7 Years of YouTube Scalability Lessons in 30 Minutes - High Scalability -

High Scalability

Example

If you started out building a dating site and instead ended up building a video sharing site (YouTube) that handles 4 billion views a day, then it's just possible you learned something along the way. And indeed, Mike Solomon, one of the original engineers at YouTube, did learn a lot and he has given a talk about it at PyCon: Scalability at YouTube.

This isn't an architecture driven talk where we are led through a description of how a lot of boxes connect to each other. Mike could give that sort of talk. He has worked on building YouTube's servlet infrastructure, video indexing feature, video transcoding system, their full text search, a CDN, and much more. But instead, he's taken a step back, took a long look around at what time has wrought, and shared some deep lessons, obviously hard won from experience.

The key takeaway away of the talk for me was doing **a lot with really simple tools**. While many teams are moving on to more complex ecosystems, YouTube really does keep it simple. They program primarily in Python, use MySQL as their database, they've stuck with Apache, and even new features for such a massive site start as a very simple Python program.

That doesn't mean YouTube doesn't do cool stuff, they do, but what makes everything work together is more a philosophy or a way of doing things than technological hocus pocus. What made YouTube into one of the world's largest websites? Read on and see...

Stats

- 4 billion Views a day
- 60 hours of video is uploaded every minute

- 350+ million devices are YouTube enabled
- Revenue double in 2010
- The number of videos has gone up 9 orders of magnitude and the number of developers has only gone up two orders of magnitude.
- 1 million lines of Python code

Stack

- **Python -** most of the lines of code for YouTube are still in Python. Everytime you watch a YouTube video you are executing a bunch of Python code.
- **Apache** - when you think you need to get rid of it, you don't. Apache is a real rockstar technology at YouTube because they keep it simple. Every request goes through Apache.
- **Linux** - the benefit of Linux is there's always a way to get in and see how your system is behaving. No matter how bad your app is behaving, you can take a look at it with Linux tools like strace and tcpdump.
- **MySQL** - is used a lot. When you watch a video you are getting data from MySQL. Sometime it's used a relational database or a blob store. It's about tuning and making choices about how you organize your data.
- **Vitess** - a new project released by YouTube, written in Go, it's a frontend to MySQL. It does a lot of optimization on the fly, it rewrites queries and acts as a proxy. Currently it serves every YouTube database request. It's RPC based.
- **Zookeeper** - a distributed lock server. It's used for configuration. Really interesting piece of technology. Hard to use correctly so read the manual
- **Wiseguy** - a CGI servlet container.
- **Spitfire** - a templating system. It has an abstract syntax tree that let's them do transformations to make things go faster.
- **Serialization formats** - no matter which one you use, they are all expensive. Measure. Don't use pickle. Not a good choice. Found protocol buffers slow. They wrote their own BSON implementation which is 10-15 time faster than the one you can download.

General Lessons

- **Tao of YouTube**: choose the simplest solution possible with the loosest guarantees that are practical. The reason you want all these things is you need flexibility to solve problems. The minute you over specify something you paint yourself into a corner. You aren't going to make those guarantees. Your problem becomes automatically more complex when you try and make all those guarantees. You leave yourself no way out.
- **That whole process is what scalability is about**. A scalable system is one that's not in your way. That you are unaware of. It's not buzz words. It's a general problem solving ethos.
- **Hallmark of big system design**: Every system is tailored to its specific requirements. Everything depends on the specifics of what you are building.
- **YouTube is not asynchronous**, everything is blocking.
- **Believes more in philosophy than doctrine**. Make it simple. What does that mean? You'll know when you see it. If you do code review that changes thousands of lines of code and many files then there was probably a simpler way. Your first demo should be simple, then iterate.
- **To solve a problem: One word - simple**. Look for the most simple thing that will address the problem space. There are lots of complex problems, but the first solution doesn't need to

be complicated. The complexity will come naturally over time.
- **A lot of YouTube systems start as one Python file** and become large ecosystems after many many years. All their prototype were written in Python and survived for a surprising amount of time.
- **In a design review**:
  - What's the first solution?
  - How are you going to iterate?
  - What do we know about how this data is going to be used?
- **Things change over time**. How YouTube started out has no bearing on what happens later. YouTube started out as a dating site. If they had designed for that they would have different conversation. Stay flexible.
- **YouTube CDN**. Originally contracted it out. Was very expensive so they did it themselves. You can build a pretty good video CDN if you have a good hardware dude. You build a very large rack, stick machines in, then take lighttpd, and then override the 404 handler to find the video that you didn't find. That took two weeks and it's first day served 60 gigabits. You can do a lot with really simple tools.
- **You have to measure**. Vitess swapped out one its protocols for an HTTP implementation. Even though it was in C it was slow. So they ripped out HTTP and did a direct socket call using python and that was 8% cheaper on global CPU. The enveloping for HTTP is really expensive.

Scalability Techniques

- These are not new ideas, but it's amazing how a few core ideas can apply in a lot different dimensions.
- **Divide and Conquer - The Scalability Technique**
  - This is the scalability technique. Everything is about partitioning out work. Deciding how to execute it. Applies to many things, from web tier, you have a lot of web servers that are more or less identically and independently and you grow them horizontally. That's divide and conquer.
  - This is the crux of database sharding. How do you partitions things out and communicate between the parts that you've subdivided. These are things you want to figure out early on because they influence how you grow.
  - Simple and loose connections are really valuable.
  - The dynamic nature of Python is a win here. No matter how bad your API is you can stub or modify or decorate your way out of a lot of problems.
- **Approximate Correctness - Cheat a Little**
  - Another favorite technique. The state of the system is that which it is reported to be. If a user can't tell a part of the system is skewing and inconsistent, then it's not.
  - A real world example. If you write a comment and someone loads the page at the same time, they might not get it for 300-400ms, the user who is reading won't care. The writer of the comment will care, so you make sure the user who wrote the comment will see it. So you cheat a little bit. Your system doesn't have to have globally consistent transactions. That would be super expensive and overkill. Not every comment is a financial transaction. So know when you can cheat.
- **Expert Knob Twiddling**
  - Ask, what do you know about your consistency model? For comments is eventually consistent good enough? Renting a movie is different. When renting there's money so

we'll do the best we can to never lose that. Different consistency models are needed depending on the data.

- **Jitter - Add Entropy Back into Your System**
  - Hot word in their group all of the time. If your system doesn't jitter then you get thundering herds. Distributed applications are really weather systems. Debugging them is as deterministic as predicting the weather. Jitter introduces more randomness because surprisingly, things tend to stack up.
  - For example, cache expirations. For a popular video they cache things as best they can. The most popular video they might cache for 24 hours. If everything expires at one time then every machine will calculate the expiration at the same time. This creates a thundering herd.
  - By jittering you are saying randomly expire between 18-30 hours. That prevents things from stacking up. They use this all over the place. Systems have a tendency to self synchronize as operations line up and try to destroy themselves. Fascinating to watch. You get slow disk system on one machine and everybody is waiting on a request so all of a sudden all these other requests on all these other machines are completely synchronized. This happens when you have many machines and you have many events. Each one actually removes entropy from the system so you have to add some back in.
- **Cheating - Know How to Fake Data**
  - Awesome technique. The fastest function call is the one that doesn't happen. When you have a monotonically increasing counter, like movie view counts or profile view counts, you could do a transaction every update. Or you could do a transaction every once in awhile and update by a random amount and as long as it changes from odd to even people would probably believe it's real. Know how to fake data.
- **Scalable Components - Make Your own Luck**
  - **You can look at an API and get a good feel**. Are the inputs well defined? Do you know what you are getting out? A lot of this ends up being about data. Have a tight specification of what data comes out every function and how it flows actually helps you understand the application without documentation. You can tell what's happening before and after a function is called.
  - **In Python things tend to move towards RPCs**. The structure of your code is based on the discipline of your programmers. So establish good conventions, when all else fails there's an RPC wall so you know what goes in and what comes out.
  - **Your components will not be perfect**. A component might last a month or six months, who knows. By drawing these lines you are making some of your own luck. When things go south you can swap it out and do something different. Sometimes that rewriting someing in python and C and sometimes that means getting rid of it entirely. You don't know until you are able to observe.
  - **With so many people on a team nobody can know the whole system**, so you need to define components. This is video transcode it's distinct from video search. You want well defined subcomponents. It's good software design. These things end up talking to each other so having a good data specification is helpful. The greatest sin he made was communication between the servlet layer and the template layer to be a dictionary. Very bad idea. Should have added a WatchPage and said a watch page had a video and some comments and some related videos. This caused a lot of problems because the dictionary can have a few hundred attributes. They don't always make the right choice.
- **Efficiency - Traded Off for Scalability**
  - **Efficiency is traded off for scalability**. The most efficient thing is to write it in C

and cram it into one process, but that's not scalable.
    - **Focus on the macro level, your components, and how they break out**. Does it makes sense to do this an RPC or do it inline? Break it into a subpackage and just someday this may be different.
    - **Focus on algorithms**. In Python the effort to implement a good algorithm is low. There's the bisect module, for example, where you can take a list, do something meaningful, and serialize it to disk and read it back again. There's a penalty versus C, but it's very easy.
    - **Measurement**. In Python measurement is like reading tea leaves. There's a lot of things in Python that are counter intuitive, like the cost of grabage colleciton. Most of chunks of their apps spend their time serializing. Profiling serialization is very depending on what you are putting in. Serializing ints is very different than serializing big blobs.
- **Efficiency in Python - Knowing What Not to Do**
    - **More about knowing what not to do**. How dynamic you make things correlates to how expensive it is to run your Python app.
    - **Dummer code is easier to grep for and easier to maintain**. The more magical the code is the harder is to figure out how it works.
    - **They don't do a lot of OO**. They use a lot of namespaces. Use classes to organize data, but rarely for OO.
    - **What is your code tree going to look like?** He wants these words to describe it: simple, pragmatic, elegant, orthogonal, composable. This is an ideal, reality is a bit different.

Related Articles

- Super-sizing YouTube with Python
- Scaling MySQL Databases for the Web
- YouTube Architecture
- On HackerNews