Scaling the Netflix API
Please read the notes associated with each slide for the full context of the presentation
What do I mean by “scale”?
But There Are Many Ways to Scale!

Organization

Systems

Devices

Development

Testing

To have an effective engineering organization, you need to scale in a variety of ways, not just in your systems. This presentation discusses these scaling needs. Of course, I will focus a bit on systems, but that is not the only area that requires focus to be successful.
But first, some background...
Global Streaming Video

Monday, July 29, 13
Netflix strives to be the global streaming video leader for TV shows and movies
Monday, July 29, 13
We now have more than 38 million global subscribers in more than 40 countries
Netflix subscribers are watching more than 1 billion hours a month

Those subscribers consume more than a billion hours of streaming video a month which accounts for about 33% of the peak Internet traffic in the US.
Our 38 million Netflix subscribers are watching shows (like House of Cards) and movies on virtually any device that has a streaming video screen. We are now on more than 1,000 different device types.
Monday, July 29, 13

Shortly after streaming launched, in 2008, we launched our REST API. I describe it as a One-Size-Fits-All (OSFA) type of implementation because the API itself sets the rules and requires anyone who interfaces with it to adhere to those rules. Everyone is treated the same.
The OSFA API launched to support the 1,000 flowers model. That is, we would plant the seeds in the ground (by providing access to our content) and see what flowers sprout up in the myriad fields throughout the US. The 1,000 flowers are public API developers. At the launch of the public API, the content was fully liberated and the bird was set free to fly around in the open world.
And at launch, the API was exclusively targeted towards and consumed by the 1,000 flowers (i.e., External developers). So all of the API traffic was coming from them.
Monday, July 29, 13

Some examples of the flowers...
But as streaming gained more steam...
The API evolved to support more of the devices that were getting built. The 1,000 flowers were still supported as well, but as the devices ramped up, the devices became a bigger focus.
Meanwhile, the balance of requests by audience had completely flipped. Overwhelmingly, the majority of traffic was coming from Netflix-ready devices and a shrinking percentage was from the 1,000 flowers. The flowers now represent less than 0.1% of the total traffic to the Netflix API.
To support the growing member base, growing number of devices, and growing feature-set for the application, we have needed to scale in a variety of ways. First, the organization...
Our application used to be more like a monolithic application running out of data centers. As the streaming application started to emerge, and as we started to shift to the cloud, we began to split out different services towards a distributed, SOA-based architecture.
I like to think of this distributed architecture as being shaped like an hourglass...
Monday, July 29, 13

In the top end of the hourglass, we have our device and UI teams who build out great user experiences on Netflix-branded devices. To put that into perspective, there are a few hundred more device types that we support than engineers at Netflix.

1000+ Device Types
Monday, July 29, 13

At the bottom end of the hourglass, there are several dozen dependency teams who focus on things like metadata, algorithms, authentication services, A/B test engines, etc.
Monday, July 29, 13

The API is at the center of the hourglass, acting as a broker of data.
Monday, July 29, 13

This hourglass architecture allows us to scale horizontal easily with our device integrations and our backend dependencies.
The glue that helps all of this work as effectively as it does is our engineering culture. We hire great, seasoned engineers with excellent judgment and engineering acumen and enable them to build systems quickly by giving them the right context and helping them work together in a highly aligned way.
Scaling...
System Resiliency
Our organization, and therefore our system, is set up to support a distributed architecture.
Dependency Relationships

Our distributed architecture, with the number of systems involved, can get quite complicated. Each of these systems talks to a large number of other systems within our architecture.
2,000,000,000
Requests Per Day to the Netflix API

Monday, July 29, 13
Distinct, Direct Dependent Services for the Netflix API
14,000,000,000

Netflix API Calls Per Day to those Dependent Services
Dependent Services with 100% SLA
99.999%^{30} = 99.7%

0.3% of 2B = 6M failures per day

2+ Hours of Downtime Per Month

Monday, July 29, 13

Assuming each of the services have SLAs of four nines, that results in more than two hours of downtime per month.
99.999%³⁰ = 99.7%

0.3% of 2B = 6M failures per day

2+ Hours of Downtime Per Month

Monday, July 29, 13
And that is if all services maintain four nines!
99.9\%^{30} = 97\% 

3\% of 2B = 60M failures per day 

20+ Hours of Downtime Per Month 

Monday, July 29, 13
If it degrades as far as to three nines, that is almost one day per month of downtime!
Monday, July 29, 13

So, back to the hourglass...
Monday, July 29, 13

In the old world, the system was vulnerable to such failures. For example, if one of our dependency services fails...
Monday, July 29, 13

Such a failure could have resulted in an outage in the API.
Monday, July 29, 13
And that outage likely would have cascaded to have some kind of substantive impact on the devices.
Monday, July 29, 13

The challenge for the API team is to be resilient against dependency outages, to ultimately insulate Netflix customers from low level system problems and to keep them happy.
To solve this problem, we created Hystrix, as wrapping technology that provides fault tolerance in a distributed environment. Hystrix is also open source and available at our github repository.
To achieve this, we implemented a series of circuit breakers for each library that we depend on. Each circuit breaker controls the interaction between the API and that dependency. This image is a view of the dependency monitor that allows us to view the health and activity of each dependency. This dashboard is designed to give a real-time view of what is happening with these dependencies (over the last two minutes). We have other dashboards that provide insight into longer-term trends, day-over-day views, etc.
This is a view of a single circuit.
Monday, July 29, 13

This circle represents the call volume and health of the dependency over the last 10 seconds. This circle is meant to be a visual indicator for health. The circle is green for healthy, yellow for borderline, and red for unhealthy. Moreover, the size of the circle represents the call volumes, where bigger circles mean more traffic.
The blue line represents the traffic trends over the last two minutes for this dependency.
Monday, July 29, 13
The green number shows the number of successful calls to this dependency over the last two minutes.
Successful, But Slower Than Expected

The yellow number shows the number of latent calls into the dependency. These calls ultimately return successful responses, but slower than expected.
Monday, July 29, 13

The blue number shows the number of calls that were handled by the short-circuited fallback mechanisms. That is, if the circuit gets tripped, the blue number will start to go up.
Timeouts, Delivering Fallbacks

The orange number shows the number of calls that have timed out, resulting in fallback responses.
The purple number shows the number of calls that fail due to queuing issues, resulting in fallback responses.
The red number shows the number of exceptions, resulting in fallback responses.
The error rate is calculated from the total number of error and fallback responses divided by the total number calls handled.
If the error rate exceeds a certain number, the circuit to the fallback scenario is automatically opened. When it returns below that threshold, the circuit is closed again.
The dashboard also shows host and cluster information for the dependency.
Monday, July 29, 13

As well as information about our SLAs.
Monday, July 29, 13
So, going back to the engineering diagram...
Monday, July 29, 13
If that same service fails today...
Monday, July 29, 13

We simply disconnect from that service.
And replace it with an appropriate fallback. The fallback, ideally is a slightly degrade, but useful offering. If we cannot get that, however, we will quickly provide a 5xx response which will help the systems shed load rather than queue things up (which could eventually cause the system as a whole to tip over).
This will keep our customers happy, even if the experience may be slightly degraded. It is important to note that different dependency libraries have different fallback scenarios. And some are more resilient than others. But the overall sentiment here is accurate at a high level.
System Infrastructure
Rather than relying on data centers, we have moved everything to the cloud! Enables rapid scaling with relative ease. Adding new servers, in new locations, take minutes. And this is critical when the service needs to grow from 1B requests a month to 2B requests a day in a relatively short period of time.
That is much more preferable for us than spending our time, money and energy in data centers, adding servers, dealing with power supplies, etc.
Instead, we spend time in tools such as Asgard, created by Netflix staff, to help us manage our instance types and counts in AWS. Asgard is available in our open source repository at github.
Another feature afforded to us through AWS to help us scale is Autoscaling. This is the Netflix API request rates over a span of time. The red line represents a potential capacity needed in a data center to ensure that the spikes could be handled without spending a ton more than is needed for the really unlikely scenarios.
Through autoscaling, instead of buying new servers based on projected spikes in traffic and having systems administrators add them to the farm, the cloud can dynamically and automatically add and remove servers based on need.
More than 36 Million Subscribers

Going global has a different set of scaling challenges. AWS enables us to add instances in new regions that are closer to our customers.
To help us manage our traffic across regions, as well as within given regions, we created Zuul. Zuul is open source in our github repository.
Zuul

- Multi-Region Resiliency
- Insights
- Stress Testing
- Canary Testing
- Dynamic Routing

- Load Shedding
- Security
- Static Response Handling
- Authentication

Monday, July 29, 13

Zuul does a variety of things for us. Zuul fronts our entire streaming application as well as a range of other services within our system.
Moreover, Zuul is the routing engine that we use for Isthmus, which is designed to marshall traffic between regions, for failover, performance or other reasons.
Forced Failure

Hystrix and other techniques throughout our engineering organization help keep things resilient. We also have an army of tools that introduce failures to the system which will help us identify problems before they become really big problems.
The army is the Simian Army, which is a fleet of monkeys who are designed to do a variety of things, in an automated way, in our cloud implementation. Chaos Monkey, for example, periodically terminates AWS instances in production to see how the system as a whole will respond once that server disappears. Latency Monkey introduces latencies and errors into a system to see how it responds. The system is too complex to know how things will respond in various circumstances, so the monkeys expose that information to us in a variety of ways. The monkeys are also available in our open source github repository.
Most companies focus on a small handful of device implementations, most notably Android and iOS devices.
At Netflix, we have more than 1,000 different device types that we support. Across those devices, there is a high degree of variability. As a result, we have seen inefficiencies and problems emerge across our implementations. Those issues also translate into issues with the API interaction.
For example, screen size could significantly affect what the API should deliver to the UI. TVs with bigger screens that can potentially fit more titles and more metadata per title than a mobile phone. Do we need to send all of the extra bits for fields or items that are not needed, requiring the device itself to drop items on the floor? Or can we optimize the delivery of those bits on a per-device basis?
Different devices have different controlling functions as well. For devices with swipe technologies, such as the iPad, do we need to pre-load a lot of extra titles in case a user swipes the row quickly to see the last of 500 titles in their queue? Or for up-down-left-right controllers, would devices be more optimized by fetching a few items at a time when they are needed? Other devices support voice or hand gestures or pointer technologies. How might those impact the user experience and therefore the metadata needed to support them?
Technical Capabilities

The technical specs on these devices differ greatly. Some have significant memory space while others do not, impacting how much data can be handled at a given time. Processing power and hard-drive space could also play a role in how the UI performs, in turn potentially influencing the optimal way for fetching content from the API. All of these differences could result in different potential optimizations across these devices.
Many UI teams needing metadata means many requests to the API team. In the one-size-fits-all API world, we essentially needed to funnel these requests and then prioritize them. That means that some teams would need to wait for API work to be done. It also meant that, because they all shared the same endpoints, we were often adding variations to the endpoints resulting in a more complex system as well as a lot of spaghetti code. Make teams wait due to prioritization was exacerbated by the fact that tasks took longer because the technical debt was increasing, causing time to build and test to increase. Moreover, many of the incoming requests were asking us to do more of the same kinds of customizations. This created a spiral that would be very difficult to break out of...
That variability ultimately caused us to do some introspection on our API layer.
Many other companies have seen similar issues and have introduced orchestration layers that enable more flexible interaction models.
Odata, HYQL, ql.io, rest.li and others are examples of orchestration layers. They address the same problems that we have seen, but we have approached the solution in a very different way.
Resource-Based API

vs.

Experience-Based API

We evolved our discussion towards what ultimately became a discussion between resource-based APIs and experience-based APIs.
Resource-Based Requests

- /users/<id>/ratings/title
- /users/<id>/queues
- /users/<id>/queues/instant
- /users/<id>/recommendations
- /catalog/titles/movie
- /catalog/titles/series
- /catalog/people

The original OSFA API was very resource oriented with granular requests for specific data, delivering specific documents in specific formats.
The interaction model looked basically like this, with (in this example) the PS3 making many calls across the network to the OSFA API. The API ultimately called back to dependent services to get the corresponding data needed to satisfy the requests.
In this mode, there is a very clear divide between the Client Code and the Server Code. That divide is the network border.
And the responsibilities have the same distribution as well. The Client Code handles the rendering of the interface (as well as asking the server for data). The Server Code is responsible of gathering, formatting and delivering the data to the UIs.
And ultimately, it works. The PS3 interface looks like this and was populated by this interaction model.
But we believe this is not the optimal way to handle it. In fact, assembling a UI through many resource-based API calls is akin to pointillism paintings. The picture looks great when fully assembled, but it is done by assembling many points put together in the right way.
We have decided to pursue an experience-based approach instead. Rather than making many API requests to assemble the PS3 home screen, the PS3 will potentially make a single request to a custom, optimized endpoint.
In an experience-based interaction, the PS3 can potentially make a single request across the network border to a scripting layer (currently Groovy), in this example to provide the data for the PS3 home screen. The call goes to a very specific, custom endpoint for the PS3 or for a shared UI. The Groovy script then interprets what is needed for the PS3 home screen and triggers a series of calls to the Java API running in the same JVM as the Groovy scripts. The Java API is essentially a series of methods that individually know how to gather the corresponding data from the dependent services. The Java API then returns the data to the Groovy script who then formats and delivers the very specific data back to the PS3.
We also introduced RxJava into this layer to improve our ability to handle concurrency and callbacks. RxJava is open source in our github repository.
In this model, the border between Client Code and Server Code is no longer the network border. It is now back on the server. The Groovy is essentially a client adapter written by the client teams.
Monday, July 29, 13

And the distribution of work changes as well. The client teams continue to handle UI rendering, but now are also responsible for the formatting and delivery of content. The API team, in terms of the data side of things, is responsible for the data gathering and hand-off to the client adapters. Of course, the API team does many other things, including resiliency, scaling, dependency interactions, etc. This model is essentially a platform for API development.
If resource-based APIs assemble data like pointillism, experience-based APIs assemble data like a photograph. The experience-based approach captures and delivers it all at once.
Scaling...
Again, the dependency chains in our system are quite complicated.
Monday, July 29, 13

That is a lot of change in the system!
Testing Philosophy:

Act Fast, React Fast

Monday, July 29, 13
As a result, our philosophy is to act fast (ie. get code into production as quickly as possible), then react fast (ie. response to issues quickly as they arise).
That Doesn’t Mean We Don’t Test

- Unit tests
- Functional tests
- Regression scripts
- Continuous integration
- Capacity planning
- Load / Performance tests

That said, we do spend a lot of time testing. We just don’t intend to make the system bullet-proof before deploying. Instead, we have employed some techniques to help us learn more about what the new code will look like in production. The key to it all is to automate everything, to get humans away from repeatable tasks.
Cloud-Based Deployment Techniques

Two such examples are canary deployments and what we call red/black deployments.
Monday, July 29, 13

The canary deployments are comparable to canaries in coal mines. We have many servers in production running the current codebase. We will then introduce a single (or perhaps a few) new server(s) into production running new code. Monitoring the canary servers will show what the new code will look like in production.
Canary Analysis Automation

The health of the canary is automated as well, comparing its metrics against the fleet of production servers.
If the canary encounters problems, it will register in any number of ways. The problems will be determined based on a comprehensive set of tools that will automatically perform health analysis on the canary.

Monday, July 29, 13

**Current Code**

In Production

Single Canary Instance

To Test New Code with Production Traffic

(around 1% or less of traffic)
If the canary shows errors, we pull it/them down, re-evaluate the new code, debug it, etc.
Monday, July 29, 13

We will then repeat the process until the analysis of canary servers look good.
Monday, July 29, 13

We will then repeat the process until the analysis of canary servers look good.
We also use Zuul to funnel varying degrees of traffic to the canaries to evaluate how much load the canary can take relative to the current production instances. If the RPS, for example, drops, the canary may fail the Zuul stress test.
If the new code looks good in the canary, we can then use a technique that we call red/black deployments to launch the code. Start with red, where production code is running. Fire up a new set of servers (black) equal to the count in red with the new code.
Then switch the pointer to have external requests point to the black servers. Sometimes, however, we may find an error in the black cluster that was not detected by the canary. For example, some issues can only be seen with full load.
Monday, July 29, 13

Then switch the pointer to have external requests point to the black servers. Sometimes, however, we may find an error in the black cluster that was not detected by the canary. For example, some issues can only be seen with full load.
Monday, July 29, 13
If a problem is encountered from the black servers, it is easy to rollback quickly by switching the pointer back to red. We will then re-evaluate the new code, debug it, etc.
Once we have debugged the code, we will put another canary up to evaluate the new changes in production.
Stress Test with Zuul

Monday, July 29, 13
And we will stress the canary again...
If the new code looks good in the canary, we can then bring up another set of servers with the new code.
Then we will switch production traffic to the new code.
Monday, July 29, 13

If everything still looks good, we disable the red servers and the new code becomes the new red servers.
All of the open source components discussed here, as well as many others, can be found at the Netflix github repository.

https://www.github.com/Netflix
Scaling the Netflix API

Help Wanted!

Daniel Jacobson
@daniel_jacobson
http://www.linkedin.com/in/danieljacobson
http://www.slideshare.net/danieljacobson

Monday, July 29, 13