Book Proposal: REST and Streaming

Jim Higson, May 2014

Make it fast

There is a common belief, sometimes true, that creating fast systems requires writing fast algorithms. But in this age of cheap, fast CPUs and loosely coupled, remote resources our tasks normally spend more time waiting for I/O than any other activity. For internet computing, the efficient use of I/O will usually deliver more benefit than any other kind of optimisation.

We benefit today from a generation of highly asynchronous servers: Node.js, Netty, NginX. Here the stream, not the whole resource, is the primary abstraction. However, our common REST clients are yet to embrace streaming and do not use a resource until it has arrived completely. We should try to act earlier: in practice there is usually no difference between being reacting *earlier* and being *faster*.

On a Journey: availability, reactivity, and fault tolerance

A passenger checks her email on her phone. As the train moves through the countryside the reception drops while fetching her inbox. Let's look inside the email webapp. The web developer's standard toolkit encourages us to consider connections that terminate early but which were partially successful as if they were wholly unsuccessful. With applications following the path set out by their tooling, our email webapp disregards the partially retrieved inbox without inspection. For our passenger, although several emails have been retrieved, the application behaves as if it got nothing. Later her inbox will be downloaded from scratch, repeating the 90% which was already successfully transferred.

By integrating streaming into REST we step back from this dichotomy of messages being wholly successful or unsuccessful. The resource is conceptualised as having many parts which are useful in themselves, so that each part is handled immediately on receipt. When an early disconnection occurs the content previously delivered has already been handled: no special cases are needed to salvage the remains.

Even if the connection is good, displaying the inbox progressively is also known to improve the user perception of application speed.

A Vote: caching, distribution, and security

We wish to provide a REST service for election results. When a client requests historical data, the static resource is delivered much as we would expect. For data representing an ongoing vote, rather than switch to WebSockets, we keep the same semantic REST interface. Under streaming REST, the best information so far is immediately sent, followed by the remainder dispatched live as the polls are called. When all results are known, the JSON closes as usual to form a standard, complete, cacheable resource. A client wishing to fetch results after-the-fact requests the same URL for the historic data as was used during the election for the live stream. This is possible because cool URLs[1] are semantic: they locate a resource by its meaning, indifferent to when the request is made.

The unification of live and historic data frees the application developer from coding separately for either type. For the election we might display colours on a map - the results could be to-the-second or decades past, but through the entire front-end stack we need only code this once.

A building with fewer doors is inherently more secure, given the same budget available to secure it. Likewise, a service with a single, unified REST interface is easier to secure than one built on REST and WebSockets because all our effort can be concentrated on a single entry-point.

[1] http://www.w3.org/Provider/Style/URI.html