21 days ago

It's always nice to understand how the backend of my favorite game works. Also I'm currently developing an android messaging app. In my case we use Openfire. Good job with the articles, they are very informative.

1 • Reply • Share ›

Shawn Clark • 9 days ago

XMPP with Erlang screams of Tigase. Might that have been the implementation that you use / had started with? I would have been curious to hear if you did or not and if you collaborated with them on the project. I had the fortunate opportunity to work with Tigase as part of Disney's Club Penguin presence and cross world chat server. Very extensible application with great scalability.

Also curious if you contributed anything to XMPP Extensions? http://xmpp.org/xmpp-protocols...

• Reply • Share ›

michalptaszek Rioter ➦ Shawn Clark • 8 days ago

Hey Shawn,

Tigase is actually written in Java. As a part of our initial evaluation (huh, 4+ years ago - time flies fast!) we considered few alternatives to vanilla ejabberd (in version 2.1.8) - one of them was OpenFire, and the other one - Tigase.

AFAIR both of the Java-based implementations did not match stability, scalability and performance characteristics compared to out-of-the-box ejabberd, which got even better once profiled and optimized.

In the same time, all 3 projects have grown tremendously - it would be really interesting to compare them now - maybe the results of our evaluations would be different :)

Reply • Share ›

Shawn Clark Shawn Clark • 9 days ago

Found a year old video on scaling the chat.

Reply • Share ›

Shawn Clark Shawn Clark • 9 days ago

Found an answer to the second question. Looks like Riot did a more detailed writeup of how they used XMPP. https://engineering.riotgames....

Reply • Share ›

Maxim V • 19 days ago

Thank you for great post!

Do you persist chat messages, which were delivered to user's c2s process, but user's socket had disconnected by that time?

Reply • Share ›

michalptaszek Rioter Maxim V • 19 days ago

Yes, all p2p messages that reached the chat servers are stored persistently in the data store for future delivery (including these that hit c2s which has "invalid" state with a dead client socket).

Reply • Share ›

Maxim V michalptaszek • 19 days ago

I see. Any plans for a post about chat rooms and in-game chats architecture?

architecture?

Reply • Share ›

michalptaszek Rioter ➦ Maxim V • 19 days ago

We definitely want that feature for new moderated lobby chat rooms (which are already on PBE).

pre- and post-game chat rooms do not require persistency (since you are not likely to read any messages posted to them anymore), however we might add memory-based server-side caching to allow clients to download messages that were published to that room before the join (so that no one will ever miss early role call ;)).

1 Reply • Share ›

obeycelestia ➦ michalptaszek • 19 days ago

Pre- and post- might still have logging for backend analysis later even though there's no persistence accessible from the client, right?

Not from a client perspective, but you may want to analyze message volume pre vs in vs post, etc, or to corroborate a Tribunal-type thing looking in the future, etc.

Reply • Share ›

michalptaszek Rioter ➦ obeycelestia • 16 days ago

100% true. This however utilizes read-only data store specifically optimized for analytics.

1 Reply • Share ›

obeycelestia ➦ michalptaszek • 16 days ago

Awesome! That's how I would've figured it would go since RO-data write concerns are always easier to work with (if only that was what everyone wanted by default. :p)

Also, thanks for your patience and interest in talking to us about all this stuff! I find it fascinating.

1 Reply • Share ›

Asado • 20 days ago

Great article, great architecture.
I didn't get one thing: what would have happened to Alice and Bob, when the server wouldn't reconnect for let's say several hours due to a fatal hardware error?

Reply • Share ›

obeycelestia ➦ Asado • 20 days ago

I believe their messages would accumulate in Node A's persisted message

store and just chill until it regained connectivity, then they would be processed like normal?

Did I understand the question correctly?

Reply • Share ›

michalptaszek Rioter ➚ obeycelestia • 20 days ago

Right, everything would be stored in either message history/offline message spool in Riak, and merged with the remaining clusters once the connectivity is brought back.

In the same time network failures lasting few hours wouldn't be left unnoticed. We have HW redundancy (2 routers, 2 switches, 2 load balancers) to prevent that from happening. If it's really really critical I would say we would shut down the "smaller" disconnected part of the cluster and let the other part handle all the users until we replace the broken equipment.

Reply • Share ›

DashK • 21 days ago

Have you guys looked at existing XMPP server implementation like Jabber, OpenFire? What are the reasons that you guys go with implementing your own XMPP server?

Reply • Share ›

michalptaszek Rioter ➚ DashK • 21 days ago

Absolutely. We started with open source implementation of ejabberd, and over time (huh, it's been more than 4 years) we effectively rewritten 95% of it (efficiency, custom features, stability, etc). I think of it more as of long term evolution of the software than building a server from scratch.

Reply • Share ›

obeycelestia • 21 days ago

Erlang seems like a good choice for the actor model. But really, any language that supports functional programming and actors would be a good choice. :)

Apropos of the netsplit example:

Let's say Alice sent a message while Node A was siloed off from B and C and her connection to Node A gets interrupted. There's no way it could reach the persisted messages datastore log, right? In that case is there an extra layer of redundancy on the AIR client side if there's no response from the server?

Reply • Share ›

Nathan A. ➚ obeycelestia • 21 days ago

This isn't really true. Part of the reason this works so well in Erlang is because the OTP framework, concurrency model, and fault-tolerance semantics were designed and integrated right along with the language and runtime

designed and integrated right along with the language and runtime environment. So each component is built and iterated on with the same overarching focus and purpose.

For example, actor-model concurrency on a VM that can't preempt the execution contexts of the actors and shuffle them around a small pile of OS-level threads is going to have a very, very difficult time producing the soft-realtime semantics you get with Erlang (sans when NIFs muck things up for you, though better now with dirty schedulers).

This is great for systems which need to maintain a vary narrow deviation on latency, ie. control planes, messaging services, transaction processing, etc.

1 • Reply • Share ›

obeycelestia Nathan A. • 21 days ago

It isn't really false, either? Not really a dichotomy there. :)

I admit to thinking of Clojure when I wrote the first paragraph, but a number of other languages that support native concurrency and fault-tolerant semantics would've worked well. I agree that Erlang was a particularly apropos choice for the domain, however!

When building scaling applications I've personally found that the best tools for the job are the ones that give me clear, simple, concise ways to idiomatically express the process rather than hack it into a utility library somewhere to cover up a lack of native support for the data and/or processes I'm working with. I'm a fan of easily grok-able code so I can understand the context of any changes that needs to be made to the model since everything start to multiply at scale. Erlang certainly delivers there.

I tend to work backwards from understandability to performance (looks like Riot did too,) as they went from a pure implementation to 'unrolling their loops' through C NIFs in the most heavily used pieces for performance. I like their general approach and it's good to see it works under such a heavy user-load.

• Reply • Share ›

Nathan A. obeycelestia • 21 days ago

I chose the example above intentionally. There are three ways that I know of to attempt to get soft-realtime semantics with a simple coherent concurrency model like M actors scheduled over N threads on the JVM. Clojure included.

1) is not really deal with it at all, but mitigate the most common source of runtime blocking (GC pauses) by using a pricey proprietary JVM like Zing. But you're still stuck with myriad other ways to block the scheduler pool.

2) is to have the inverse of the first problem by using a framework which instruments the code, has an observer agent, and monkeys around with bytecode injection at runtime like Quasar. But you're still burned by VM semantics that the framework can't prevent from blocking even though it can force-yield your application code.

3) hope you can actually solve the problem by creating a Frankonsolution of #1 and #2

see more

1 • Reply • Share ›

obeycelestia Nathan A. • 21 days ago

Each of these languages we've mentioned is Turing complete and none of them provide anything the others theoretically couldn't with a library or something akin to that-- just like software just provides whatever hardware can at a virtual level.

As you rightly said, everything is a trade-off and you just have to choose at what level you try and solve the problem. Performance is just up to trying to solve the problem in the most idiomatic way your language can be used to express it while weighing the option of shelling-out to another, lower-level routine at the expense of your native language's semantics in favor of an increase in execution-time. Erlang solves the problem very natively, so it's nice and well specified and all the language semantics support it.

You have a lot of good points and I'm certainly not a JVM apologist and can readily point out many of its pain points, haha. I guess my original observation was that Erlang wasn't the only viable choice, but it was a very good one. I don't really think we're fundamentally disagreeing. :)

• Reply • Share ›

Nathan A. obeycelestia • 21 days ago

FWIW, I'm not trying to argue with you. That said, I'm not sure I'm seeing the connection to Turing Completeness. That seems orthogonal to the shape and character of the semantics different implementations of abstraction expose as they bubble up from the hardware and trickle down from the software.

Ruby is Turing Complete, yet it's mainline runtime lacks the necessary integration with the underlying system to actually expose parallel thread execution. In 1.8 this was due to lack of exposing the Ruby threading implementation to the kernel + the global interpreter lock. In 1.9 this was still the case because of

the latter. So despite being a Turing Complete language, no amount of Ruby code you'd write to run on the MRI runtime would be able to exercise or utilize the underlying system the same way as the same Ruby code running on the JVM via JRuby. Both the same Turing Complete language, but extremely different underlying systems and abstractions to produce drastically different behavior.

Going back to my original point, this is what makes Erlang + OTP + BEAM somewhat unique for certain shapes of problems that sometimes can't be replicated, and often can only be mimicked, in other systems that didn't design the entire stack to facilitate the programming model that Erlang provides. To put a finer point on it, it would be possible to write a VM that could run an Erlang program, while still being completely incapable of doing the rest of what people expect from Erlang operationally and executionally (I am stunned that's a word :-)).

2 • Reply • Share ›

obeycelestia Nathan A. • 21 days ago

Totally wasn't reading argumentative into it, just very precise. I appreciate this type of discussion and I hope you're enjoying it as much as I am. :D

Your point was a fine-grained one and one I didn't parse completely on first read. I think I see more precisely what you were trying to say now.

What I meant with the Turing complete thing was exactly what you expressed with Ruby. It all does the same thing, just expressed in different ways in the underlying implementation. Performance is all about the specific implementation matching the problem set and not just solving the problem generically (also hilariously a word.) Like choosing a sorting algorithm based on the expected problem composition to maximize performance. Erlang is solving the problem specifically.

I think me missing the original main thrust of your statement led to us agreeing vehemently from opposite end of the idea. :P

• Reply • Share ›

michalptaszek Rioter obeycelestia • 21 days ago

I haven't explained it here, but each server essentially runs 2 services: Chat and Riak. When posting messages to "offline" contacts they also get persisted in Riak.

Once the network connectivity is re-established Riak cluster would merge back, potentially creating siblings for conflicting updates. Since each object

stored in Riak is a CRDT, we can resolve the conflicts automatically on reads, and deliver missing messages back to the clients, if needed.

At the moment we don't support client-side buffering/resends/message receipts, so clients treat sending a message as "fire and forget" event.

1 • Reply • Share ›

obeycelestia michalptaszek • 21 days ago

That's the default behavior I'd expect. I know how hard putting together a realtime uber-graph with conflict-resolution mechanics is. I hear you guys are re-engineering the client with AstralFoxy (last I heard) so that's a completely reasonable behavior to have while trying to tackle the n-to-n social graph problem that one eventually has to address when making these types of systems. :p

1 • Reply • Share ›

Nikko • 21 days ago

The RSS feed <link>'ed to in your blog's <head> at http://engineering.riotgames.c... is http://engineering.riotgames.c.... This URL returns a 403 error. The RSS feed linked in the body of your page is http://engineering.riotgames.c... and that one works. You might want to update the URL in the <link> tag to a working URL.

• Reply • Share ›

michalptaszek Rioter Nikko • 21 days ago