Zuul 2 : The Netflix Journey to Asynchronous, Non-Blocking Systems

Netflix Technology Blog 8 min read · Sep 21, 2016

We recently made a major architectural change to Zuul, our cloud gateway. Did anyone even notice!?

Probably not… Zuul 2 does the same thing that its predecessor did — acting as the front door to Netflix's server infrastructure, handling traffic from all Netflix users around the world. It also routes requests, supports developers' testing and debugging, provides deep insight into our overall service health, protects Netflix from attacks, and channels traffic to other cloud regions when an AWS region is in trouble. The major architectural difference between Zuul 2 and the original is that Zuul 2 is running on an asynchronous and non-blocking framework, using Netty. After running in production for the last several months, the primary advantage (one that we expected when embarking on this work) is that it provides the capability for devices and web browsers to have persistent connections back to Netflix at Netflix scale. With more than 83 million members, each with multiple connected devices, this is a massive scale challenge. By having a persistent connection to our cloud

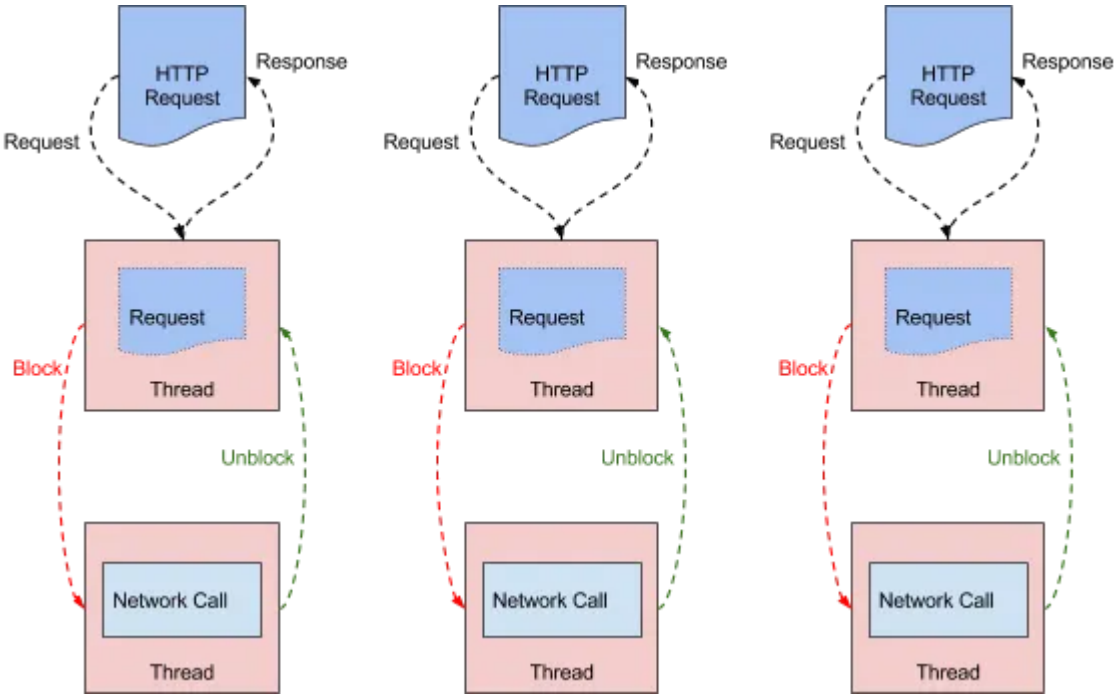
infrastructure, we can enable lots of interesting product features and innovations, reduce overall device requests, improve device performance, and understand and debug the customer experience better.

We also hoped the Zuul 2 would offer resiliency benefits and performance improvements, in terms of latencies, throughput, and costs. But as you will learn in this post, our aspirations have differed from the results.

Differences Between Blocking vs. Non-Blocking Systems

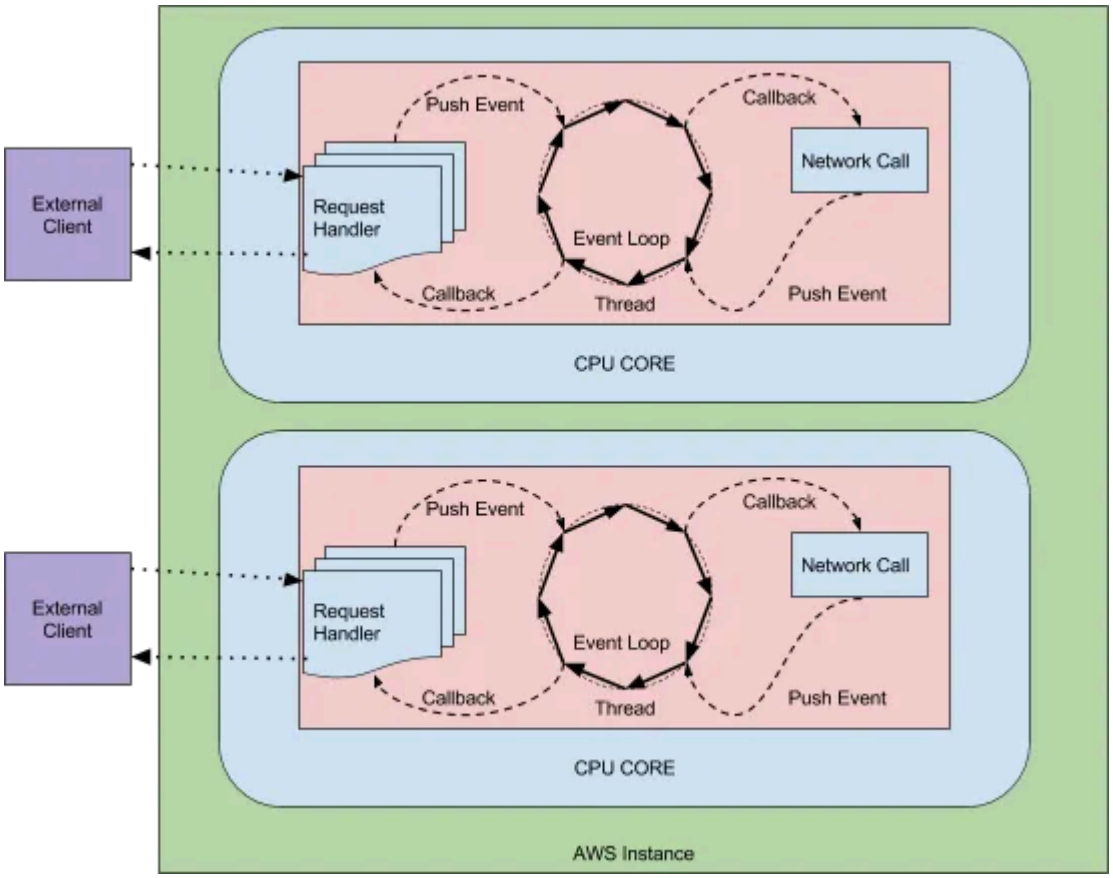
To understand why we built Zuul 2, you must first understand the architectural differences between asynchronous and non-blocking ("async") systems vs. multithreaded, blocking ("blocking") systems, both in theory and in practice.

Zuul 1 was built on the Servlet framework. Such systems are blocking and multithreaded, which means they process requests by using one thread per connection. I/O operations are done by choosing a worker thread from a thread pool to execute the I/O, and the request thread is blocked until the worker thread completes. The worker thread notifies the request thread when its work is complete. This works well with modern multi-core AWS instances handling 100's of concurrent connections each. But when things go wrong, like backend latency increases or device retries due to errors, the count of active connections and threads increases. When this happens, nodes get into trouble and can go into a death spiral where backed up threads spike server loads and overwhelm the cluster. To offset these risks, we built in throttling mechanisms and libraries (e.g., Hystrix) to help keep our blocking systems stable during these events.



Multithreaded System Architecture

Async systems operate differently, with generally one thread per CPU core handling all requests and responses. The lifecycle of the request and response is handled through events and callbacks. Because there is not a thread for each request, the cost of connections is cheap. This is the cost of a file descriptor, and the addition of a listener. Whereas the cost of a connection in the blocking model is a thread and with heavy memory and system overhead. There are some efficiency gains because data stays on the same CPU, making better use of CPU level caches and requiring fewer context switches. The fallout of backend latency and “retry storms” (customers and devices retrying requests when problems occur) is also less stressful on the system because connections and increased events in the queue are far less expensive than piling up threads.



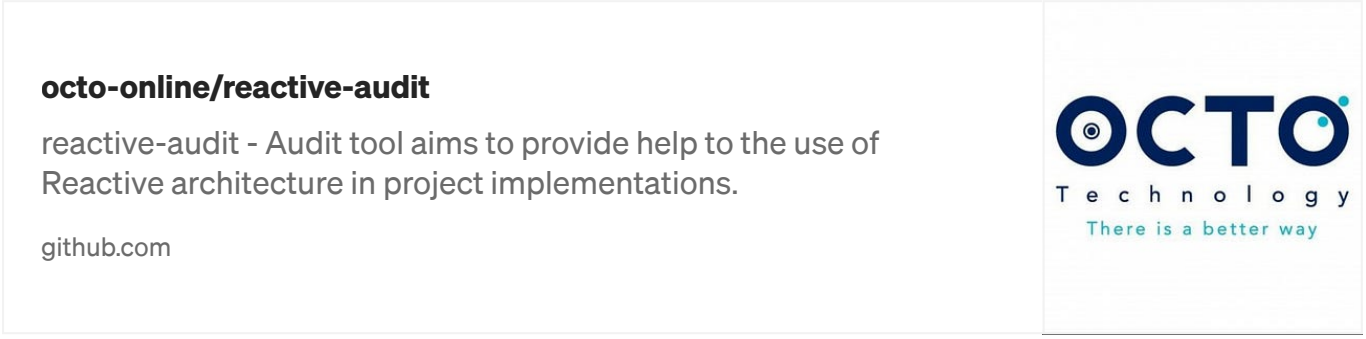
Asynchronous and Non-blocking System Architecture

The advantages of async systems sound glorious, but the above benefits come at a cost to operations. Blocking systems are easy to grok and debug. A thread is always doing a single operation so the thread's stack is an accurate snapshot of the progress of a request or spawned task; and a thread dump can be read to follow a request spanning multiple threads by following locks. An exception thrown just pops up the stack. A "catch-all" exception handler can cleanup everything that isn't explicitly caught.

Async, by contrast, is callback based and driven by an event loop. The event loop's stack trace is meaningless when trying to follow a request. It is difficult to follow a request as events and callbacks are processed, and the tools to help with debugging this are sorely lacking in this area. Edge cases, unhandled exceptions, and incorrectly handled state changes create dangling resources resulting in ByteBuf leaks, file descriptor leaks, lost responses, etc. These types of issues have proven to be quite difficult to debug because it is difficult to know which event wasn't handled properly or cleaned up appropriately.

Building Non-Blocking Zuul

Building Zuul 2 within Netflix's infrastructure was more challenging than expected. Many services within the Netflix ecosystem were built with an assumption of blocking. Netflix's core networking libraries are also built with blocking architectural assumptions; many libraries rely on thread local variables to build up and store context about a request. Thread local variables don't work in an async non-blocking world where multiple requests are processed on the same thread. Consequently, much of the complexity of building Zuul 2 was in teasing out dark corners where thread local variables were being used. Other challenges involved converting blocking networking logic into non-blocking networking code, and finding blocking code deep inside libraries, fixing resource leaks, and converting core infrastructure to run asynchronously. There is no one-size-fits-all strategy for converting blocking network logic to async; they must be individually analyzed and refactored. The same applies to core Netflix libraries, where some code was modified and some had to be forked and refactored to work with async. The open source project Reactive-Audit was helpful by instrumenting our servers to discover cases where code blocks and libraries were blocking.



We took an interesting approach to building Zuul 2. Because blocking systems can run code asynchronously, we started by first changing our Zuul Filters and filter chaining code to run asynchronously. Zuul Filters contain the specific logic that we create to do our gateway functions (routing, logging, reverse proxying, ddos prevention, etc). We refactored core Zuul, the base Zuul Filter classes, and our Zuul Filters using RxJava to allow them to run asynchronously. We now have two types of filters that are used together: async used for I/O operations, and a sync filter that run logical

operations that don't require I/O. Async Zuul Filters allowed us to execute the exact same filter logic in both a blocking system and a non-blocking system. This gave us the ability to work with one filter set so that we could develop gateway features for our partners while also developing the Netty-based architecture in a single codebase. With async Zuul Filters in place, building Zuul 2 was "just" a matter of making the rest of our Zuul infrastructure run asynchronously and non-blocking. The same Zuul Filters could just drop into both architectures.

Results of Zuul 2 in Production

Hypotheses varied greatly on benefits of async architecture with our gateway. Some thought we would see an order of magnitude increase in efficiency due to the reduction of context switching and more efficient use of CPU caches and others expected that we'd see no efficiency gain at all. Opinions also varied on the complexity of the change and development effort.

So what did we gain by doing this architectural change? And was it worth it? This topic is hotly debated. The Cloud Gateway team pioneered the effort to create and test async-based services at Netflix. There was a lot of interest in understanding how microservices using async would operate at Netflix, and Zuul looked like an ideal service for seeing benefits.

While we did not see a significant efficiency benefit in migrating to async and non-blocking, we did achieve the goals of connection scaling. Zuul does benefit by greatly decreasing the cost of network connections which will enable push and bi-directional communication to and from devices. These

features will enable more real-time user experience innovations and will reduce overall cloud costs by replacing “chatty” device protocols today (which account for a significant portion of API traffic) with push notifications. There also is some resiliency advantage in handling retry storms and latency from origin systems better than the blocking model. We are continuing to improve on this area; however it should be noted that the resiliency advantages have not been straightforward or without effort and tuning.

With the ability to drop Zuul’s core business logic into either blocking or async architectures, we have an interesting apples-to-apples comparison of blocking to async. So how do two systems doing the exact same real work, although in very different ways, compare in terms of features, performance and resiliency? After running Zuul 2 in production for the last several months, our evaluation is that the more CPU-bound a system is, the less of an efficiency gain we see.

We have several different Zuul clusters that front origin services like API, playback, website, and logging. Each origin service demands that different operations be handled by the corresponding Zuul cluster. The Zuul cluster that fronts our API service, for example, does the most on-box work of all our clusters, including metrics calculations, logging, and decrypting incoming payloads and compressing responses. We see no efficiency gain by swapping an async Zuul 2 for a blocking one for this cluster. From a capacity and CPU point of view they are essentially equivalent, which makes sense given how CPU-intensive the Zuul service fronting API is. They also tend to degrade at about the same throughput per node.

The Zuul cluster that fronts our Logging services has a different performance profile. Zuul is generally receiving logging and analytics messages from devices and is write-heavy, so requests are large, but responses are small and not encrypted by Zuul. As a result, Zuul is doing much less work for this cluster. While still CPU-bound, we see about a 25%

increase in throughput corresponding with a 25% reduction in CPU utilization by running Netty-based Zuul. We thus observed that the less work a system actually does, the more efficiency we gain from async.

Overall, the value we get from this architectural change is high, with connection scaling being the primary benefit, but it does come at a cost. We have a system that is much more complex to debug, code, and test, and we are working within an ecosystem at Netflix that operates on an assumption of blocking systems. It is unlikely that the ecosystem will change anytime soon, so as we add and integrate more features to our gateway it is likely that we will need to continue to tease out thread local variables and other assumptions of blocking in client libraries and other supporting code. We will also need to rewrite blocking calls asynchronously. This is an engineering challenge unique to working with a well established platform and body of code that makes assumptions of blocking. Building and integrating Zuul 2 in a greenfield would have avoided some of these complexities, but we operate in an environment where these libraries and services are essential to the functionality of our gateway and operation within Netflix's ecosystem.

We are in the process of releasing Zuul 2 as open source. Once it is released, we'd love to hear from you about your experiences with it and hope you will share your contributions! We plan on adding new features such as http/2 and websocket support to Zuul 2 so that the community can also benefit from these innovations.

— The Cloud Gateway Team (Mikey Cohen, Mike Smith, Susheel Aroskar, Arthur Gonigberg, Gayathri Varadarajan, and Sudheer Vinukonda)

Netflix/zuul

zuul - Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.

github.com

. . .

Originally published at techblog.netflix.com on September 21, 2016.

Zuul | Open Source | Cloud | Cloud Gateway