SE 464
Week 5

Case Study: Push Notifications

Push Notifications

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
[Lecture 12: Push Notifications]

Last Time: Architecture Example

- Showed National Gun Violence Memorial architecture design case study.
- It's another article publishing system, so arch is like Wikipedia.
  - Caching and load balancers on frontend,
  - Stateless app,
  - SQL DB with read-replicas.
  - S3 file store was used for large media files (photos).
- Like Wikipedia, the design scales.

Limitations of Client-Server Architectures

- So far, everything we have talked about is a Client-Server architecture.
- Client (web browser, smartphone app, desktop app) makes requests, and the Server gives responses.
- The client starts all interactions. For example:
  - User clicks web link or app button
  - Javascript running on browser makes a REST request.
- In what situation would a **server** should start an interaction?
  - Deliver a WhatsApp message to a user.
  - Notify an Uber customer that their driver has arrived.
  - Notify an Ebay user that they were outbid.
  - Notify an Ebay user that they won an auction.

Caused by another user's action.

Email is a simple way to send notifications

- Many services notify users by sending an email.
- To send an email just connect to an SMTP server. SMTP services are offered by every cloud provider and other 3rd parties.
- SMS messages can also be programmatically sent by connecting to a service like Twilio or SNS.

- Email/SMS notifications suffice for many apps, but they're limited.
- These can include links to your app, but cannot directly interact with your app.
- What we really want is some way to send a request to a client app.

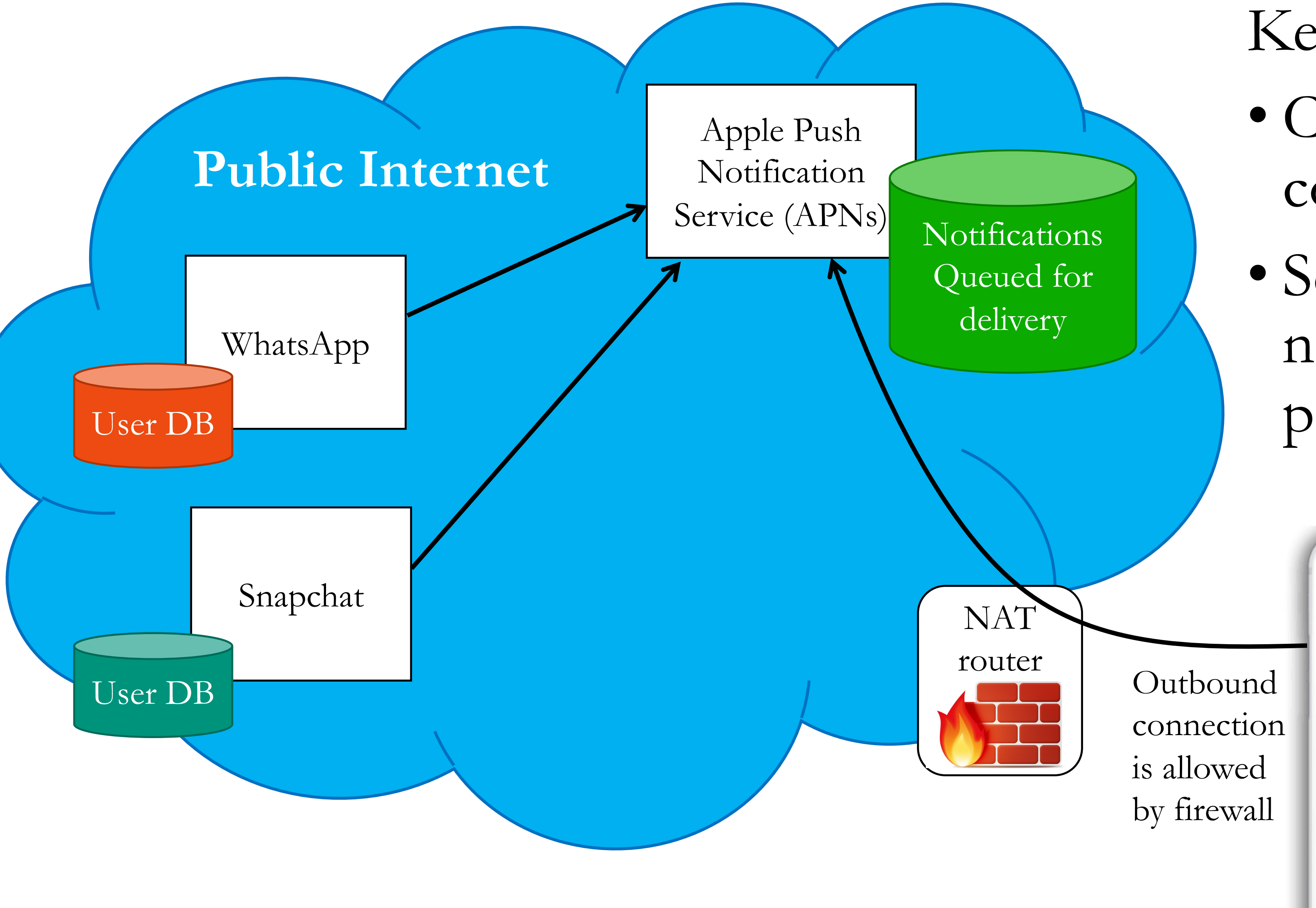
The Internet is not really a network of peers

- **Client** cannot implement a REST API beca
  it is not easily reachable. Why?

  - IP addresses change when devices move.
  - IP addresses are usually *private* (NATed).
  - Device or network may have a firewall.
  - Client does not control a DNS server.
  - May be powered off, or out of radio contact.
  - App may not always be running.
- So, most services rely on clients initiating all requests themselves, sent to always-listening services with well-known hostnames.
  - Server actions are *synchronous* with client.
  - But this is not always sufficient!
- *Push Notifications:* hacks to send msgs to clients.

Motorola DynaTAC ~1984

Solution: Push Notification Service

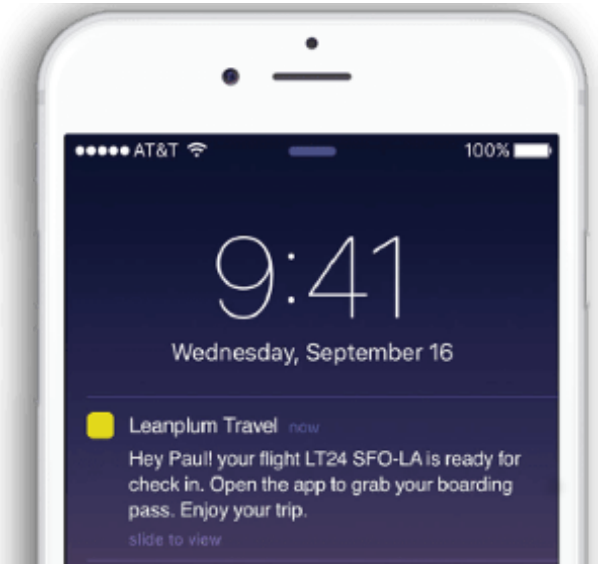


Key points:

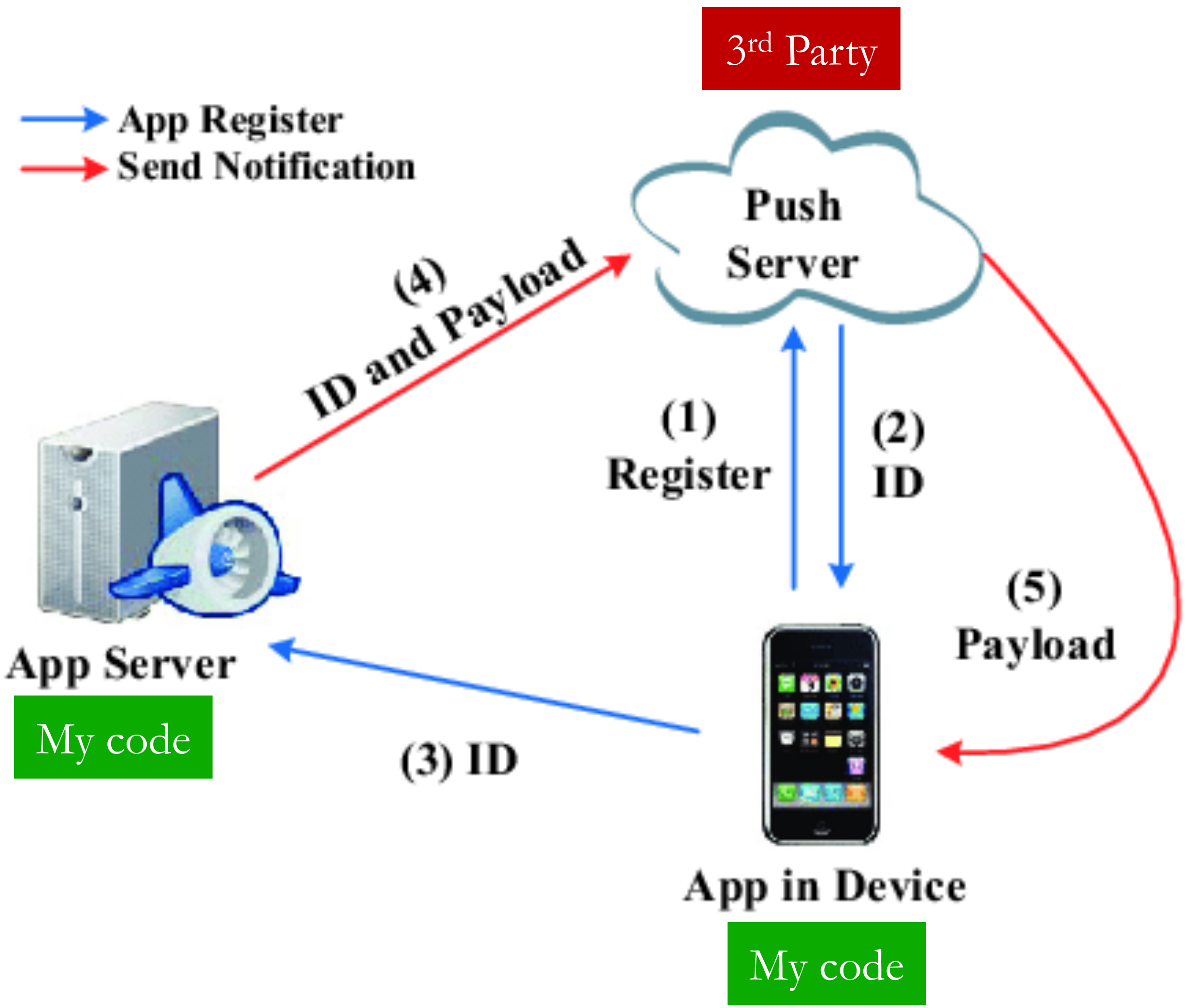
- Client's OS makes one connection for all apps.
- Services send notifications through 3rd party (Apple or Google).

Smartphone Push Notifications

- *Location registration* is used for iOS and Android push notifications.
  - Apple & Google run a **push notification service** (**PNS**) for apps on their OS.
    - *Called:* Apple Push Notifications (APN) & Google Cloud Messaging (GCM)
- Whenever phone gets a new IP address, the OS opens a long-lived connection to the PNS. PNS stores: ⟨user id, IP address⟩
- Smartphone apps like WhatsApp or Snapchat cannot contact user's phone directly; send user notifications to the PNS: ⟨user id, message⟩
  - The PNS relays the message to the user's current IP address
  - OS can show notification even if app is not running.
- On iOS, to protect users' privacy, different apps have different user ids (called device tokens).
- *If you took CS-340*: How to deal with NAT?
  - OS sends keepalive msgs. Just one port is needed for all apps.



Device Registration



Every times device connects to the network, OS creates a long-lived connection to the PNS.

1. App registers for notifications.
2. PNS returns a unique push ID.
3. App sends push ID to its backend service. Backend service stores the user's push ID in a database for later use.

Much time passes …

The backend app finally has a notification for the user!

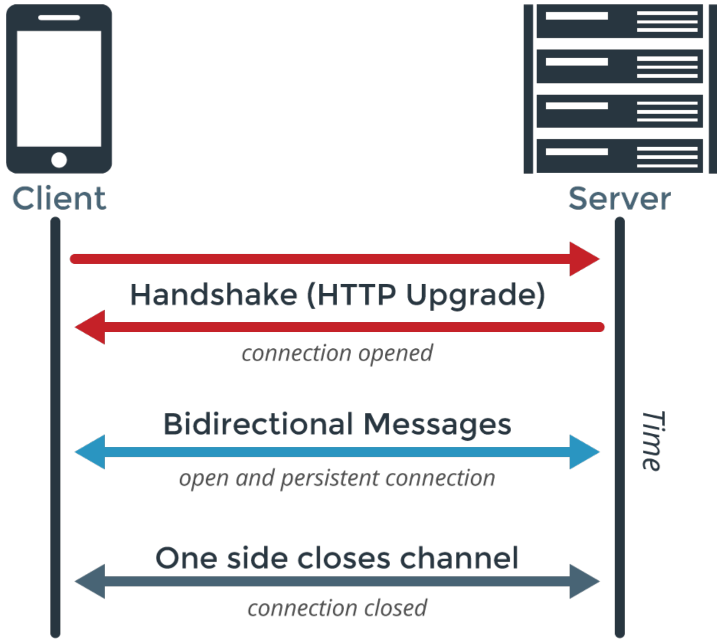
4. Backend gets the push ID for that user from its database. Sends notification request to PNS.
5. PNS uses push ID to identify the long-lived connection to the client. It relays the notification.

Web Browser Push Notifications

- Web browsers were designed to **pull** data from servers.
  - Server implements REST api, browser makes REST request to fetch data.
- Modern applications also desire **push**ed updates from the service.
  Eg., there is a new message for you, an edit occurred on a shared document, …
  - Client can make repeated requests for new data (**polling**), but this is a poor solution. Requires a tradeoff between latency and network overhead.
- **Websockets** are the preferred modern solution.
- **Long-polling** was the solution prior to websockets.
  - Both present some architectural challenges (similar to smartphone PNS).

Websockets

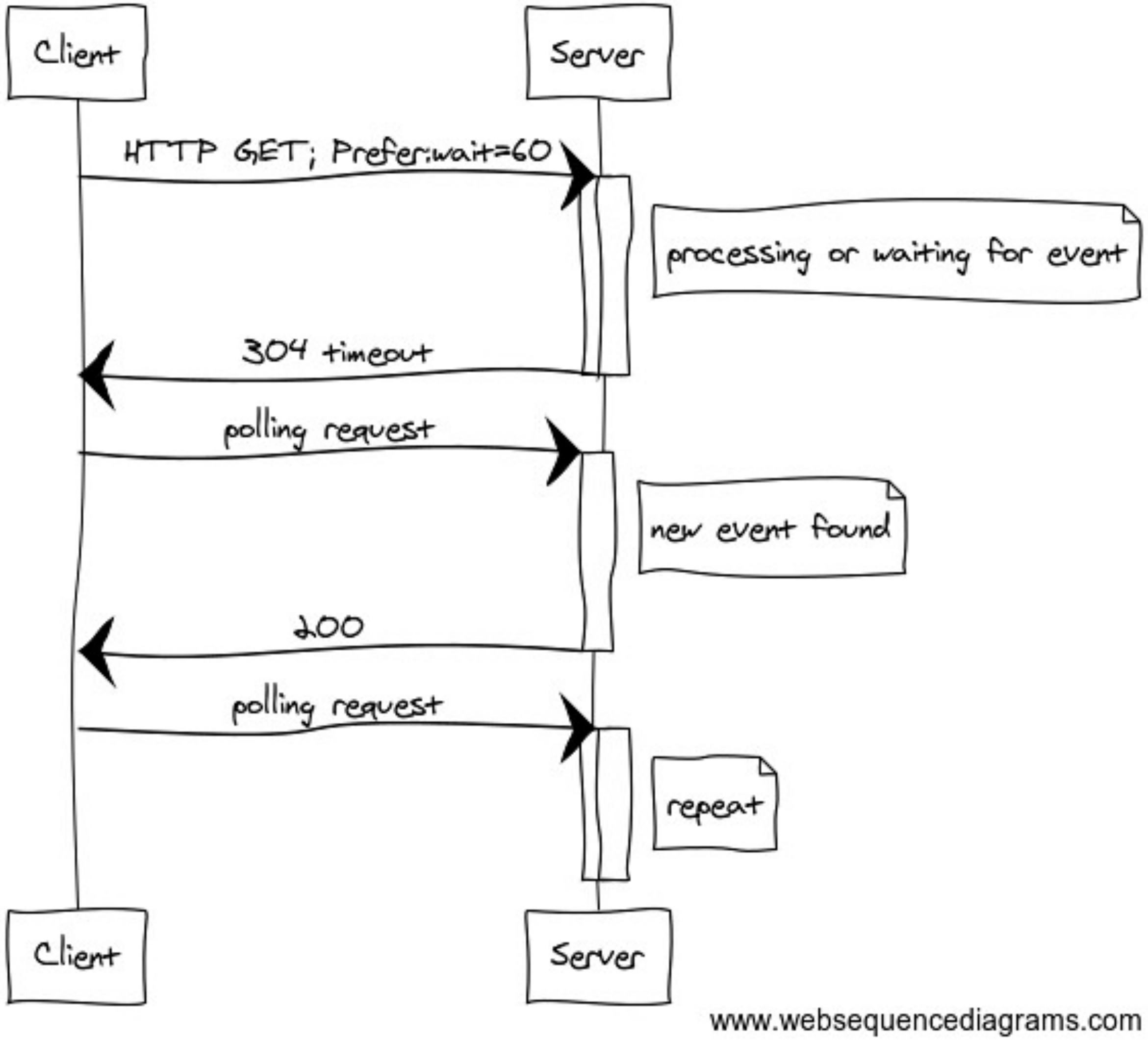
- A Websocket is a **long-lived**, **bi-directional** network connection.
  - It's similar to a TCP socket, but it's available to Javascript code in a browser.
- JS app creates a websocket connection to server.
  - Client can send API requests through the websocket.
  - Responses comes back through the websocket.
- The connection remains open!
- Server can send messages **at any time**, independent of client requests.



Old-style solution – HTTP long-polling / Comet

- Client sends an HTTP request.
- Server waits… sends a response only when new data is ready. If no data is available within 60 seconds, then send an empty response.
- Client then makes another long-polling request (infinite loop).

- **Client instantiates** the request.
- Server controls when the **response** is sent.
- Server always has one outstanding request from the client available to send data.
- *Cons:* Periodic requests every 60 seconds are wasteful. Periodic gap in service.



Comparison

HTTP	WebSocket
Duplex	
Half	Full
Messaging Pattern	
Request-reponse	Bi-directional
Service Push	
Not natively supported. Client polling or streaming download techniques used.	Core feature
Overhead	
Moderate overhead per request/connection.	Moderate overhead to establish & maintain the connection, then minimal overhead per message.
Intermediary/Edge Caching	
Core feature	Not possible
Supported Clients	
Broad support	Modern languages & clients

https://blogs.windows.com/windowsdeveloper/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/

Architectural challenges

- Whether using APN, GCM, Websockets, or HTTP long-polling, the challenge is finding the **one** long-lived connection to the client.
- A network socket (connection) is tied to **one IP address**.
- Notifications originating from anywhere in the large, distributed system must be somehow directed to the **one** appropriate notification server instance that the client is connected to.
- To solve this problem, notifications are often a separate microservice.

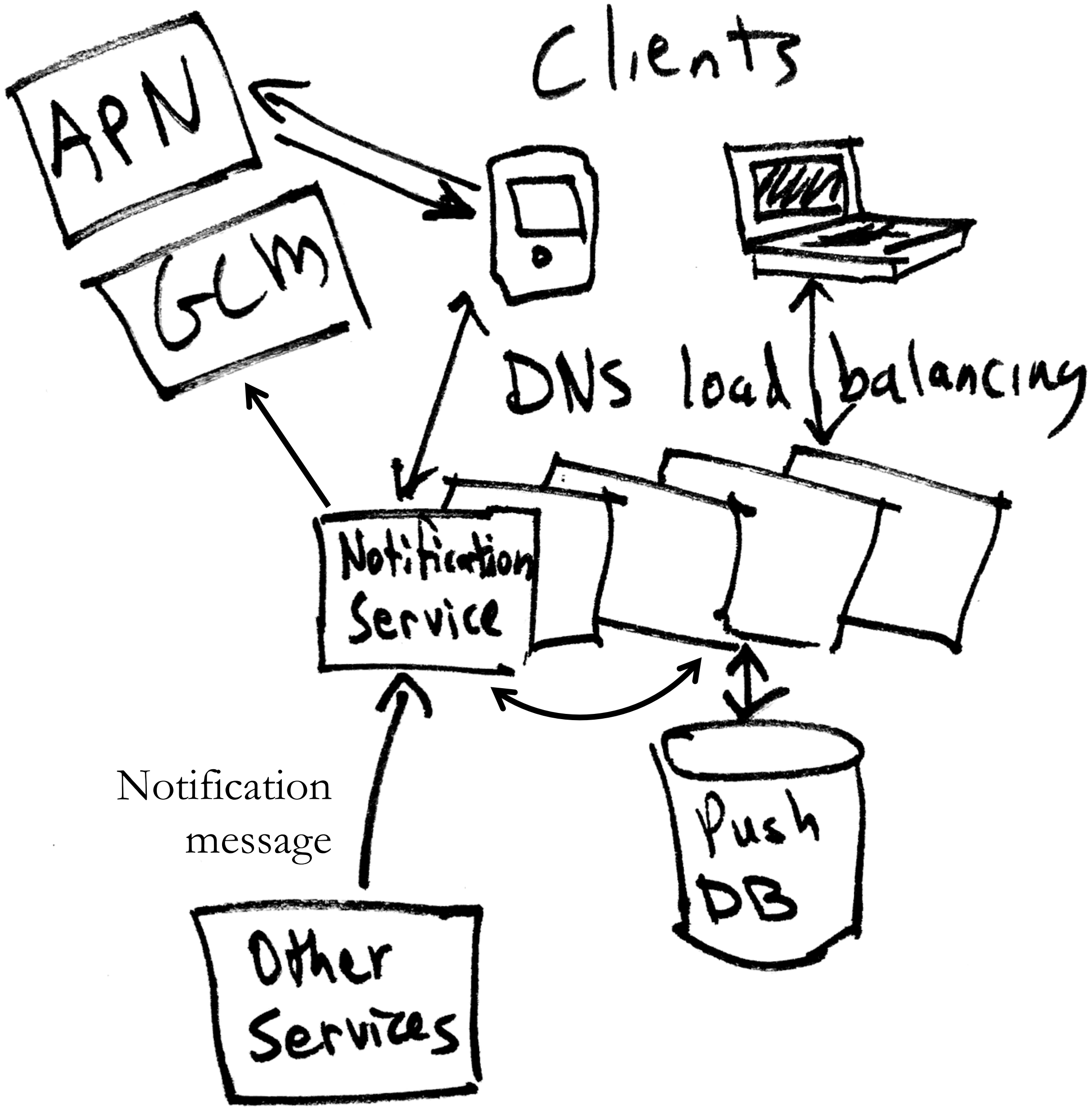
Notification Service API

Clients connect themselves in two ways:

- Opening a websocket.
- Making an API request providing a push ID usable on APN or GCM.
- In both of the cases above, the user's location is stored in a database.

Other microservices send a notification through an api call:

- POST /notification/[user_id] + *JSON notification body*
- Implementation looks up the location of the connection and relays the message.



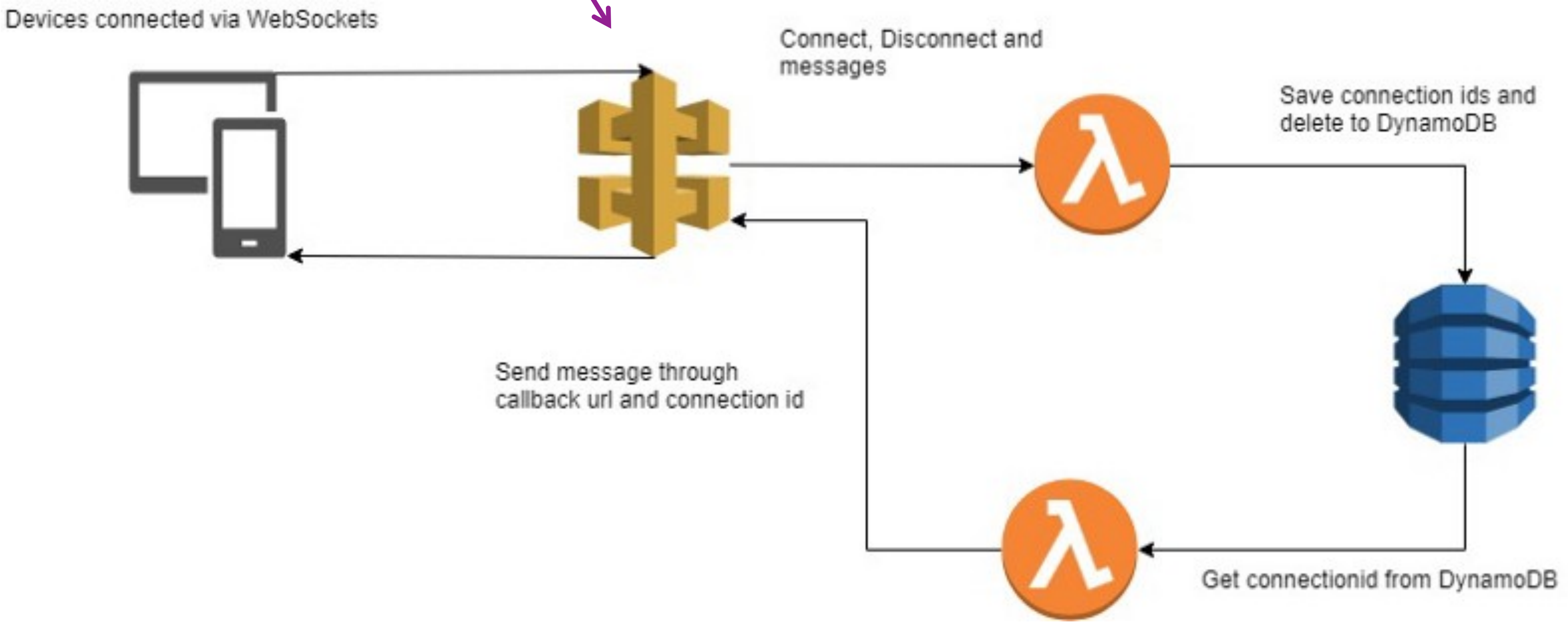
Notification Service Database Example

User	Type	Address
Alice	Android/GCM	device_id:3902390823
Steve	iOS/APNs	device_id:498092390
Steve	Websocket	5.29.193.4 : 129.29.3.2 : 29392
Steve	Websocket	5.29.193.1 : 129.29.3.2 : 9002

(notification server address : client address : client port)

- Steve will receive notifications on three different devices.
- He's running the app in two different browser tabs, and each tab is connected to a different instance of the notification service.

AWS API Gateway with websockets



Details at: https://hackernoon.com/websockets-api-gateway-9d4aca493d39

- Clients have long-lived websocket connections to gateway.
- Requests are handled by Serverless Functions (Lambdas). When connection is established, save connection id. Later use connection id to push data to clients.

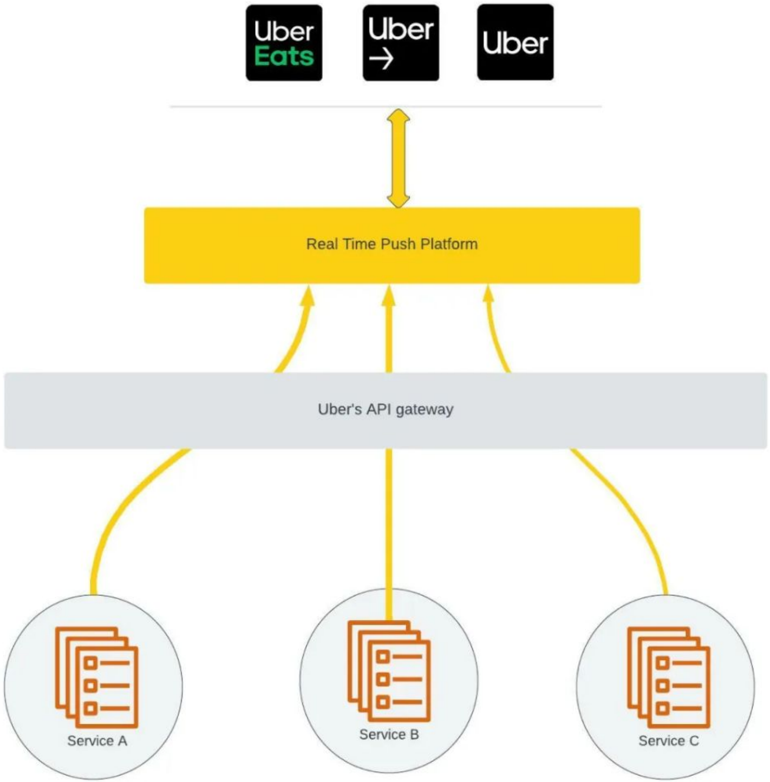
Recap

- Traditional web/app design uses a **client-server** model, but sometimes we want to **push** data to client instead of client always **pulling**.
- Asynchronously sending data to clients can be a challenge.
- Mobile OSes have special **push notification services**.
  - Allows a single connection to be shared by all apps on the phone.
  - Allows notifications to be delivered even if app is not running.
- Web browsers can use **Websockets** or **Long-polling.**
  - In both cases, client is connected to one machine and service must somehow relay messages to that connection.

Case Study: Uber's Real-Time Push Platform

Original Blog: [Uber's Real-Time Push Platform]

Overview



Uber Eats
Uber
Uber
Real Time Push Platform
Uber's API gateway
Service A
Service B
Service C

Problem

- **Context**: Uber trips involve multiple entities like riders and drivers interacting in real-time.
- **Synchronization Need**: As trips progress:
  - Riders and drivers must stay updated with backend systems.
  - Both should be in sync with each other's actions and intentions.
- **Scenario**:
  - Rider requests a ride.
  - Driver is available online.
  - Uber's backend system matches the rider with a driver and sends an offer.

Problem (cont.)

- **Communication Model**:
  - Driver app polls server: Checks for new trip offers
  - Rider app polls server: Verifies if a driver has been assigned
- **Challenge**:
  - Polling frequency varies
  - Data rate of change in the Uber app can range from seconds to hours

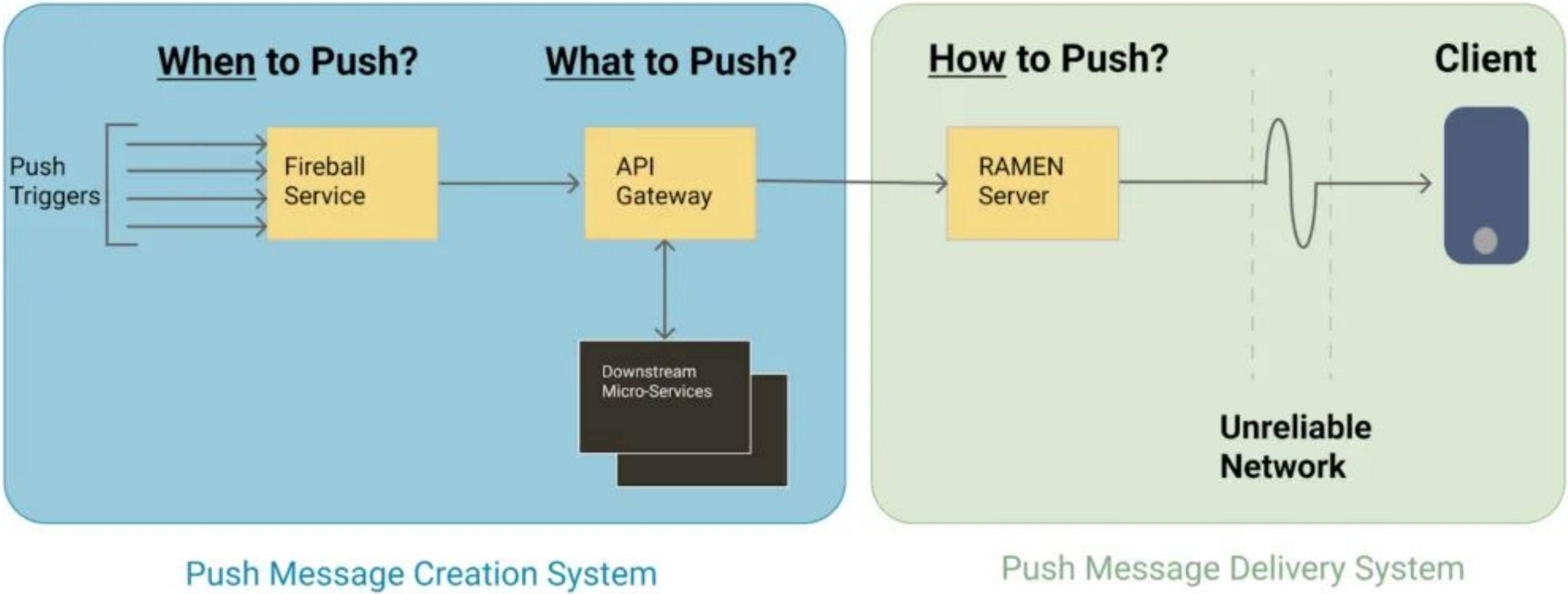
Specific Mobile App Polling Problems

- **Backend Load**:
  - 80% of backend API gateway requests were polling calls.
  - Aggressive polling increased server resource usage.
  - Bugs in polling frequency caused backend degradation.
- **Real-time Needs vs. Backend Load**:
  - As real-time data features grew, so did the backend load.
  - Approach became infeasible with the increase in dynamic data needs.
- **User Experience & Technical Challenges**:
  - Polling drained battery faster and made app sluggish.
  - Network congestion, especially in 2G/3G or unstable networks.
  - Multiple retries during each polling attempt in weak networks.

Specific Mobile App Polling Problems (cont.)

- **Developer Complications**:
  - Features growth led to overloading existing polling APIs or creating new ones.
  - Peak: app polled dozens of overloaded APIs.
  - Maintaining API-level consistency and logical separation became harder.
- **Cold Startup Issues**:
  - On app startup, all features tried to pull latest state for UI rendering.
  - Competing concurrent API calls delayed app rendering.
  - No prioritization meant increased app load time, worsened by poor networks.
- **Solution & Evolution**:
  - Developed a push messaging platform for server-to-app on-demand data.
  - Achieved efficiency improvements, but faced new challenges.
  - Upcoming: Exploration of the push platform's evolution.

Solution: RAMEN



Solution: RAMEN (cont.)

- Realtime Asynchronous MEssaging Network (RAMEN)
- **Goal**: Eliminate polling through push messaging

Solution: RAMEN (cont.)

- **Design Principles**:
  - **Easier Migration**:
    1. Leverage existing polling endpoints
    2. No need to rewrite all current polling APIs
  - **Ease of Development**:
    1. Simplify push data development to resemble polling API development
  - **Reliability**:
    1. Ensure message delivery
    2. Retry on delivery failures
  - **Wire Efficiency**:
    1. Address data usage costs, especially in developing countries
    2. Minimize data transfer between server and apps, crucial for drivers online for long durations

Overview of RAMEN

- Real-time information constantly changes across riders, drivers, restaurants, eaters, and trips
- Key challenge: Determining "when" to generate a message payload
- **Fireball**:
  - Microservice deciding "when to push a message?"
  - Uses configurations for decision-making
  - Listens to various system events to decide on push necessity
- **Example**:
  - Driver accepting an offer changes the driver and trip state
  - This change triggers Fireball
  - Fireball determines which push messages to send to involved parties
  - One trigger can result in multiple message payloads to multiple users

Generating the Message Payload

- Server calls from Uber apps served by the API gateway, including push payloads
- **API Gateway Role**:
  - Determines "what to push" after Fireball decides who and when
  - Calls various domain services to generate push payloads
- **API Types in Gateway**:
  - All APIs share a similar structure for payload generation
  - Two categories: Pull and Push APIs
  - Pull APIs: Endpoints called by mobile devices for HTTP operations
  - Push APIs: Endpoints triggered by Fireball, contain "Push" middleware to forward responses to push delivery system

Generating the Message Payload (cont.)

- **Benefits of Using an API Gateway**:
  - Shared business logic between pull and push APIs
  - Easy transition from pull API to push API for the same payload
  - Manages cross-cutting concerns: rate limiting, routing, schema validations
- **Collaboration**:
  - Fireball and gateway together generate push messages
  - Delivery to mobile devices managed by the "Push Message Delivery System"

Metadata for Push Message Payloads

- Push messages contain specific configurations for optimization
- **Priority**:
  - Need for prioritizing payload delivery due to protocol limitations and bandwidth constraints
  - Three priority categories:
    1. High: Core user experience messages
    2. Medium: Incremental UX feature messages
    3. Low: Large data payload or low-frequency, non-critical messages
  - Platform behaviors guided by priority:
    1. On connection, messages queued by descending priority
    2. High priority messages ensured reliability, server retries for RPC failures, and cross-region replication support

Metadata for Push Message Payloads (cont.)

- **Time to Live (TTL)**:
  - Aimed at enhancing real-time experiences
  - Each message given a TTL value, from seconds up to 30 minutes
  - Message delivery system persists and retries until TTL expires
- **Deduplication**:
  - Determines if repeated push messages should be deduplicated
  - For most cases, sending the latest push message sufficed, reducing overall data transfer rate

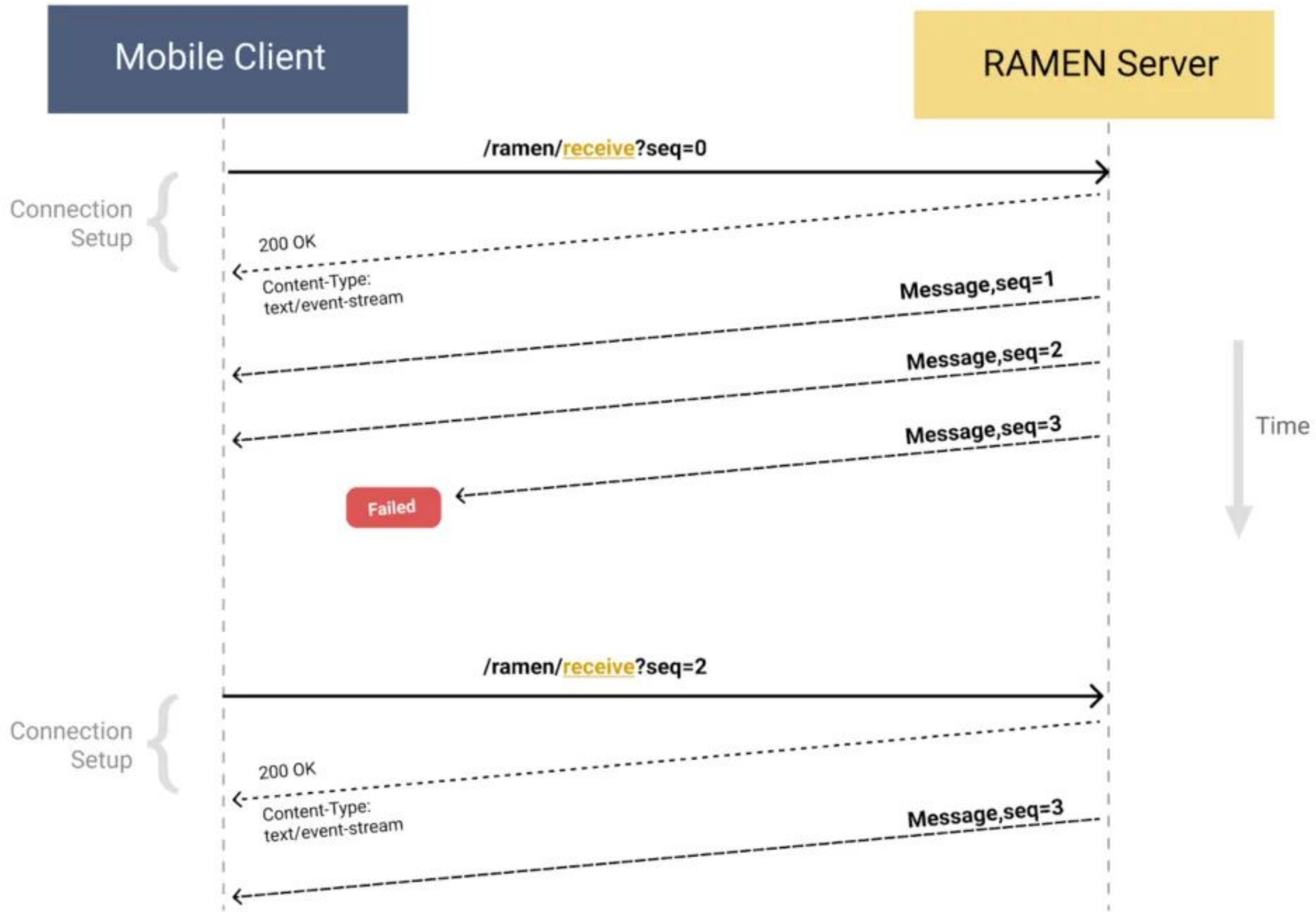
Message Delivery

- **Purpose**:
  - Deliver payload to millions of mobile apps globally, in real-time
- **Delivery Challenges**:
  - Global mobile networks have varied reliability
  - System offers an "at-least-once" delivery guarantee
- **RAMEN Protocol**:
  - Needed a reliable delivery channel; opted for a TCP-based persistent connection between apps and datacenter
  - Choices in 2015 for application protocol:
    - HTTP/1.1 with long polling
    - Web Sockets
    - Server-Sent Events (SSE)
  - (Continued on next slide)

Message Delivery (cont.)

- Choice: SSE, due to:
    - Security
    - Support in mobile SDKs
    - Minimal impact on binary size
    - Simplifies operation using existing HTTP + JSON API stack at Uber

- **SSE Limitations**:
    - Unidirectional protocol (server to app only)
    - To achieve "at-least-once" delivery, needed acknowledgments and retries
    - Custom protocol scheme defined on top of SSE for this purpose



Server-client interaction for the SSE protocol.

Message Delivery (cont.)

- **Starting Connection**:
  - Client begins by sending a request to /ramen/receive?seq=0 with sequence number (seq) 0 for a new session
  - Server responds with HTTP 200 and ‘Content-Type: text/event-stream’ to maintain SSE connection
- **Message Delivery**:
  - Server sends pending messages in descending priority order, with incremental sequence numbers
  - If a message isn't delivered (e.g., seq#3), client reconnects using the last successful seq number (e.g., seq=2), indicating non-delivery of seq#3
  - Server resends non-delivered messages based on this info
  - Simplifies resumability of streaming connection with server handling most of the record-keeping

Message Delivery (cont.)

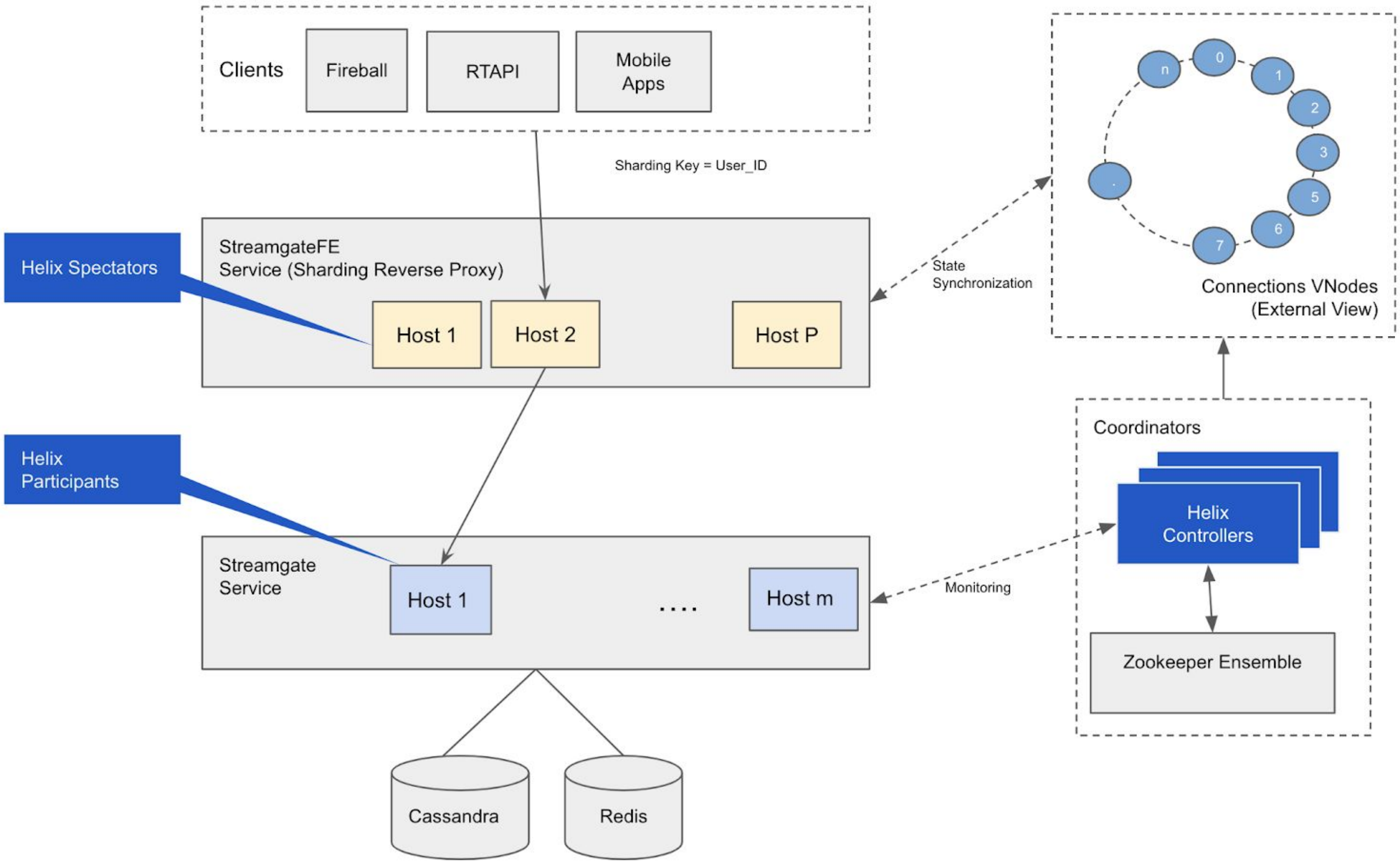
- **Heartbeat Mechanism**:
  - Server sends a byte-sized heartbeat message every 4 seconds to check connection
  - Client reconnects if no heartbeat/message received for 7 seconds
- **Acknowledgment Mechanism**:
  - Client reconnecting with higher seq acts as acknowledgment for server to delete older messages
  - On stable networks, connection may last minutes, leading to server accumulating older messages
  - To address this, client sends `/ramen/ack?seq=N` every 30 seconds irrespective of connection health

Message Delivery (cont.)

- **Protocol Flexibility**:
  - Simple design facilitates client creation in various languages and platforms
- **Device Context Storage**:
  - RAMEN Server records device context upon connection establishment
  - Context available to Fireball for user device contexts
  - Unique hash generated for each device context, isolating push messages for users on multiple devices
- **Message Storage**:
  - RAMEN stores messages in memory and a database
  - Allows for message retries until TTL expiry in unstable connections

Scaling Up RAMEN

- Rapid adoption: At peak, 70,000 QPS push messages/sec to three app types; maintained up to 600,000 concurrent streaming connections
- Integral to server-client API infrastructure
- Scaling challenges:
  - Ringpop's limited scalability with increasing nodes
  - Gossip protocol's increased convergence time with ring size
  - Single-threaded Node.js workers resulting in event loop lag
  - Potential issues: inconsistent topology information, message loss, timeouts, errors



Architecture for the new RAMEN backend server.

Scaling Up RAMEN (cont.)

- 2017: RAMEN server implementation reboot
  - **Netty**: High-performance library for network servers/clients; efficient zero-copy buffers
  - **Apache ZooKeeper**: Used for centralized sharing; ensures distributed synchronization and quick node failure detection
  - **Apache Helix**: Cluster management framework built on ZooKeeper; abstracts topology logic; monitors connected workers
  - **Redis & Apache Cassandra**: Redis as a capacity cache; Cassandra for cross-region replicated storage; combats thundering herd problems
  - (Continued on next page)

Scaling Up RAMEN (cont.)

- 2017: RAMEN server implementation reboot (cont.)
  - **Streamgate**: Implements RAMEN Protocol on Netty; handles connections, messages, and storage; interfaces with ZooKeeper
  - **StreamgateFE**: Acts as Apache Helix Spectator; reverse proxy for client requests, routing to the right Streamgate worker
  - **Helix Controllers**: Five-node service; manages topology and reallocates sharding partitions upon Streamgate node changes

Scaling Up RAMEN (cont.)

- Achievements:
  - 99.99% server-side reliability
  - Supports over ten app types on iOS, Android, Web
  - Over 1.5M concurrent connections
  - Pushes more than 250,000 messages/sec

Future Improvements to Push Infrastructure

- Goal: Improve the long tail reliability of push message delivery to devices across diverse network conditions
- Issues with RAMEN protocol:
  - **Loss of acknowledgements**: Delayed acknowledgements sometimes led to failed recognitions, making it challenging to identify genuine message losses versus failed acknowledgements
  - **Connection instability**: Varied client implementations across platforms led to inconsistencies in error handling and performance
  - **Transport constraints**: The SSE-based protocol was unidirectional, lacked real-time round trip time measurement, and struggled with binary payload transfers

Future Improvements to Push Infrastructure

- Goal: Improve the long tail reliability of push message delivery to devices across diverse network conditions
- Issues with RAMEN protocol:
  - **Loss of acknowledgements**: Delayed acknowledgements sometimes led to failed recognitions, making it challenging to identify genuine message losses versus failed acknowledgements
  - **Connection instability**: Varied client implementations across platforms led to inconsistencies in error handling and performance
  - **Transport constraints**: The SSE-based protocol was unidirectional, lacked real-time round trip time measurement, and struggled with binary payload transfers

Future Improvements to Push Infrastructure (cont.)

- **gRPC-based RAMEN Protocol**:
  - Initiated in 2019 to rectify the above issues
  - Built on the popular RPC stack, gRPC, with standardized client-server implementations across many languages
  - gRPC provides interoperability with the QUIC transport layer

Future Improvements to Push Infrastructure (cont.)

- Key Improvements with gRPC:
    - Instant acknowledgements through reverse stream, enabling better reliability
    - Real-time acknowledgements for real-time network condition assessments
    - Abstraction layers for stream multiplexing, application-level network prioritization, and flow control algorithms
    - Flexibility in message payload serialization
    - Efficient support for various apps and devices due to robust client implementations

Summary - Key Success Factors

- **Separation of Concerns**:
    - Distinct roles for message triggering, creation, and delivery
    - Clean division between Apache Helix, topology, and core streaming logic
    - Enabled gRPC support with the same architecture but varied wire protocols
- **Industry Standard Technologies**:
    - Enhances robustness and cost-effectiveness
    - Minimal maintenance overhead
    - Notably stable: Helix and Zookeeper
- **Simpler Design**:
    - Scaled across diverse network conditions and numerous apps
    - Protocol's simplicity ensured swift scalability