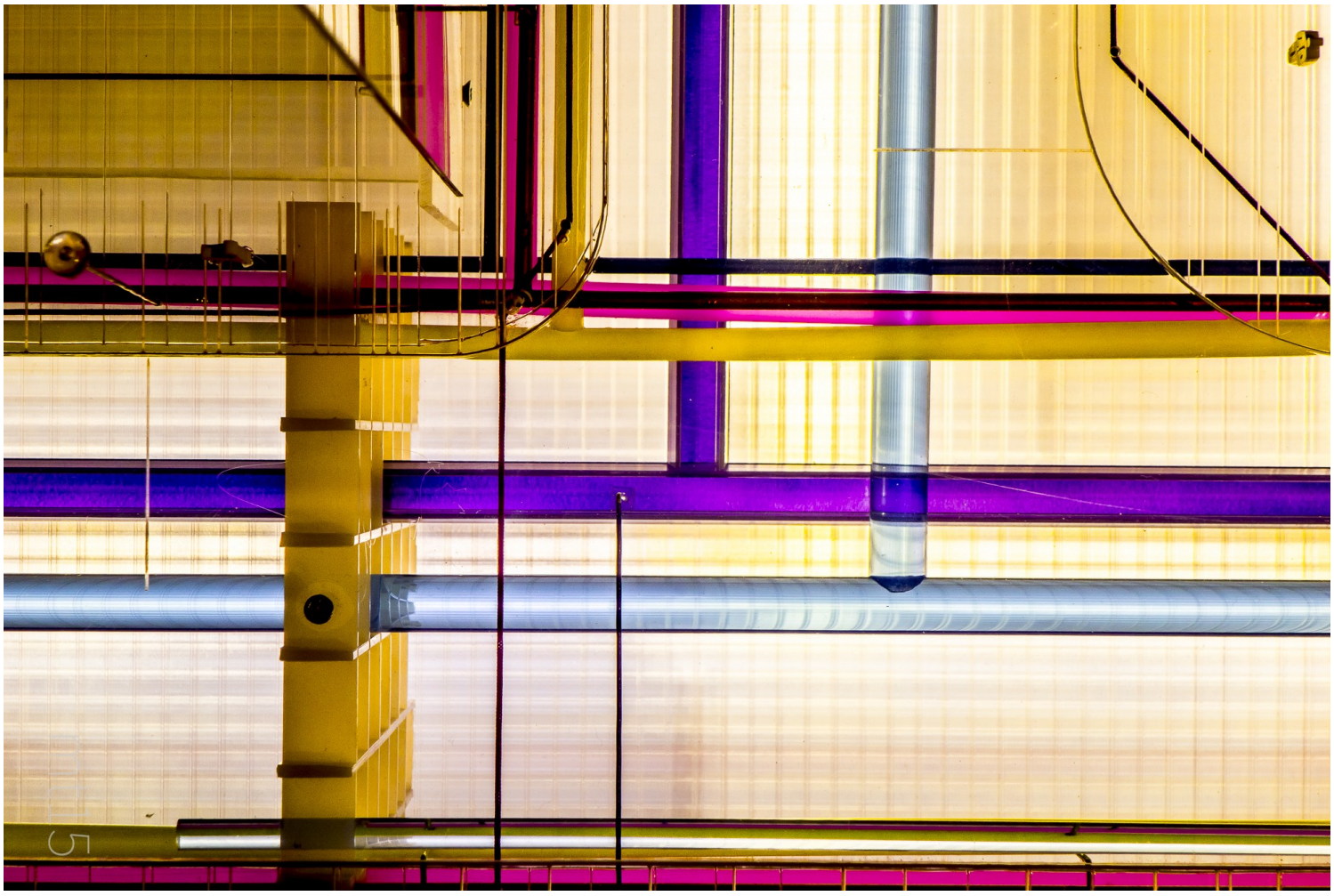
Slack Engineering Follow
Jun 2, 2016 · 8 min read

Making Slack Feel Like Slack

Our plans for deduplicating client code and improving edge caching, by Haim Grosman, Mike Fleming, and Keith Adams



"Tubes" by Manel Torralba (cc-by)

Systems problems are rooted in impossible dreams. Your file system wants to give you infinite, fast, durable storage. Your garbage collector and your kernel's virtual memory subsystem both strive, in very

different ways, to provide the illusion of infinite, fast, volatile memory. The constraints of physical reality make these hopes impossible to realize in *every* case, but astonishingly many of the common cases can be handled well.

Your Slack client strives to be a consistent, compact, zero latency, searchable replica of all of the files, messages, custom emojis, voice calls, bots, sound effects, etc. that your team is sharing in real time. Since Slack clients run on physical devices, this is impossible, so we must make do. Slack engineers working on the client-side have rummaged around in the systems toolbox to handle problems like:

1. **Latency to the Slack mothership.** Many people have a slow round-trip time to our data centers. How can we hide this latency? Avoid round-trips? Piggy-back predictable sequels on existing round-trips? Use our more limited edge presence wisely?
2. **Connectivity**. Mobile inherently means unpredictable network conditions. Connections come and go, and vary in their quality and network affinities, even in the middle of sessions. How can we maintain as much utility as possible in degraded conditions, and exploit good conditions when available?
3. **Cache coherency**. Slack clients statefully cache some information. This information needs to be be evicted or updated in the client when it changes.
4. **Managing cache capacity**. Teams contain too much data to leave all of it on members' phones. We need to be discriminating about how we use the finite resource of storage.

Writing a high-quality Slack client is *tricky*. We've done it a few times over now, and are boiling the lessons down into a modest-sized native library called **libslack**. With the caveat that libslack is still a work-in-progress, we are expecting to reap the following benefits:

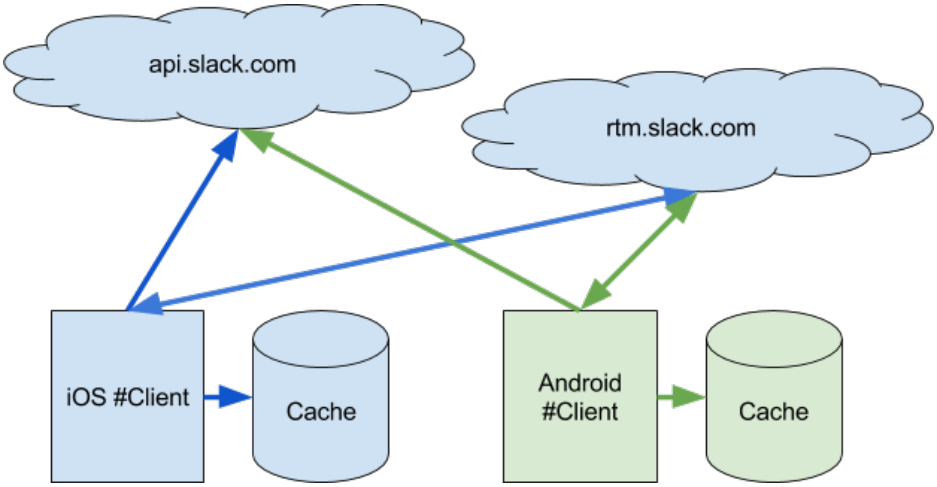
1. **Concentration of software effort**. It's easier to debug and tune one piece of code than *N*.
2. **Better client code**. By providing native language-level types for objects like users, messages, channels, and teams, instead of least-common-denominator JSON, we can avoid some classes of bugs caused by dynamic typing.

3. **Better edge caching.** The goals of a good Slack *client* have significant overlap with the goals for a Slack-aware cache in our edge points-of-presence. We're creating and deploying an edge cache built around libslack.

Mobile Tower of Babel

We at Slack find ourselves wrangling a vast and growing body of native client code. People use Slack from the web, from desktop clients for Mac OS, Windows, and Linux, and from native mobile clients for iOS, Android, and Windows Mobile. Some people also connect over XMPP or IRC using gateways that we operate on behalf of the team, and these gateways behave, for all intents and purposes, as alternative 'clients' that also happen to be servers for other protocols. Each of these clients is a mostly-separate codebase, with its own features, bugs, roadmap, code idioms, experts in the codebase, and gotchas. While having separate codebases with similar purposes is irritating to the engineering spirit, some of this is just a necessary evil in 2016. To date, platform capabilities have been shifting too fast for any of the Grand Unified Native Frameworks to reliably produce native-feeling, high-performance, feature-rich apps. So we roll up our sleeves and write different clients for different platforms.

This would be fine, of course, if all of the effort in a client went towards doing platform-specific, hardware-constrained things. Writing multi-touch UI code in an iOS-specific way, for example, is a perfectly natural thing. But a lot of the engineering effort in these clients is the sort of platform-agnostic, distributed caching work that we hope to deduplicate in libslack.



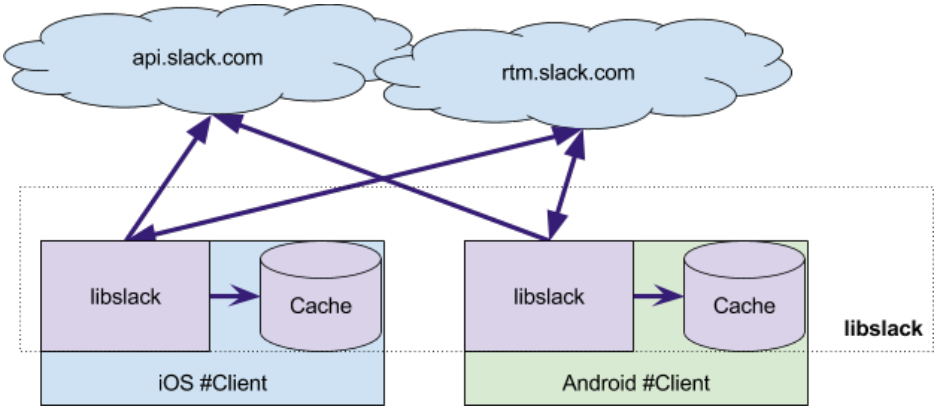
libslack

To understand libslack, we need a rough model of the interaction between a Slack client and the mothership. The two big client-facing channels are the REST API, and the Real-Time Messaging API. The former is a typical set of GET/POST endpoints for app-level state. The RTM API, on the other hand, is WebSocket-based, and is where full-duplex, latency-sensitive conversational stuff (humans sending messages to other humans) actually happens.

Libslack presents an object-oriented interface to a model of a Slack team. From the perspective of a libslack client, the non-local aspects of interacting with a team are entirely sealed off. Libslack's virtual team is an always-on, always-consistent team as seen by a logged-in user. All of the behind-the-scenes systems work of caching, syncing, and hiding latency is hidden behind this virtual team abstraction. Libslack is about 20kloc of C++ at this writing, and is functional enough to support prototyping of demo clients. We are in the process of migrating the flagship Android and iOS Slack clients to libslack, and plan to use it universally going forward.

With respect to language choice: C++ provides a lowest-common-denominator environment common to all the desktop and mobile environments we are targeting. It also provides the kind of low-level control of resource footprint that is appropriate for libraries that may be linked into unknown, potentially much larger applications, with their own ideas about how to use CPU and memory.

So now, where before we had several different copies of the code talking to the Slack mothership, we now have just one:



Libslack Conceptual model

Libslack expects to be called from an application language with a lowest-common-denominator of object-oriented features: roughly, classes and dynamic dispatch. We use an open source interface definition language to generate cross-language bindings for C++, Java, and Objective-C. The core object is a SlackAPI, which represents one user's view of a team. Within libslack, RTM messages are translated into *events*, and a SlackAPI manages a pool of worker threads that consume events from a central event queue. Libslack treats these events as cache coherency messages that enable it to update its cached model of the team.

The client subscribes to state updates, first creating views of a SlackAPI, and then registering callbacks to fire when the underlying model state changes. For example, an Objective-C client that displays the channels a user subscribes to might look like:

```
@implementation ChannelsViewController
{
  LSSlackApi* api_;
  LSVmChannelsList* channelList_;
  // ...
}
```

```
{ // In init code …
  channelList_ = [api_ createVmChannelList];
  [channelList_ subscribe:self];
```

```
  // ...
}
```

```
- (void)onSectionChannelUpdated:(LSSectionType)section
              prevIndex:(int64_t)prevIndex
               newIndex:(int64_t)newIndex
                channel:(nonnull LSVmChannelListItem
*)channel
{ // Code to update view of channels here
}
@end
```

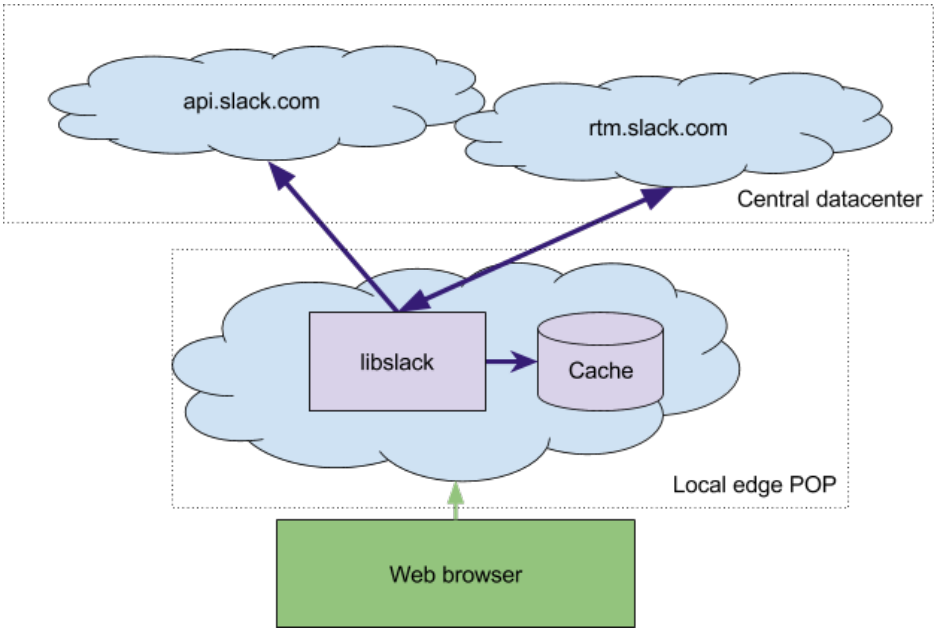
Compared with directly consuming the REST API and rtm stream, client code benefits from:

1. **Strong typing.** Typical native clients unpack the JSON returned by REST endpoints into native structures in an error-prone and repetitive way. This work is hidden in libslack.
2. **Async API.** The work of noticing and responding to update events has been pulled off of the main thread, where synchronous communication tends to cause lag and jitter for UI updates.

libslack as a Smart Edge Cache

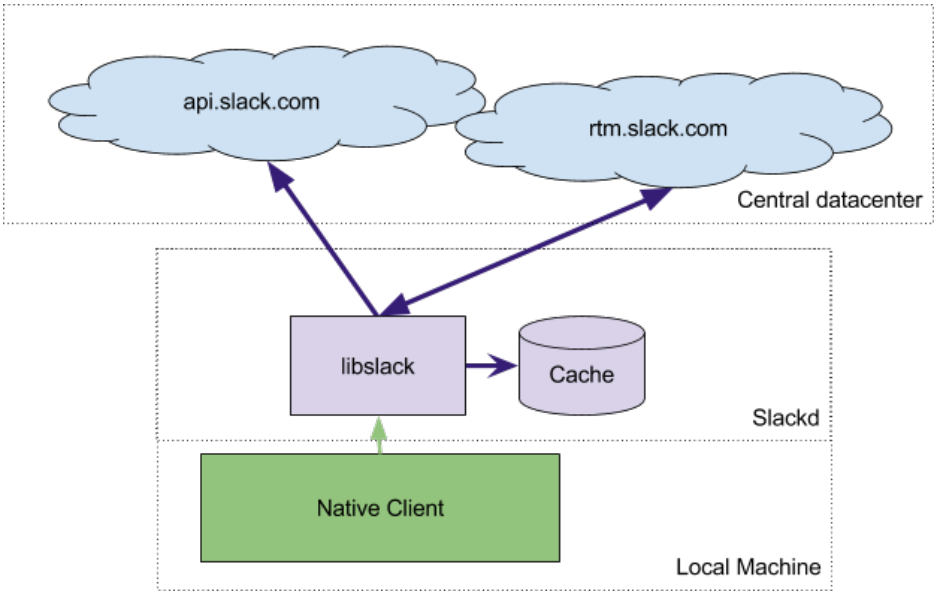
What about the web? Many people use Slack through a web browser. Since we have no way to inject native code into a web browser, it is hard to directly apply libslack.

However, the vast majority of people using Slack in web browsers are doing so over **stable, low-latency, high-bandwidth internet connections**, like office wifi or wired ethernet. Some of these are internet-distant from Slack's orbiting mothership, but we have edge capacity near them.



The layer of indirection here initially seems strange, but it allows us to hide the long round-trips between the edge POP and the central data center. The majority of requests can be satisfied from the local POP's edge cache. The application-aware edge cache is an active proxy, where libslack fits in naturally.

Desktop clients

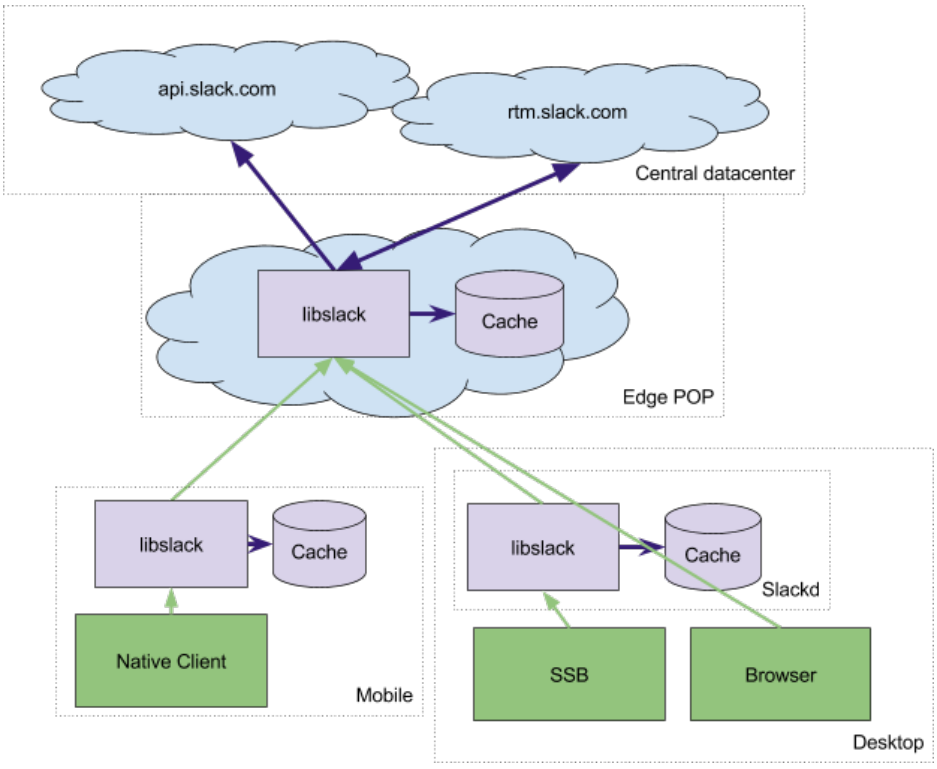


Our desktop clients for Mac OS, Linux, and Windows are conceptually similar to the mobile clients, but leverage the work that has gone into making the web client.

In a libslack-ified world, we could have desktop clients simply talk to an edge-resident caching proxy just as web servers do. However, one of the core limitations of the web client is removed, in that we have the option of installing native code, and using more local machine resources. We're actively prototyping a system we call 'Slackd', which runs the application cache locally, and accesses it over ports from localhost in the desktop apps. This is a similar architecture to the web setup described above, but instead of running the caching proxy in the local edge POP, we'll be running it right on people's hot little laptops. The latency-hiding strategy, and protocol spoken between slackd and the web application, are identical, but we now have an even shorter round-trip time.

Putting it all together

Integrating all of the above pieces together, we get something like this:



We've changed the native mobile client so that its instance of libslack "stacks" onto the edge POP instance. Notice that *all* communication across swathes of the Internet that Slack cannot influence happen to the edge pop, and that in most cases, it is a client version of libslack communicating with a server version of libslack. This gives us the opportunity to experiment with different protocols and wire formats with changes to a single codebase.

libslack is still early in its lifecycle, and the picture we're drawing here might be incomplete. It does not make sense to release into open source at this time, as its API and basic design have not yet settled. But we remain excited about libslack and its future. It has the potential to consolidate effort not only across client codebases, but also between back-end and client. We'll keep you posted here as things progress. Finally, if you're interested in doing this kind of work, we are looking for you too.