

---

# System Design: Lessons From Netflix's Notification Service Design

Author: Ravi Tandon

---

Netflix, the widely popular video-streaming platform, released its platform for managing and scaling its notification service. This blog post will discuss the architecture of the system's design and the lessons that they learned having built and scaled the system. They named their system **Zuul**. Here are a couple of useful links:

- [Link to the YouTube talk](#)
- [Link to the Github Repository](#)

Alright. Let's get started, then!

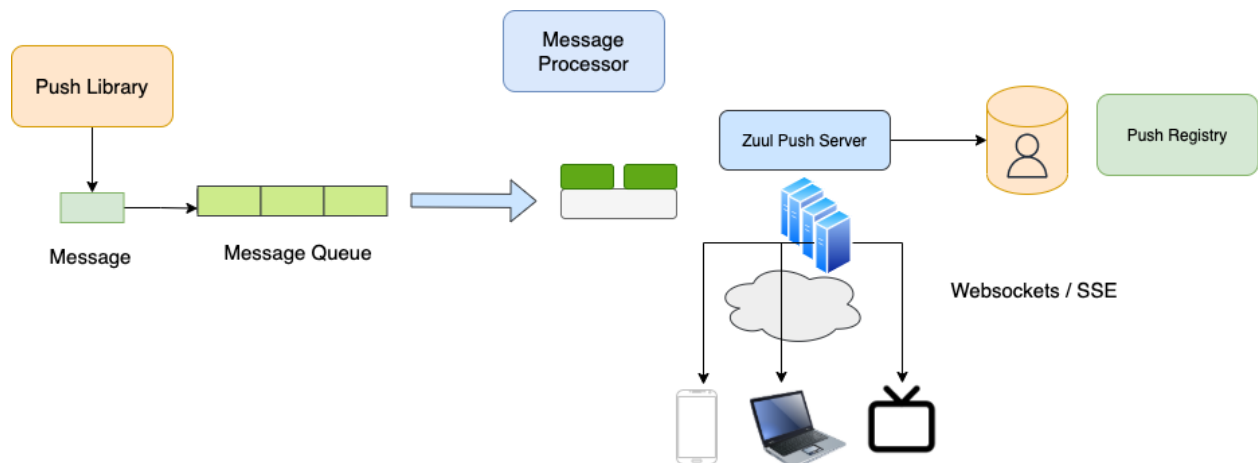
---

## High-Level Architecture

The basic components of the Zuul Push Architecture are as follows:

1. Zuul Push Servers
2. Push Registry
3. Message Processor
4. Push Message Queue
5. Push Library
6. WebSockets/SSE

Zuul is not a single service but a complete infrastructural platform made of multiple components. *Zuul Push Servers* sit on the network edge and accept client connections.

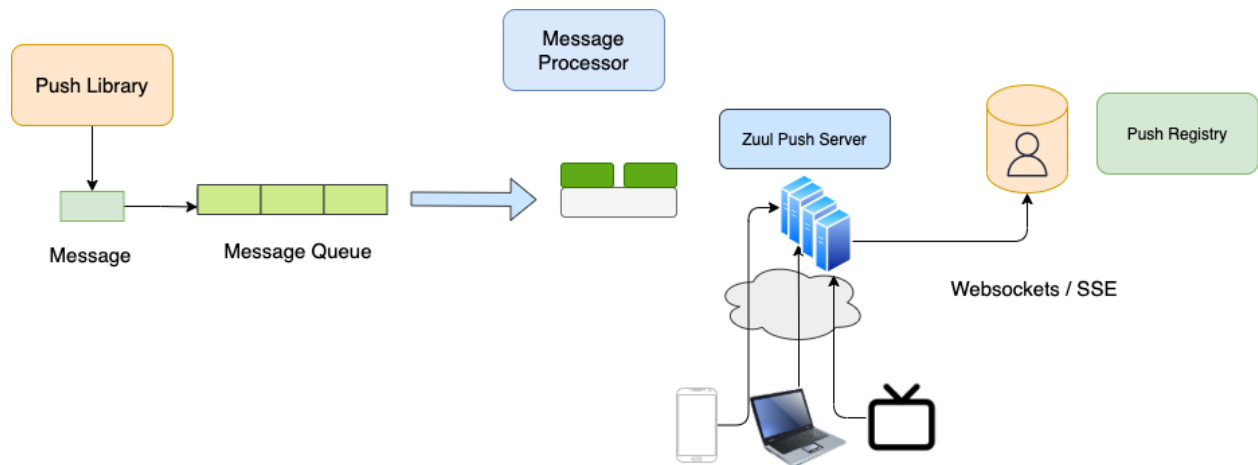


The clients connect to servers using either *WebSockets* or *SSE*. The Websocket connections are persistent and async in nature. Websockets enable the clients to push updates from the edge to the Zuul Servers. Each of the clients connects to a Zuul server. A *Push Registry* is built to keep track of which client is connected to which server. Senders use the Push Library to send messages to clients. The call that they use is a single, asynchronous send message call. The message is sent to a push message queue. A *Message Processor* handles the messages, finds the server to which a message must be delivered, and sends the message to the Push Server.

### What are WebSockets?

[WebSockets](#), developed by Michael Carter and Ian Hickson, essentially is a thin transport layer built on top of a device's TCP/IP stack. They provide a TCP communication layer to web applications that are as close to raw as possible, without the developer dealing with raw sockets. They enable a two-way interactive communication session between a client (on the user's browser, mobile, or Television, etc.) and a server. With this API, a client can send messages to a server and receive event-driven responses without polling the server for a reply. This is a [good article](#) on the differences between WebSockets and the HTTP Protocol.

**Use Case Of WebSockets:** The emergence of WebSockets was a need to support duplex communication between the client and the server. Let's take the example of Netflix. You are watching a movie, and the movie starts to crash in between. With a traditional HTTP connection, the server will need to poll the client to check for regular updates. It makes the entire process resource-intensive. The client establishes a connection with the server, creates a connection, and then closes the connection. The process, if done repeatedly for thousands of concurrent users, can consume a large set of resources.



## Component Diagram In Detail

This section discusses the design and implementation of the different components with the Zuul Notification System.

### Zuul Cluster

The Zuul Cluster can handle an aggregate of 10M concurrent connections at peak. It is based on the Zuul API Gateway. The Zuul API Gateway can support more than 1 Million Requests Per Second. **“The Cloud Gateway team at Netflix runs and operates more than 80 clusters of Zuul 2, sending traffic to about 100 (and growing) backend service clusters which amount to more than 1 million requests per second.”** The underlying design is non-blocking and supports asynchronous connections. [This blog](#) discusses the design of the Zuul API Gateway in detail. Before we go into the detailed design, let’s discuss why the system designers preferred an asynchronous model.

### Disadvantages With A Synchronous Approach

In the traditional synchronous model, the server would allocate a different thread per connection. This design would quickly exhaust all the resources (CPU and Memory) and wouldn’t be able to scale to meet even 10,000 concurrent connections. In this thread, each server maintains its thread stack that consumes memory. Besides, the 10,000 different threads that run will consume most of the CPU, albeit idle, and waste a lot of the resources.

Zuul 2 uses [Netty](#) as the framework for the development of its client-server applications. It is an open-source, asynchronous event-driven network application framework for the rapid development of maintainable high-performance protocol servers & clients. Netty is used by Cassandra and Hadoop platforms and has been well tested in the wild.

## Push Registry

The push registry is a data store that stores mapping from each client to the server to which it is connected. Any specific data store can be plugged into it. The characteristics of the required data store are:

- Low read latency; the write latency can be high
- Auto-record expiry - prevents phantom registration records
  - Each connection in the Zuul server creates an entry in the registry when a new client connects
  - If the connection drops and fails to clean up the registry, we end up with what are called “*phantom registries*.”
  - One of the ways to solve this problem is to have a system for automatic expiration in place. This can be done by associating a TTL with each entry in the Push Registry.
- The data store should support sharding for improvement of performance and replication for fault-tolerance.
- Redis, Cassandra, Amazon’s Dynamo DB all satisfy these requirements.
- The Netflix team eventually went on to build a system called *Dynomite* over Redis. Dynomite supports auto-sharding, a quorum for reads and writes, and cross-region replication for fault tolerance.

## Message Processing

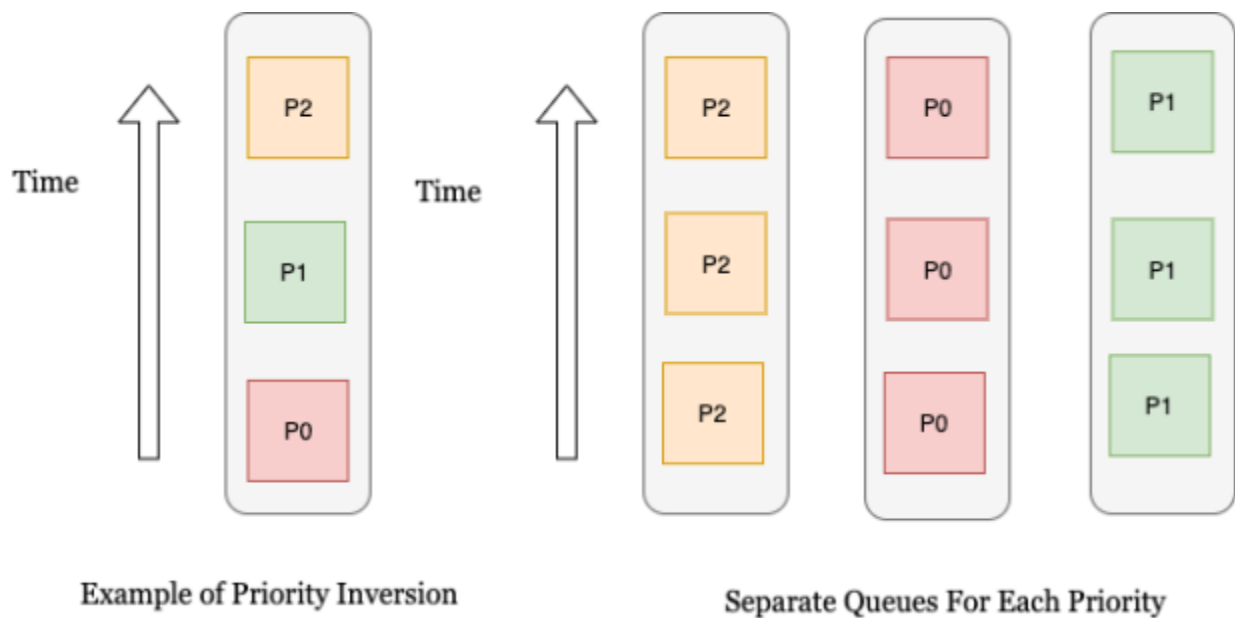
The Netflix team decided to go with Apache’s Kafka platform for message queuing. Refer to our discussion on the tradeoffs of using Apache Kafka over other services such as RabbitMQ and Amazon’s SQS in a [previous blog](#). The message senders use the message bus in two different ways.

**Fire & Forget**: Some of the senders adopt the “*fire and forget*” approach. In this approach, the senders send the message on the queue and do not check back on

whether the message has been delivered to the client. This could use cases where a weak consistency model may work well instead of gaining some performance. An example could be a request for collecting likes on a video. Eventual consistency will work.

**Subscription Based:** Some of the senders adopt a stricter model, where they subscribe to updates from the Push Registry to ensure that the message is delivered and received by the client. The sender can then do a re-try in case the client does not receive the message. This is a strict consistency model. Examples of use-cases can be sending updates (such as software upgrades, pushing new content to the clients, etc.). The status is maintained in the “*Status Queue*.” The “*message queue*” is replicated across regions to ensure that queue remains fault-tolerant.

To prevent priority inversion ([video](#), [blog](#)), a separate queue is maintained for each priority. In the image below, we can see how having individual threads for each priority removes the problem of priority inversion.



To scale up the message processor service, multiple instances of the service are run in parallel.

## Operational Lessons

The most important lessons for any system are learned from running the system in an actual production setting. This tends to be the most exciting and relevant part of the process of a system's design.

### **Persistent Connections Make Zuul Push Server Stateful**

In Zuul, the clients make persistent connections with the Zuul servers. These tend to be long-lived and stable connections. While they are great for a system's overall efficiency, they tend to be terrible for operations. Let's take a few examples:

**Rollbacks are painful:** Let's take the example of a bug being pushed in production. Let's imagine that the fix requires a new software push and resetting connections from the clients. It could be a change in the TCP stack and old connections may still be running with the bug. Now, in order to achieve this, all the old connections will need to be reset. In order to achieve that a manual process must be triggered to reset connections from the servers. In addition, all the push registries need to be updated and can cause bugs in the system

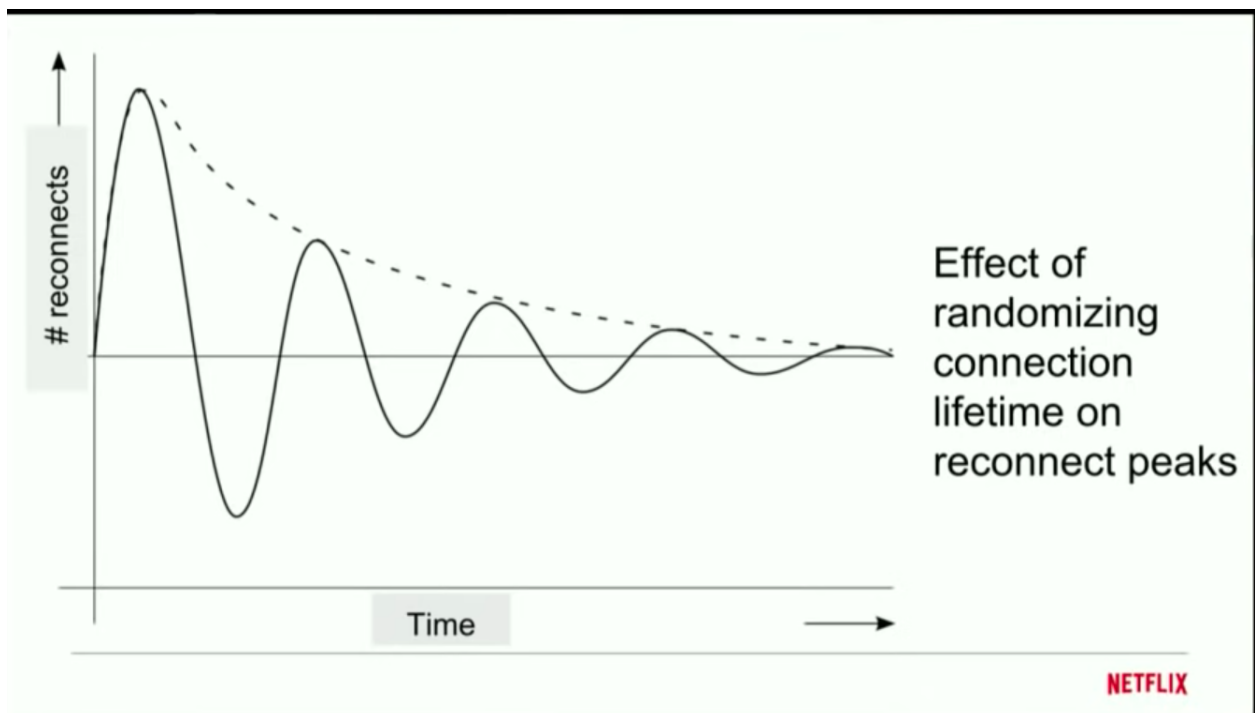
**Migrations are painful:** Migration of servers is very, very tricky. In addition to operational issues (mentioned above), it can lead to the problem of the *thundering herd problem*. Let's understand this better. Let's say there are 10,000 active connections on server 1. Now, we need to migrate these to server 2 because we want to reset server 1. If we do it in one go, server 2 will get 10,000 requests over a short period of time, the server will get overwhelmed and the clients will see a very sluggish performance.

I think, we now get a sense of how the performance of the system can degrade :)

### **How did they solve the problem?**

Here comes the exciting part. There are two interesting engineering tricks that the team used. These are as follows:

- a) **Terminate connections periodically**: By terminating connections periodically, they ensured that a client doesn't stick to one single server. It is pretty much at the root of the *thundering herd* problem. Now, when the client disconnects and reconnects, it will most probably connect to a new server. If the server is buggy or not very performant, the load balancer can route requests away from the server and balance the load incrementally and not in a single go.
- b) **Randomize each connection's lifetime**: In addition to terminating connections periodically, they also ensured that each client's connections has a different lifetime. The impact was dampening the number of reconnects per unit time. Instead of each of the 10,000 connections hitting and reconnecting through the load-balancer simultaneously, the connections would dampen out and reconnect at different timestamps.





**Lessons:** To avoid the problem of *thundering herds*:

- Terminate connections periodically.
- Randomize the lifetime of each connection.

### **Optimizing Push Server**


Another observation that the team had was that most connections to the servers remained idle over the lifetime of the connections. The seemingly optimal approach was to keep a single beefy server with all the connections crammed to it. Why the approach? They believed that fewer servers would make the operational cost low as the IT team would have to maintain fewer servers. However, the design fails in case of server failures. A failure of a single beefy server would lead to many reconnects. It gave rise to a thundering herd in case of a server failure and can cause a stampede in your server. Solution: [Goldilocks strategy](#). *“The Goldilocks principle is named by analogy to the children's story "The Three Bears", in which a young girl named Goldilocks tastes three different bowls of porridge and finds she prefers porridge that is neither too hot nor too cold but has just the right temperature.”* The team decided to go with a strategy where the server size was neither too large, nor too small. They ended up using Amazon's *m4 large*.

**Lesson:** Keeping costs the same, a high number of small servers is better than a low number of big servers.

### **Auto-Scale Server**

Typically, systems with built-in elasticity scale well. Elasticity is the ability of a system to scale on resources dynamically and adapt to the incoming traffic load. In the case of Netflix's system, they needed to take a decision on which metrics to auto-scale the system on? Should they go with RPS (i.e. requests per second), or CPU usage? Both the metrics, in this case, seemed not to be useful. Given that the threads remain idle for most of the time, CPU is not a great metric. Similarly, given that the connections remain persistent, requests/second isn't useful as well. They went with the number of open connections as the metric to optimize bind elasticity.

## References

1. [The WebSocket API \(WebSockets\) - Web APIs | MDN](#)
2.  [Scaling Push Messaging for Millions of Devices @Netflix](#)
3. [Zuul Github Repository](#)