Next Generation Client APIs with Envoy Mobile

QCon London - March 2020

Jose Nino
@junr03

Some slides built in collaboration with: @goaway & @rebello95
Server
Networking
Networking
Client
This is fine.
Jose Nino
@junr03
Infrastructure Engineer

≈4 years @ Lyft
This is fine.
We recognized two important dimensions in API management: what data we send (shape), and how we send it (transport).
Service A
“server”

Service B
“client”

---

Server-side Interface Definition Language (IDL)

```proto
#service_a.proto
service ServiceA {
  rpc UpdateA(ARequest) returns (AResponse) {
    option (http.http_options).path = "/service_a";
    option (http.http_options).method = "POST";
  }
}
```
What is **envoy**?

- Network Proxy driven by Open-source (OSS) community
- Common substrate for network behavior
Strongly-typed APIs and a Universal Network Primitive
The End

(circa early 2019)
Strongly-typed APIs and a Universal Network Primitive
What do we want from our Client APIs?

Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.
What do we want from our Client APIs?

*Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.*

<table>
<thead>
<tr>
<th>Feature</th>
<th>Consistency</th>
<th>Performance</th>
<th>Extensibility</th>
<th>Resilience</th>
<th>Security</th>
<th>Observability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Clients as another node in the mesh

Mobile Client → Edge

Edge → Service A

Service A → Sidecar

Sidecar → Service B

Service B → Sidecar

Sidecar → 3rd-party

3rd-party → Storage

Storage → twilio

API
Next generation Client APIs with Envoy Mobile

We recognized two important dimensions in API management: **what** data we send (shape), and **how** we send it (transport).

~1 year in Client Networking
We recognized two important dimensions in API management: *what data we send* (shape), and *how* we send it (transport).
Early API workflow at Lyft

Tech spec

- handwritten.swift
- handwritten.kt
- handwritten.py/.go
Programming errors

iOS

```json
{
  "key_a": "hello"
}
```

Android

```json
{
  "kye_a": "hello"
}
```

Server
Programming errors

{ "key_a": "hello"
}

{ "kye_a": "hello"
}

Server

iOS

Android

@rebello95
Programming errors

iOS

```
{  "key_a": "hello"
}
```

Server

Android

```
{  "kye_a": "hello"
}
```
Programming errors

iOS

```json
{
  "key_a": "hello"
}
```

Android

```json
{
  "kye_a": "hello"
}
```

Server

This is fine.
Programming errors

if (android):
    else:

```json
{  "key_a": "hello"
}
```

```json
{  "kye_a": "hello"
}
```
Problems

- Handwritten code led to mistakes/typos
- No visibility into deserialization failures at compile time
YAML schema integration
YAML definitions

paths:
  /foo:
    post:
      parameters:
        - in: body
          schema:
            $ref: '#/definitions/FooRequest'
      responses:
        200:
          schema:
            $ref: '#/definitions/FooResponse'
YAML results

- Manual code generation for Android
- No integration on iOS
- No single source of truth
Consistency

We needed a source of truth that provided guarantees about the shape of any given API.
Envisioned workflow

Tech spec → IDL repo

foo_api.proto

foo_api.swift
foo_api.kt
foo_api.py/go
Why protobuf?

- Platform-neutral IDL for defining and serializing strongly-typed APIs.
- JSON support (migration)
- Already used server-side (consistency)
IDL pipeline

/api/v1/foo.proto

Generators

- foo.swift
- foo.kt
- foo.py
- foo.go
- +7 others
IDL pipeline

/api/v1/foo.proto → Generators →
- foo.swift → FooV1API library
- foo.kt → foo-v1-api library
- foo.py
- foo.go
- +7 others
Using generated APIs in Swift
Generated models

// Modules/FooV1API/FooAPI.swift

public struct FooRequest: Codable, Equatable {
    ...
}

public struct FooResponse: Codable, Equatable {
    public let id: Int64
    public let firstName: String
}
import RxSwift

public protocol FooAPI {
    func updateFoo(_ request: FooRequest) -> Single<Result<FooResponse, FooAPIError>>
}

public struct FooClientAPI: FooAPI {
    // Abstracted implementation
}
import RxSwift

public protocol FooAPI {
    func updateFoo(_ request: FooRequest) -> Single<Result<FooResponse, FooAPIError>>
}

public struct FooClientAPI: FooAPI {
    // Abstracted implementation (hint hint wink wink!)
}
Generated mocks

// Modules/FooV1APIMock/FooAPI.swift

open class FooMockAPI: FooAPI {
    open lazy var mockUpdateFoo: (FooRequest) -> Result<FooResponse, FooAPIError> = {
        ...
    }

    open func updateFoo(_ request: FooRequest) -> Single<Result<FooResponse, FooAPIError>> {
        ...
    }
}
Results

- Single source of truth for all APIs
- Consistent, generated code on all platforms
- Highly testable
- Underlying transport fully abstracted 🌟 565958
Performance

Experimenting with payload encoding
Existing system

Client

Service

JSON request

JSON response
Content negotiation

Client

JSON request
Content-Type: json
Accept: x-protobuf, json

Middleware
JSON > Protobuf

Protobuf request
Content-Type: x-protobuf

Middleware
No change

Service

Protobuf response
Content-Type: x-protobuf

Protobuf response
Content-Type: x-protobuf
Wins from protobuf

- 40%+ reduction in payload sizes
- Faster response times
- Higher success rates
- Transparent to engineers
Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.

What do we want from our Client APIs?

Consistency ✓ ✓
Performance ✓ ✓
Extensibility ? ✓
Resilience ? ✓
Security ? ✓
Observability ? ✓
Extensibility

Using protobuf annotations to declare API behavior
service Foo {
  rpc GetFoo(FooRequest) returns (FooResponse) {
    option (http.http_options).path = "/foo/receive";
    option (http.http_options).method = "GET";
    option (http.http_options).client_poll_ms = 5000;
    option (http.http_options).gzip_enabled = true;
  }
}
What do we want from our Client APIs?

Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.

<table>
<thead>
<tr>
<th>Feature</th>
<th>✔️</th>
<th>☹️</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Performance</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Extensibility</td>
<td>☹️</td>
<td>✔️</td>
</tr>
<tr>
<td>Resilience</td>
<td>☹️</td>
<td>✔️</td>
</tr>
<tr>
<td>Security</td>
<td>☹️</td>
<td>✔️</td>
</tr>
<tr>
<td>Observability</td>
<td>☹️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
We recognized two important dimensions in API management: *what* data we send (shape), *and how* we send it (transport).
Clients as another node in the mesh

- Client
- Edge
- Service A
  - Sidecar
- Service B
  - Sidecar
- 3rd-party
- Storage

@goaway @junr03
Standardizing infrastructure
Why is world domination transport standardization useful?

- Write once, deploy everywhere
- Common tooling for common problems
- Reduce cognitive load
Envoy Mobile to the rescue!

- v0.1: proof-of-concept
- v0.2: laying the foundation
- v0.3: first production-ready release
Envoy as a Library

How did we turn a network proxy into a networking library?
Build System: bazel

- Cross-platform support
- Used by Envoy
Layered Architecture

Platform (iOS/Android)

//library/swift:ios_framework
//library/kotlin:android_framework
//:android_dist

Bridge (C)

//library/common:main_interface
//library/common:c_types

Native (C++/Envoy)

//library/common/...
@envoy//source/...
How to run a process in an app?

picture of an engine (a very fast one)
Threading contexts

Application Threads

Envoy Main Thread

Callback Threads
## Library Matrix

<table>
<thead>
<tr>
<th>Platform (iOS/Android)</th>
<th>Bridge (C types/bindings)</th>
<th>Native (C++/Envoy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Envoy Main Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Callback Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform (iOS/Android)</td>
<td>Bridge (C types/bindings)</td>
<td>Native (C++/Envoy)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Application Threads</td>
<td>EnvoyEngine</td>
<td></td>
</tr>
<tr>
<td>Envoys Main Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Callback Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform (iOS/Android)</td>
<td>Bridge (C types/bindings)</td>
<td>Native (C++/Envoy)</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Application Threads</td>
<td>EnvoyEngine</td>
<td></td>
</tr>
<tr>
<td>Envoys Main Thread</td>
<td>EnvoyRunner</td>
<td></td>
</tr>
<tr>
<td>Callback Threads</td>
<td>Dispatcher</td>
<td>Http Manager</td>
</tr>
</tbody>
</table>

- **Library Lifecycle - using Envoy Constructs**

- **Platform (iOS/Android)**
  - Application Threads
    - EnvoyEngine

- **Bridge (C types/bindings)**
  - EnvoyRunner
  - Dispatcher
  - Http Manager
Library Lifecycle - starting a stream

Platform (iOS/Android)  Bridge (C types/bindings)  Native (C++/Envoy)

Application Threads
- EnvoyEngine
- HttpStream

Envoy Main Thread
- EnvoyRunner

Callback Threads
- Dispatcher
- Http Manager

EnvoyEngine → HttpStream → EnvoyRunner → Dispatcher → Http Manager
Library Lifecycle - dispatching a stream

Platform (iOS/Android) | Bridge (C types/bindings) | Native (C++/Envoy)

Application Threads

- EnvoyEngine
- HttpStream

Envoy Main Thread

- EnvoyRunner
- Dispatcher
- Http Manager

Callback Threads
Http Filters

- Foundational extension point
- Enact behavior
- Provide the guarantees we want!
let envoy = try EnvoyClientBuilder(domain: "api.envoyproxy.io")
  .addLogLevel(.warn)
  .addStatsFlushSeconds(60)
  .build()

val envoy = EnvoyClientBuilder(
  Domain("api.envoyproxy.io"))
  .addLogLevel(LogLevel.WARN)
  .addStatsFlushSeconds(60)
  ...
  .build()
let envoy = try EnvoyClientBuilder(domain: "api.envoyproxy.io")
  .addLogLevel(.warn)
  .addStatsFlushSeconds(60)
  .build()
Build an Engine

```swift
let envoy = try EnvoyClientBuilder(domain: "api.envoyproxy.io")
    .addLogLevel(.warn)
    .addStatsFlushSeconds(60)
    .build()
```
Build a Request

```swift
let request = RequestBuilder(path: "/pb.api.v1.Foo/GetBar")
    .addHeader(name: "x-custom-header", value: "foobar")
    .build()
```
let request = RequestBuilder(path: "/pb.api.v1.Foo/GetBar")
    .addHeader(name: "x-custom-header", value: "foobart")
    .build()
Build a Response Handler

```swift
let handler = ResponseHandler()
    .onHeaders { headers, status, _ ->
        ...
    }
    .onData { data ->
        // Deserialize message data here
    }
    ...
```
let handler = ResponseHandler()
    .onHeaders { headers, status, _ ->
        // Product logic
    }
    .onData { data ->
        // Product logic
    }
...
What do we want from our Client APIs?

Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.

Consistency ✓ ✓
Performance ✓ ✓
Extensibility 😕 ✓
Resilience ? ✓
Security ? ✓
Observability ? ✓
IDL pipeline

/api/v1/foo.proto

Generators

foo.swift
foo.kt
foo.py
foo.go
+7 others

FooV1API library

foo-v1-api library

Client

@goaway @junr03
Ecosystem

Shape + Transport = Ecosystem!
Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.

<table>
<thead>
<tr>
<th>What do we want from our Client APIs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
</tr>
<tr>
<td>Performance</td>
</tr>
<tr>
<td>Extensibility</td>
</tr>
<tr>
<td>Resilience</td>
</tr>
<tr>
<td>Security</td>
</tr>
<tr>
<td>Observability</td>
</tr>
</tbody>
</table>

- Consistency: ✓ ✓
- Performance: ✓ ✓
- Extensibility: ? ✓
- Resilience: ? ✓
- Security: ? ✓
- Observability: ? ✓
Extensibility

Using protobuf annotations to declare API behavior
Declarative APIs

```protobuf
def service Foo {
    rpc GetFoo(FooRequest) returns (FooResponse) {
        option (http.http_options).path = "/foo/receive";
        option (http.http_options).method = "GET";
        option (http.http_options).client_poll_ms = 5000;
        option (http.http_options).gzip_enabled = true;
    }
}
```

@rebello95
Extensibility

Using protobuf annotations to declare API behavior and Envoy Mobile to enact consistent behavior.
Http Filters

- Foundational extension point
- Enact behavior
- Provide the guarantees we want!
Envoy filters

Client

Mobile

Compression Filter
gzip request

Edge

Compression Filter
Unzip payload

Proxy

@junr03
Advanced transport

- Dynamic Forward Proxy ✅
- Compression 🚧
- Deferred Requests 🔒 (Resilience)
- OAuth 🔒 (Security)
- Request Signing 🔒 (Security)
- Network condition 🔒 (Resilience)
Deferred Requests

Client → Mobile → Deferred Request Filter → Proxy → Service

Client → Mobile → Deferred Request Filter → Proxy → Service
Request Signing

Client
- Mobile
- Request Signature Filter
  - sign request

Edge
- Request Signature Filter
  - Verify signature
- Proxy

@junn03
Advanced transport

- Dynamic Forward Proxy ✅
- Compression 🚧
- Deferred Requests 🔔 (Resilience)
- OAuth 🔔 (Security)
- Request Signing 🔔 (Security)
- Network condition 🔔 (Resilience)
Observability

True end-to-end insight
Observability

\texttt{ts(envoy\_mobile.cluster.api.upstream\_rq.count)}

\texttt{ts(envoy\_edge.cluster.*.upstream\_rq.count)}
Time-series Metrics

Metrics Service

@goaway @junr03
Dashboards!

ts(envoy_mobile...)

ts(envoy_edge...)

@goaway @junnr03
Three 9s of reliability at the server-side is meaningless if the user of a mobile client is only able to complete the desired product flows a fraction of the time.

What do we want from our Client APIs?

- Consistency ✓ ✓
- Extensibility ✓ ✓ ✓
- Performance ✓ ✓ ✓
- Resilience ✓ ✓ ✓
- Security ✓ ✓ ✓
- Observability ✓ ✓ ✓
Onwards!

- Additional functionality in filters
- Protocol Experimentation with QUIC
- **OSS Code Generators**: complete OSS ecosystem for model-based APIs.
Next Generation Client APIs

Model-based APIs defined in a strongly-typed IDL so platform engineers can iterate on behavior using a common transport layer.
Alpha-Beta Production Experiment @Lyft!
Join us!