How We Built Filmgrain, Part 2 of 2

By Ryan on July 02, 2013

Filmgrain ranks movies by popularity on Twitter and enables users to watch what people are tweeting about those movies in real time. Part 1 [1] describes how we built our backend making extensive use of Redis [2]. This post focuses on why we chose to build a TCP API, how we structured our API, and the technologies we use on the endpoints so that they can handle a large number of concurrent TCP clients.

Why Build A TCP API?

While an HTTP API is ideal for many types of apps, the fact that there is so little discussion of alternatives is troubling. I fear that we as a community of developers have gotten dogmatic in our thinking that an HTTP API is always the best choice. In fact, I believe it is definitely the *wrong* choice if you're building an app that would benefit from doing things other than simply downloading and presenting information to the user.

There is a huge opportunity for a next generation of apps that feel more alive to users by moving way from the poll-then-present model as embodied by the refresh button or gesture that exists in many apps. This is what made push notifications so exciting—instead of checking my email by opening the app and hitting refresh, I'm made aware of new emails as they arrive.

The very concept of needing to refresh is the result of thinking-in-HTTP-APIs and, by moving away from the need to refresh, we can build noticeably better experiences for our users. To that end, we chose to build a TCP API for Filmgrain to see how it worked out.

Structuring Our TCP API

Choosing to build a TCP API means that many of the decisions that are made for

you when building an HTTP API now fall on you as the developer. This includes how requests are made, how they are encoded, and how each request itself is structured.

The first choice we made was that each mobile client would open and maintain a single TCP connection to one of our endpoints and that all communication between the mobile client and the backend would take place over this connection. Since TCP provides bidirectional communication and guarantees in-order delivery over the same connection, this choice provided us with a solid framework for communicating. TCP says nothing, however, about how messages are encoded and structured so we had to decide that next.

We chose newline terminated JSON as the encoding for messages between the client and server. This ended up being an easy decision since JSON is familiar and flexible, and newlines make sense as indicators for the end of a message.

The Communication Protocol

We structure each message as a JSON object with two properties: an action and its parameters. All messages between the clients and the backend follow this same structure. The JSON looks like this:

```
{"action": "action_name", "params": { ... }}
```

An example session of Filmgrain will look something like this:

Upon opening the app...

```
{"action": "connect", "params": { ... }}
{"action": "set_movies", "params": {"movies": { ... }}}
```

When the user picks a movie...

```
{"action": "start_feed", "params": {"movie_id": 1234}}
{"action": "post_tweet", "params": {"tweet": { ... }}}

{"action": "post_tweet", "params": {"tweet": { ... }}}
```

```
{"action": "post_tweet", "params": {"tweet": { ... }}}
```

When the user leaves that movie...

```
{"action": "stop_feed", "params": {"tweet": { ... }}}
```

Load Balancing

Adopting a load balancing strategy is important since each client connects directly to an endpoint and these endpoints can only handle so many concurrent connections. Spreading these TCP connections among multiple endpoints can be accomplished in many ways, but we think we've chosen a simple and scalable way.

When the app is opened, before it can connect to an endpoint it first downloads a JSON file from S3 containing a list of available endpoint IP addresses and ports. The mobile client then chooses one of these endpoints at random and connects.

By having each mobile client pick an endpoint at random, this balances the load among the endpoints reasonably well. In the event of one endpoint getting overburdened, we gave each endpoint the ability to tell any of its clients to connect to a different endpoint.

The JSON file listing available endpoints is updated automatically as we bring up and down endpoints, and, since we've hosted it on S3, it has extremely high availability. We believe this strategy is an simple, durable, and inexpensive way of balancing the load among endpoints.

Security

Choosing to build a TCP API doesn't mean you will have to give up the option of encrypting all of your communication. In fact, we chose to make our API secure from the start. Each mobile client opens a secure socket connection to one of our endpoints and the encryption is provided by all of the same tools that are used to handle HTTPS, including OpenSSL and an SSL certificate.

Building The Endpoints

We wrote our endpoint process in Python and use gevent [3] so that each endpoint can support a large number of concurrent clients. We chose gevent over an evented framework like Twisted [4] because it is simpler and cleaner to write standard python for gevent than it is to write callback based python for Twisted.

The coroutine model of gevent is great to work with and we've been very happy with the performance and stability. By saving us from having to use threads or callback-ridden code, gevent has made our lives much easier.

That's All

Building the Filmgrain app on top of our TCP API was an eye-opening experience. Once we had the mobile client and the server chatting, adding all of the functionality became trivial from a networking perspective. HTTP is just a burden and I encourage more app developers to consider building a straight TCP API for their next app.

Discuss this post on reddit [5]

Discuss this post on Hacker News [6]

1. http://blog.filmgrainapp.com/2013/06/25/how-we-built-filmgrain-part-1-of-2/
2. http://redis.io/
3. http://www.gevent.org/
4. http://twistedmatrix.com/trac/
5. http://www.reddit.com/r/programming/comments/1hi769/building_a_tcp_api_how_we_built_our_realtime_app/
6. https://news.ycombinator.com/item?id=5980270