Our Design Philosophy

How does Twitter's new streaming API differ from ESME's design?

Twitter now has another take on a message-streaming API over HTTP, using what looks like a non-HTTP-1.1-compliant form of request pipelining (sending multiple responses over a single open connection). See the documentation at http://apiwiki.twitter.com/Streaming-API-Documentation

The advantage of their mechanism is that it's a smoother experience. What we've done with chunking/long polling is to simulate a stream of data on top of a non-streaming protocol. What Twitter has done is to say "this is a one-way conversation, we've got an open TCP/IP connection, so let's use it." Implementing what they have would require going below the current set of abstractions that Lift provides above the Servlets.

At a practical level, the difference is at one layer... the one dealing with the HTTP requests. At the layers above, events flow either way.

At the basic philosophical level, Twitter's implementation is purer. It treats a stream of information as a stream of information. I like it, but I'm not sure what the benefits would be vs. the development costs of implementing such a mechanism (unless there's an en mass migration of microblogging clients to such a mechanism).

A significant disadvantage of Twitter's design is the requirement of only one streaming connection per account. As much as I dislike the approach of using session cookies to uniquely identify API message queues, it is a heck of a lot better than what is going to happen when Twitter clients start to implement this API, which will be:

1. I log in with Seesmic Web (which has implemented the Twitter streaming API)
2. I receive messages 1, 2, and 3.
3. I log in on a different computer with Twhirl (which has also implemented the Twitter streaming API)
(3.1 Twitter disconnects the Seesmic connection invisibly from the user)
4. I receive message 4 in Twhirl
(4.1 Seesmic tries to reconnect, which results in Twhirl being disconnected)
5. I receive message 5 in Seesmic
6. And so on....

End result: 1 really confused user trying to connect from two banned IP addresses.

I prefer to stick with what Lift provides for the moment. I need to do the conceptual exercise, but on first glance I don't think Twitter's approach results in much of a gain over our approach. Fewer connection attempts, which will help a lot at Twitter-scale, but which I'm not sure makes a big difference at Enterprise-scale.

Another drawback (and I'm really not sure on this one) is that I don't think a lot of HTTP client libraries give easy access to a request that is still open. The design of the queue API is extremely simple from a client programming perspective. I think that's a big upside.

Design-related questions

Question: In an enterprise context it could be an requirement to send a link to someone else pointing to a specific potentially old message in a certain Pool.

Yes. That's perfectly reasonable. That message is like a static file on disk. Once it's written, it remains unchanged until it's deleted. This is an ideal application of a REST-style approach. That's why I've advocated for a “message based” approach first, but a REST/static approach when the message based approach doesn't make sense. What I am opposed to is a “try to make everything fit the REST model” approach to API design.

Question: Would it be costly in your model to get the message nr. X (+ n older messages) in a users timeline?.

A message will exist outside of a timeline. There exists a cache of recently accessed messages. Sometimes there will be a historic message that is referenced and that will be materialized from backing store and rendered. It will likely fall out of cache if it's historical and not accessed again.

Question: I don't understand why we would need to store all entries in a cache, instead of only keeping the last n entries for each user based on some heuristics such as the last 3 days or something. I would somehow expect that the probability that a user wants to see a message is exponentially decreasing with the messages age. For example that someone wants to see a message that is the 1000 newest message in his timeline is probably almost zero.

Some people mine their timelines for information. I agree that some aging policy is necessary as 36B entries will consume a lot of storage in
RAM or on disk, but the last 1,000 is likely too few based on what I have seen of actual user behavior.

In terms of an aging policy in an RDBMS, the cost of aging out old entries is likely to be an index scan or something on that order (DELETE FROM mailbox WHERE date < xxx or a user-by-user DELETE WHERE id IN (SELECT messages > 1000 in mailbox))

Important Links

- Statefulness and algorithms for social networks