highscalability.com

How Urban Airship Scaled to 2.5 Billion Notifications During the U.S. Election - High Scalability -

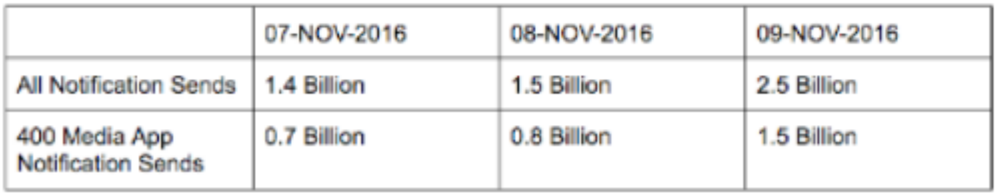
12-15 minutes

This is a guest post by Urban Airship. Contributors: Adam Lowry, Sean Moran, Mike Herrick, Lisa Orr, Todd Johnson, Christine Ciandrini, Ashish Warty, Nick Adlard, Mele Sax-Barnett, Niall Kelly, Graham Forest, and Gavin McQuillan

Urban Airship is trusted by thousands of businesses looking to grow with mobile. Urban Airship is a seven year old SaaS company and has a freemium business model so you can try it for free. For more information, visit www.urbanairship.com. Urban Airship now averages more than one billion push notifications delivered daily. This post highlights Urban Airship notification usage for the 2016 U.S. election, exploring the architecture of the system--the Core Delivery Pipeline--that delivers billions of real-time notifications for news publishers.

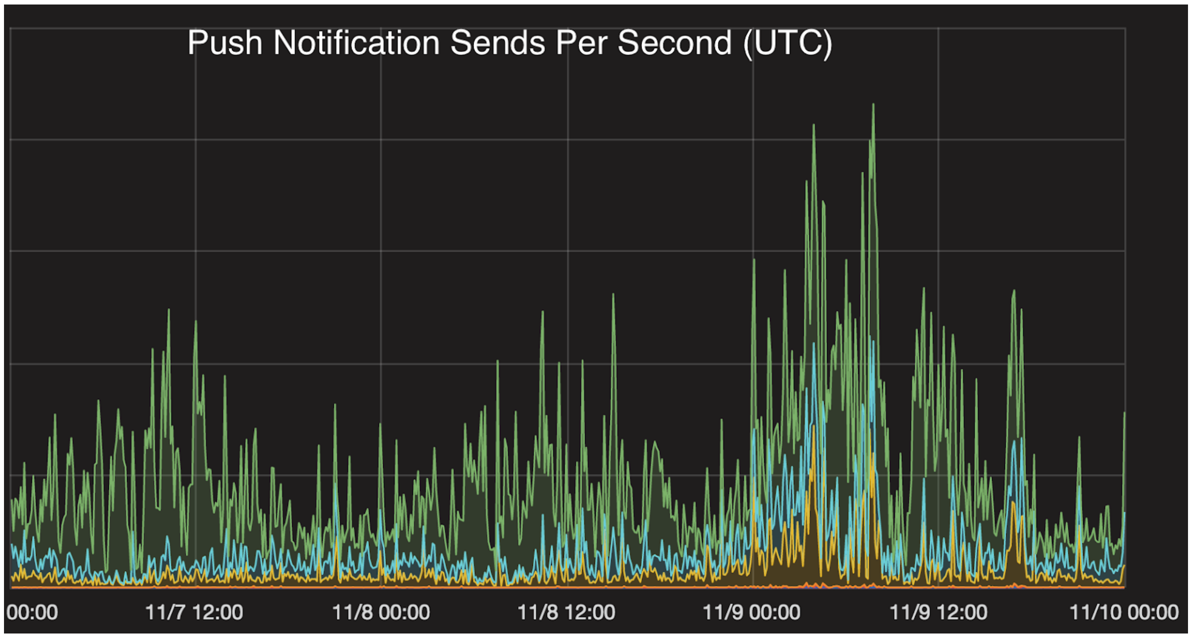
2016 U.S. Election

In the 24 hours surrounding Election Day, Urban Airship delivered 2.5 billion notifications—its highest daily volume ever. This is equivalent to 8 notification per person in the United States or 1 notification for every active smartphone in the world. While Urban Airship powers more than 45,000 apps across every industry vertical, analysis of the election usage data shows that more than 400 media apps were responsible for 60% of this record volume, sending 1.5 billion notifications in a single day as election results were tracked and reported.

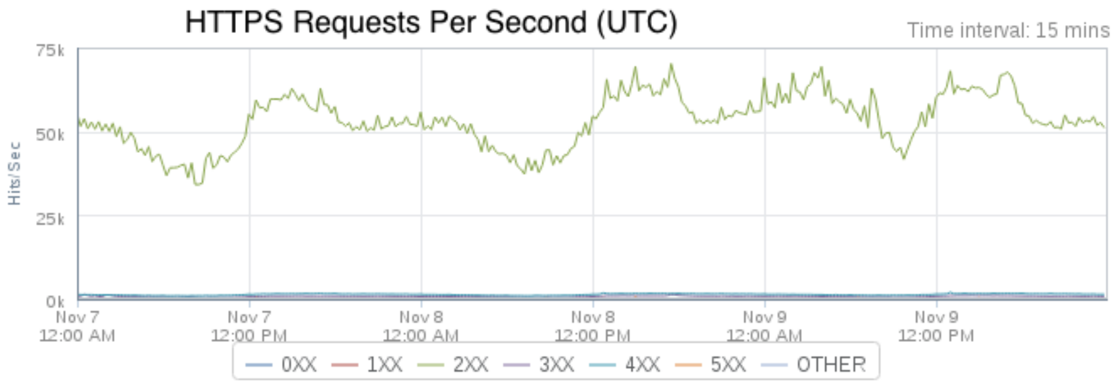


	07-NOV-2016	08-NOV-2016	09-NOV-2016
All Notification Sends	1.4 Billion	1.5 Billion	2.5 Billion
400 Media App Notification Sends	0.7 Billion	0.8 Billion	1.5 Billion

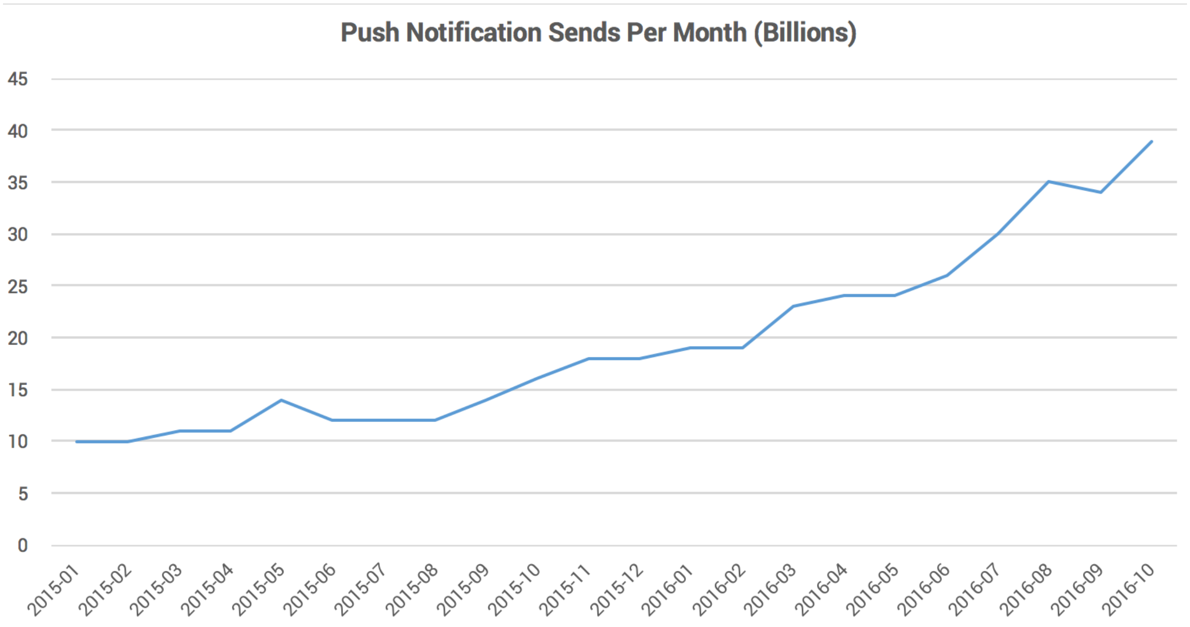
Notification volume was steady and peaked when the presidential election concluded.



HTTPS ingress traffic to the Urban Airship API peaked at nearly 75K per second during the election. Most of this traffic comes from the Urban Airship SDK communicating with the Urban Airship API.



Push notification volume has been rapidly increasing. Key recent drivers have been Brexit, the Olympics, and the U.S. election. October 2016 monthly notification volume has increased 150% year over year.



Core Delivery Pipeline Architecture Overview

The Core Delivery Pipeline (CDP) is the Urban Airship system responsible for materializing device addresses from audience selectors, and delivering notifications. Low latency is expected for all the notifications we send, whether they go to tens of millions of users simultaneously, target multiple complex sub-segments, contain personalized content, or anything in between. Here’s an overview of our architecture, and some of the things we’ve learned along the way.

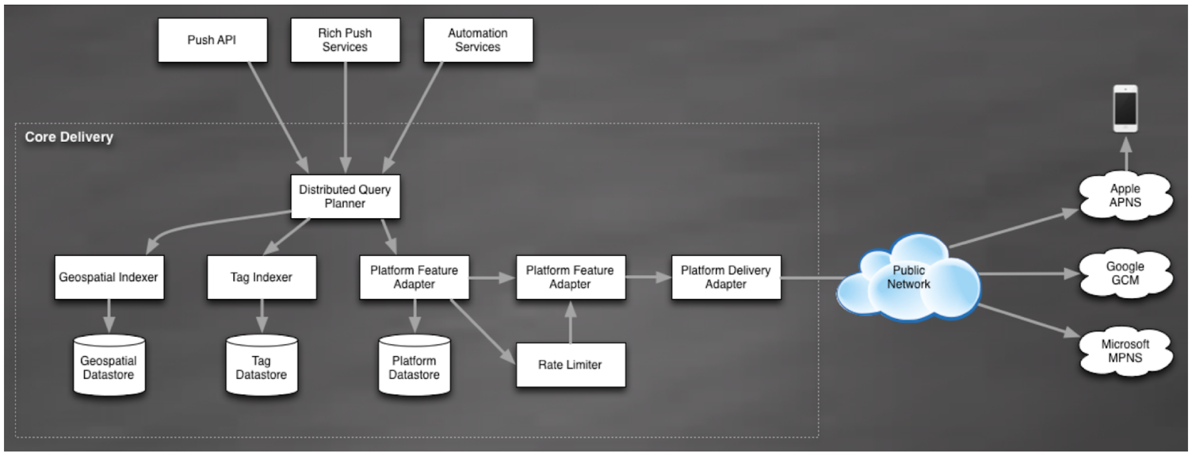
How We Started

What initially started in 2009 as a webapp and some workers has transformed into a service-oriented architecture. As pieces of the old system began to run into scale issues, we extracted them into one or more new services that were designed to satisfy the same feature set, but at a larger scale with better performance. Many of our original APIs and workers were written in Python, which we extracted into high-concurrency Java services. Where we originally stored device data in a set of Postgres shards, our scale quickly outpaced our capacity to add new shards, so we moved to a multiple database architecture using HBase and Cassandra.

The CDP is a collection of services that handle segmentation and push notification delivery. These services provide the same type of data in response to requests, but each service has that data indexed in a very different way for performance reasons. For example, we have a system that is responsible for handling broadcast messages, delivering the same notification payload to every device registered to the associated application. This service and its underlying datastore are designed very differently than the services we have that is responsible for delivering notifications based on location or user profile attributes.

We consider any long-lived process a service. These long-lived processes follow a general template regarding metrics, configuration, and logging for ease of deployment and operation. Typically our services fall into one of two groups: RPC services, or consumer services. RPC services provide commands to synchronously interact with the service using an in-house library very similar to GRPC, while consumer services process messages off of Kafka streams and perform service-specific operations on those messages.

Urban Airship Core Delivery High Level Architecture



Databases

To meet our performance and scale requirements we rely heavily on HBase and Cassandra for our data storage needs. While HBase and Cassandra are both columnar NoSQL stores, they have very different tradeoffs that influence which store we use and for what purpose.

HBase is very good at high-throughput scans where the expected cardinality of the response is very high, while Cassandra is good at lower cardinality lookups where the response is expected to contain just a handful of results. Both allow for high volumes of write throughput, which is a requirement for us, because all metadata updates from users' phones are applied in real time.

Their failure characteristics differ as well. HBase favors consistency and partition tolerance in the event of failure, while Cassandra favors availability and partition tolerance. Each of the CDP services has a very specific use case and as such has a highly specialized schema designed to facilitate the required access pattern as well as limiting the storage footprint. As a general rule, each database is only accessed by a single service, which is responsible for providing database access to other services via a less specialized interface.

Enforcing this 1:1 relationship between service and its backing database has a number of benefits.

- By treating the backing datastore of a service as an implementation detail, and not a shared resource, we gain flexibility.

- We can adjust the data model for a service while only changing that service's code.
- Usage tracking is more straightforward, which makes capacity planning easier.
- Troubleshooting is easier. Sometimes an issue lies with the service code, other times it is a backing database issue. Having the service and database be a logical unit vastly simplifies the troubleshooting process. We don't have to wonder "Who else could be accessing this database to make it behave in such a way?" Instead, we can rely on our application-level metrics from the service itself and only worry about one set of access patterns.
- Because there is only one service interacting with a database, we can perform nearly all of our maintenance activities without taking downtime. Heavy maintenance tasks become a service-level concern: data repair, schema migration and even switching to a completely different database can be done without disrupting service.

  It's true that there can be some performance tradeoffs when breaking out applications into smaller services.

However, we've found that the flexibility we gain in meeting our high scalability and high availability requirements more than worth it.

Data Modeling

Most of our services deal with the same data, just in different formats. Everything has to be consistent. To keep all of these services' data up to date we rely heavily on Kafka. Kafka is extremely fast and is also durable. Speed comes with certain tradeoffs. Kafka messages are only guaranteed to be sent at least once, and they aren't guaranteed to arrive in order.

How do we deal with this? We've modeled all mutation paths to be commutative: operations can be applied in any order and end up with the same result. They're also idempotent. This has a nice side-effect that we can replay Kafka streams for one-off data repair jobs, backfills, or even migrations.

To do this we take advantage of the concept of a "cell version," which exists in both HBase and Cassandra. Typically this is a timestamp, but it can be any number you'd like (there are some exceptions; for example, MAX_LONG can cause some strange behavior depending on your version of HBase or Cassandra and how your schema deals with deletes).

For us, the general rule for these cells is that they can have multiple versions, and the way we order versions is by their provided timestamp. With this behavior in mind, we can decompose our incoming messages into a specific set of columns, and combine that layout with custom application logic for tombstoning, taking into account the timestamp. That allows blind writes to the underlying datastore while maintaining the integrity of our data.

Just blindly writing changes to both Cassandra and HBase isn’t without its issues. A great example is the case of repeated writes of the same data in the event of a replay. While the state of the data won’t change due to the effort we put in to make the records idempotent, the duplicate data will have to be compacted out. In the most extreme cases, these extra records can cause significant compaction latency and backup. Because of this detail, we monitor our compaction times and queue depths closely as getting behind on compaction in both Cassandra and HBase can cause serious problems.

By ensuring messages from the stream follow a strict set of rules, and designing the consuming service to expect out-of-order and repeated messages, we can keep a very large number of disparate services in sync with only a second or two of lag on updates.

Service Design

Most of our services are written in Java, but in a very opinionated and modern style. We have a set of general guidelines that we take into consideration when designing a Java service:

- **Do one thing, do it well** - When designing a service, it should have a single responsibility. It is up to the implementer to decide what's included in a responsibility, but she or he will need to be ready to justify in a code review situation.
- **No shared operational state** - When designing a service, assume there will always be at least three instances of it running. A service needs to be able to handle the same exact request any other instance can without any external coordination. Those familiar with Kafka will note that Kafka consumers externally coordinate partition ownership for a topic:group pair. This guideline refers to service specific external coordination, not utilizing libraries or clients that may coordinate externally under the covers.

- **Bound your queues** - We use queues all over our services, they're a great way to batch up requests and fan things out to workers that will end up doing tasks that block externally. All queues should be bounded. Bounding queues does raise a number of questions, however:
- What happens to producers when the queue is full? Should they block? Except? Drop?
- How big should my queue be? To answer this question it helps to assume the queue is always full.
- How do I shut down cleanly?
- Each service will have a different answer for these questions depending on the exact use case.
- **Name custom thread pools and register an UncaughtExceptionHandler** - If we end up creating our own thread pool, we use the constructor or helper method from Executors to allow us to provide a ThreadFactory. With that ThreadFactory we can properly name our threads, set their daemon state, and register an UncaughtExceptionHandler to handle the case where an exception makes it to the top of the stack. These steps make debugging our service much easier, and spare us some late-night frustration.

- **Prefer immutable data objects over mutable state** - In a highly concurrent environment, mutable state can be dangerous. In general we use immutable data objects that can be passed between internal subsystems and queues. Having immutable objects be the main form of communication between subsystems makes designing for concurrent usage much more straightforward, and makes troubleshooting easier.

Where Do We Go From Here?

With Urban Airship's ability to send notifications through mobile wallet passes, its new support for web notifications and Apple News notifications, and its open channels capability to send notifications to any platform, device or marketing channel, we anticipate notification volume will grow exponentially. To meet this demand, we are continuing to invest heavily in our Core Delivery Pipeline architecture, services, databases, and infrastructure. To learn more about our technology and where we are headed please see GitHub, developer resources, documentation, and our jobs page.