EXHIBIT D

New Tweets per second record, and how!
https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html

(10 pages)

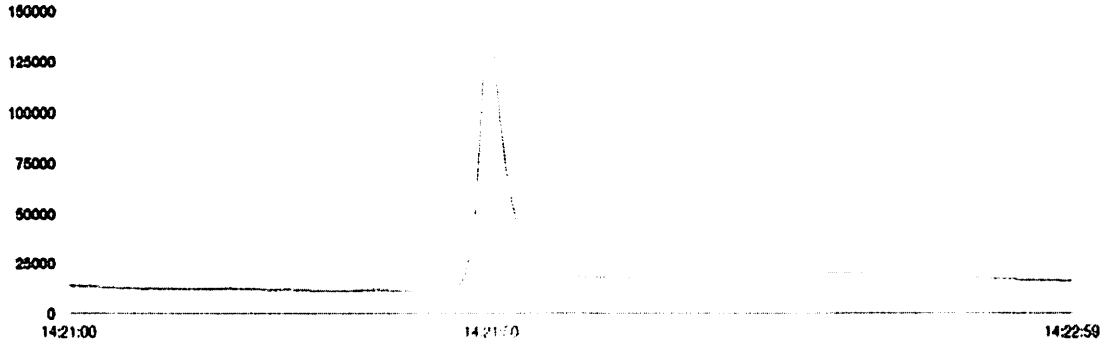
Engineering

New Tweets per second record, and how!

By @raffi
Friday, 16 August 2013

Recently, something remarkable happened on Twitter: On Saturday, August 3 in Japan, people watched an airing of Castle in the Sky (http://en.wikipedia.org/wiki/Castle_in_the_Sky), and at one moment they took to Twitter so much that we hit a one-second peak of 143,199 Tweets per second. (August 2 at 7:21:50 PDT; August 3 at 11:21:50 JST)

To give you some context of how that compares to typical numbers, we normally take in more than 500 million Tweets a day which means about 5,700 Tweets a second, on average. This particular spike was around 25 times greater than our steady state.



During this spike, our users didn't experience a blip on Twitter. That's one of our goals: to make sure Twitter is always available no matter what is happening around the world.

> New Tweets per second (TPS) record: 143,199 TPS. Typical day: more than 500 million Tweets sent; average 5,700 TPS.

This goal felt unattainable three years ago, when the 2010 World Cup put Twitter squarely in the center of a real-time, global conversation (https://blog.twitter.com/2010/2010-world-cup-global-conversation). The influx of Tweets — from every shot on goal, penalty kick and yellow or red card — repeatedly took its toll and made Twitter unavailable for short periods of time. Engineering worked throughout the nights during this time, desperately trying to find and implement order-of-magnitudes of efficiency gains. Unfortunately, those gains were quickly swamped by Twitter's rapid growth, and engineering had started to run out of low-hanging fruit to fix.

After that experience, we determined we needed to step back. We then determined we needed to re-architect the site to support the continued growth of Twitter and to keep it running smoothly. Since then we've worked hard to make sure that the service is resilient to the world's impulses. We're now able to withstand events like Castle in the Sky viewings, the Super Bowl, and the global New Year's Eve celebration. This re-architecture has not only made the service more resilient when traffic spikes to record highs, but also provides a more flexible platform on which to build more features faster, including synchronizing direct messages across devices, Twitter cards that allow Tweets to become richer and contain more content, and a rich search experience that includes stories and users. And more features are coming.

Below, we detail how we did this. We learned a lot. We changed our engineering organization. And, over the next few weeks, we'll be publishing additional posts that go into more detail about some of the topics we cover here.

Starting to re-architect

After the 2010 World Cup dust settled, we surveyed the state of our engineering. Our findings:

- We were running one of the world's largest Ruby on Rails installations, and we had pushed it pretty far — at the time, about 200 engineers were contributing to it and it had gotten Twitter through some explosive growth, both in terms of new users as well as the sheer amount of traffic that it was handling. This system was also monolithic where everything we did, from managing raw database and memcache connections through to rendering the site and presenting the public APIs, was in one codebase. Not only was it increasingly difficult for an engineer to be an expert in how it was put together, but also it was organizationally challenging for us to manage and parallelize our engineering team.
- We had reached the limit of throughput on our storage systems — we were relying on a MySQL storage system that was temporally sharded and had a single master. That system was having trouble ingesting tweets at the rate that they were showing up, and we were operationally having to create new databases at an ever increasing rate. We were experiencing read and write hot spots throughout our databases.
- We were "throwing machines at the problem" instead of engineering thorough solutions — our front-end Ruby machines were not handling the number of transactions per second that we thought was reasonable, given their horsepower. From previous experiences, we knew that the machines could do a lot more.
- Finally, from a software standpoint, we found ourselves pushed into an "optimization corner" where we had started to trade off readability and flexibility of the codebase for performance and efficiency.

We concluded that we needed to start a project to re-envision our system. We set three goals and challenges for ourselves:

- We wanted big infrastructure wins in performance, efficiency, and reliability — we wanted to improve the median latency that users experience on Twitter as well as bring in the outliers to give a uniform experience to Twitter. We wanted to reduce the number of machines needed to run Twitter by 10x. We also wanted to isolate failures across our infrastructure to prevent large outages — this is especially important as the number of machines we use go up, because it means that the chance of any single machine failing is higher. Failures are also inevitable, so we wanted to have them happen in a much more controllable manner.
- We wanted cleaner boundaries with "related" logic being in one place — we felt the downsides of running one particular monolithic codebase, so we wanted to experiment with a loosely coupled services oriented model. Our goal was to encourage the best practices of encapsulation and modularity, but this time at the systems level rather than at the class, module, or package level.
- Most importantly, we wanted to launch features fast. We wanted to be able to run small and empowered engineering teams that could make local decisions and ship user-facing changes, independent of other teams.

We prototyped the building blocks for a proof of concept and benchmark. Not everything we tried worked and not everything met the above goals. But we were able to settle on a set of principles, tools, and infrastructure that has gotten us to a much more debuggable and reliable state today.

The JVM vs the Ruby VM

First, we evaluated our front-end servers for a number of dimensions: CPU, RAM, and network. Our Ruby-based machinery was being pushed to the limit on the CPU and RAM dimensions — but we weren't serving that many requests per machine nor were we coming close to saturating our network. Because Ruby servers, at the time, had to be effectively single threaded and handle only one request at a time. Each Rails host was running a number of Unicorn processes to provide host-level concurrency, but the duplication there translated to wasteful resource utilization. When it came down to it, our Rails servers were capable of only 200 - 300 requests / sec / host.

Twitter's usage is always growing rapidly and doing the math there, it would take a lot of machines to keep up with the growth curve.

At the time, Twitter had experience deploying fairly large scale JVM-based services — our search engine was written in Java, and our Streaming Api infrastructure as well as Flock, our social graph system (https://blog.twitter.com/2010/introducing-flockdb), was written in Scala. We were enamored by the level of performance that the JVM gave us. It wasn't going to be easy to get our performance, reliability, and efficiency goals out of the Ruby VM, so we embarked on rewriting our code to be run on the JVM instead. We estimated that rewriting our codebase could get us a 10x performance improvement, on the same hardware - and now, today, we push on the order of 10 - 20K requests / sec / host.

There was a level of trust that we all had in the JVM. A lot of us had come from companies where we had experience working with, tuning, and operating large scale JVM installations. We were confident we could push the platform for Twitter in the world of the JVM. Now, we had to decompose our architecture and figure out how these different services would interact.

Programming model

In Twitter's Ruby systems, concurrency is managed at the process level: a single network request is queued up for a process to handle. That process is completely consumed until the network request is fulfilled. Adding to the complexity, architecturally, we were taking Twitter in the direction of having a service compose the responses of other services. Given that the Ruby process is single-threaded, Twitter's "response time" would be additive and extremely sensitive to the variances in the back-end systems' latencies. There were a few Ruby options that gave us concurrency; however, there wasn't one standard way to interface with all the different VM options. The JVM had constructs around concurrency and would let us build a real concurrent programming platform.

It became evident that we needed a single and consistent way to think about concurrency in our systems and, specifically, in the way we do our networking. As we all know, writing concurrent code (well) is hard and can take many forms. In fact, when we began moving off of this, we started to decompose the system into services, each team had slightly different approaches. For example, the failure semantics from clients to services didn't interact well: we had no consistent back-pressure mechanisms across services to clients and we experienced "thundering herds" when bringing latent services. These failure domains informed us that we were missing a unified, and complementary, client and server library that bundled in notions of

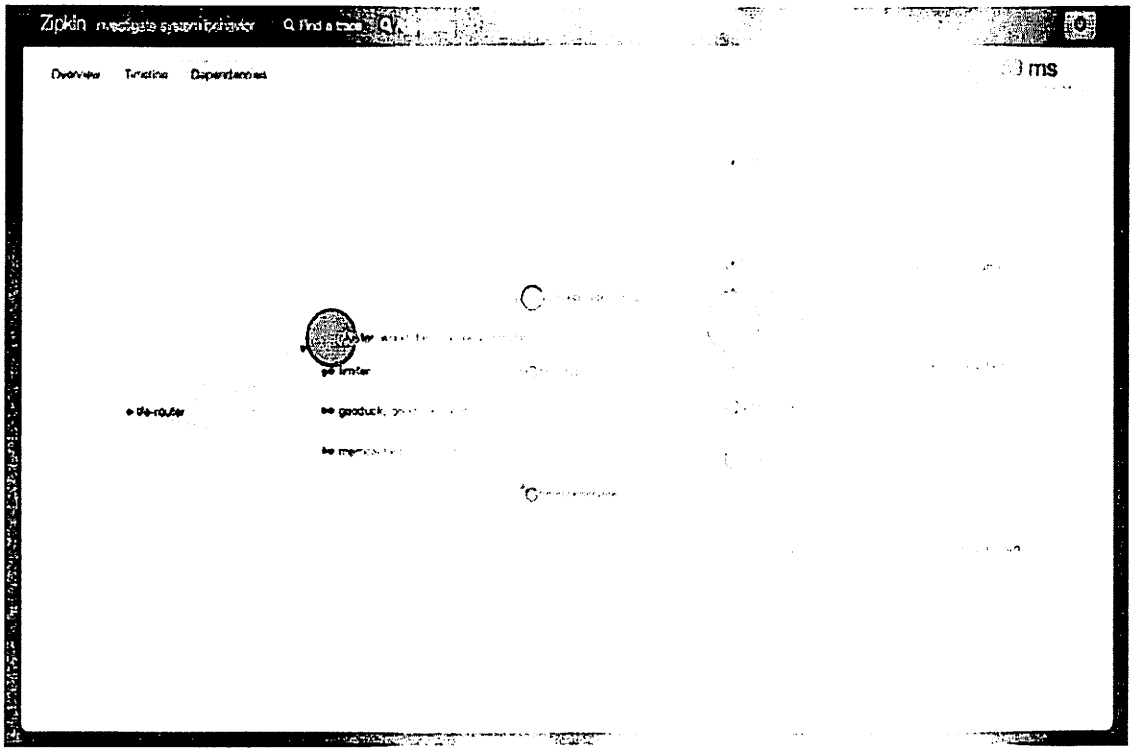
connection pools, failover strategies, and load balancing. To help us all get in the same mindset, we put together both Futures and Finagle (https://blog.twitter.com/2011/finagle-protocol-agnostic-rpc-system).

Now, not only did we have a uniform way to do things, but we also baked into our core libraries everything that all our systems needed so we could get off the ground faster. And rather than worry too much about how and every system operated, we could focus on the application and service features.

Independent systems

The largest architectural change we made was moving from our monolithic Ruby application to one that is more service oriented. We started breaking off functionality, creating Tweet, timeline, and user services — our "core nouns". This move afforded us cleaner abstraction boundaries and team-level ownership and independence. In our monolithic world, we either needed experts who understood the entire codebase or clear owners at the module or class level. Sadly, the codebase was growing too large to have global experts and, in practice, ownership at the module or class level wasn't working. Our codebase was becoming harder to maintain, and teams constantly spent time going on "archeology digs" to understand certain functionality. Or we'd organize "whale hunting expeditions" to find performance and large scale failures that occurred. At the end of the day, we were more focused on this than on shipping features, which we weren't happy about.

Our theory was, and still is, that a services oriented architecture allows us to develop the system in parallel — we agree on network boundaries, and then go develop the system internals independently. The business logic for each system was self-contained within. If we needed to change something about Tweets, we could make that change in one location, the Tweet service, and then that change would flow throughout our architecture properly. However, we find that not all teams plan for change in the services they depend on. A change in the Tweet service may require other services to do things differently if the API or representation changed. On balance, though, this worked out better than not.



This system architecture also mirrored the way we worked and now do, run the Twitter engineering organization. Engineering is [illegible] with (mostly) self-contained teams that can run independently and very quickly. This means that we bias toward teams spinning up and running their own services [illegible] and systems. This has huge implications on operations, however.

Storage

Even if we broke apart our monolithic application into services, a huge bottleneck that remained was storage. Twitter, at the time, was storing tweets in a single master MySQL database. We had taken the strategy of sharding data temporally — each row in the database was a single tweet, we stored the tweets in order in the database, and when the database filled up we spun up another one and reconfigured the software to start populating the next database. This strategy had bought us some time, but, we were still having issues keeping massive indexes in memory because they would all be serialized into a single [illegible] we were experiencing read load concentration on a small number of database machines. We needed a different partitioning strategy for Tweet storage.

We took Gizzard, our framework to create sharded and fault-tolerant distributed databases, and applied it to tweets. [illegible] Gizzard was fronting a series of MySQL databases — every tweet [illegible] the system, Gizzard hashes it, and then chooses an appropriate database. Of course, this means

we lose the ability to rely on MySQL for unique ID generation. Snowflake (https://blog.twitter.com/2010/announcing-snowflake) was born to solve that problem. Snowflake allows us to create an almost-guaranteed globally unique identifier. We rely on it to create new tweet IDs, at the tradeoff of no longer having "increment by 1" identifiers. Once we have an identifier, we can rely on Gizzard then to store it. Assuming our hashing algorithm would distribute uniformly distributed, we increase our throughput by the number of databases. Our reads are also then distributed across the entire cluster, rather than being pinned to the "most recent" database, allowing us to increase throughput there too.

Observability and statistics

We've traded our fragile monolithic application for a more robust and encapsulated, but also complex, services oriented application. We had to build tools to make managing this beast possible. Given the speed at which we were creating new services, we needed to make it incredibly easy to gather data on how well each service was doing. By default, we wanted to make data-driven decisions, so we needed to make it trivial and frictionless to get to that data.

As we were going to be spinning up, and down, a staggeringly large system, we had to make this easier for everybody. Our Observability team created two tools for engineering: Viz and Zipkin (https://blog.twitter.com/2012/distributed-systems-tracing-zipkin). Both of these tools are exposed integrated by Finagle, so all services that are built using Finagle have access to these tools.

```
stats.timeFuture("request_latency_ms") {
// dispatch to do work
}
```

The above code block is all that is needed for a service to begin sending data into Viz. From there, anybody using Viz can view a graph of the time series and graph of interesting data like the 50th and 99th percentile of request_latency_ms.

Runtime configuration and testing

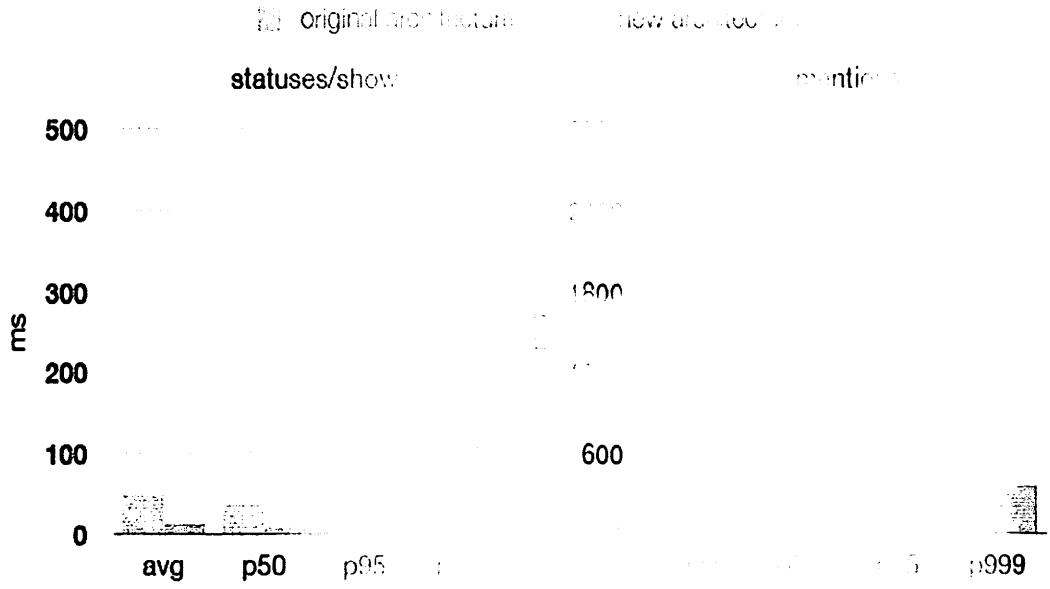
Finally, as we were putting this all together, we hit a few additional snags: launches had to be coordinated across several services, and we didn't have a place to stage services that required "live" traffic. We no longer rely on deployment as the vehicle to get new features out there, so coordination was going to be required across the application. In addition, given the relative size of Twitter, it was becoming difficult for us to run meaningful tests in totally isolated

environment. We had, relatively, no issues testing for correctness in our isolated systems — we needed a way to test for large scale iterations. We embraced runtime configuration.

We integrated a system we call Decider across all our services. It allows us to flip a single switch and have multiple systems across our [illegible] react to that change in near-real time. This means software and multiple systems can go into production when teams are ready, but a particular feature doesn't need to be "active". Decider also allows us to have the flexibility to do binary and percentage based switching such as having a feature available to [illegible] users. We can deploy code in the fully "off" and safe setting, and [illegible] and down until we are confident it's operating correctly and systems can handle the new load. All this alleviates our need to do any coordination at the team level, and instead we can do it at runtime.

Today

Twitter is more performant, efficient and reliable than [illegible]. We've sped up the site incredibly across the 50th (p50) through 99th [illegible] distributions and the number of machines involved in serving the site [illegible] decreased anywhere from 5x-12x. Over the last six months, Twitter has flirted with four 9s of availability.



Twitter engineering is now set up to mimic our software stack. We have teams that are ready for long term ownership and to be experts on their part of the Twitter infrastructure. Those teams own their interfaces and their problem domains. Not every team at Twitter needs to worry about scaling Tweets, for example. Only a few teams — those that are involved in the running of the Tweet subsystem (the Tweet service team, the storage team, the caching team, etc.) — have to scale the writes and reads of Tweets, and the rest of Twitter engineering gets APIs to help them use it.

Two goals drive us as we did all this work: Twitter should always be available for our users, and we should spend our time making Twitter more engaging, more useful and simply better for our users. Our systems and our engineering team now enable us to launch new features faster and in parallel. We can dedicate different teams to work on improvements simultaneously and have minimal bogjams for when those features collide. Services can be launched and deployed independently from each other (in the last week, for example, we had more than 50 deploys across all Twitter services), and we can defer putting everything together until we're ready to make a new build for iOS or Android.

Keep an eye on this blog and @twitterengg (http://twitter.com/twitterengg) for more posts that will dive into details on some of the topics mentioned above.

Thanks goes to Jonathan Reichhold (https://twitter.com/jreichhold), David Helder (@dhelder (https://twitter.com/dhelder)), Marcel Molina (@noradio (https://twitter.com/noradio)) and Matt Harris (@mattharris (https://twitter.com/mattharris)) for helping contribute to this blog post.

Share:

Link copied successfully