Comet: Low Latency Data for the Browser

An old web technology is slowly being resurrected from the depths of history. Browser features that have gone untouched for years are once again being employed to bring better responsiveness to UIs. Servers are learning to cope with a new way of doing things. And I'm not talking about Ajax.

New services like Jot Live and Meebo are built with a style of data transmission that is neither traditional nor Ajax. Their brand of low-latency data transfer to the browser is unique, and it is becoming ever-more common. Lacking a better term, I've taken to calling this style of event-driven, server-push data streaming "Comet". It doesn't stand for anything, and I'm not sure that it should. There is much confusion about how these techniques work, and so using pre-existing definitions and names is as likely to get as much wrong as it would get right.

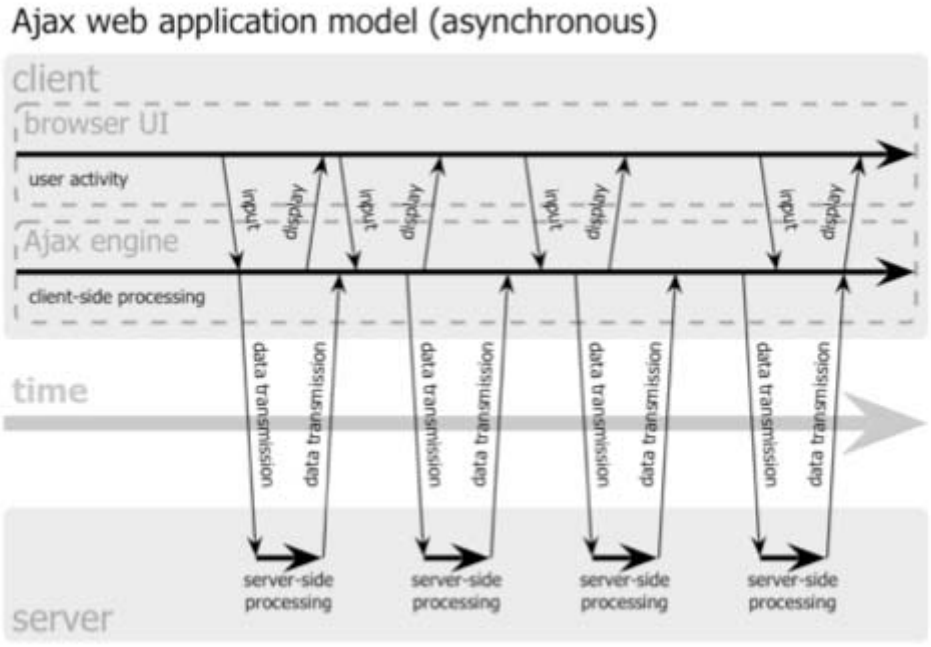
Defining Comet

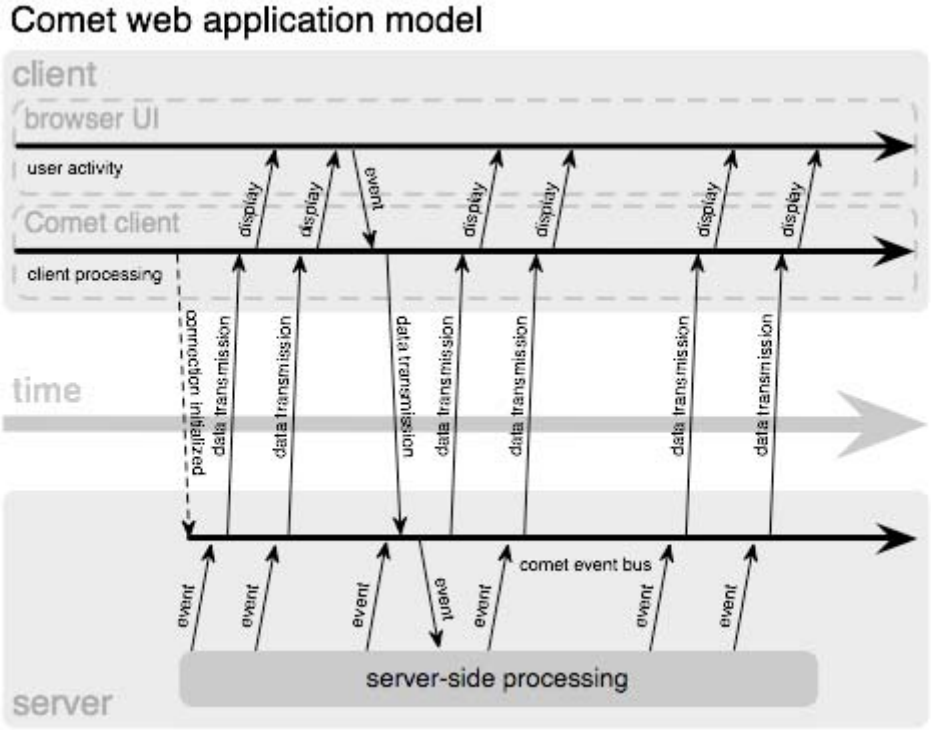
For a new term to be useful, at a minimum we need some examples of the technology, a list of the problems being solved, and properties which distinguish it from other techniques. As with Ajax, these aren't hard to find. A short list of example applications includes:

- GMail's GTalk integration
- Jot Live
- Renkoo
- cgi:irc
- Meebo

So what makes these apps special? What makes them different from other things that might at first glance appear similar? Fundamentally, they all use long-lived HTTP connections to reduce the latency with which messages are passed to the server. In essence, they do *not* poll the server occasionally. Instead the server has an open line of communication with which it can *push* data to the client.

From the perspective of network activity, we can modify JJGs original Ajax diagram to illustrate how Comet differs:





As is illustrated above, Comet applications can deliver data to the client at any time, not only in response to user input. The data is delivered over a single, previously-opened connection. This approach reduces the latency for data delivery significantly.

The architecture relies on a view of data which is event driven on both sides of the HTTP connection. Engineers familiar with SOA or message

oriented middleware will find this diagram to be amazingly familiar. The only substantive change is that the endpoint is the browser.

While Comet is similar to Ajax in that it's asynchronous, applications that implement the Comet style can communicate state changes with almost negligible latency. This makes it suitable for many types of monitoring and multi-user collaboration applications which would otherwise be difficult or impossible to handle in a browser without plugins.

Why Is Comet Better For Users?

Regular Ajax improves the responsiveness of a UI for a single user, but at the cost of allowing the context to go "stale" for long-lived pages. Changes to data from others users is lost until a user refreshes the whole page. An application can alternately return to the "bad old days" and maintain some sort of state mechanism by which it tells client about changes since the last time they've communicated. The user has to either wait until they preform some action which would kick off a request to see the updated state from other users (which might impact the action they wanted to preform!) or request changes from the server at some interval (called "polling"). Since the web is inherently multi-user, it's pretty obvious that regular Ajax imposes usability and transparency hurdles for users. Applications that employ the Comet technique can avoid this problem by *pushing* updates to all clients as they happen. UI state does not go out of sync and everyone using an application can easily understand what their changes will mean for other users. Ajax improves single-user responsiveness. Comet improves application responsiveness for collaborative, multi-user applications and does it without the performance headaches associated with intermittent polling.

But Does It Scale?

New server software is often required to make applications built using Comet scale, but the patterns for event-driven IO on the server side are becoming better distributed. Even Apache will provide a Comet-ready worker module in the upcoming 2.2 release. Until then, tools like Twisted, POE, Nevow, mod_pubsub, and other higher-level event-driven IO abstractions are making Comet available to developers on the bleeding edge. Modern OSes almost all now support some sort of kernel-level event-driven IO system as well. I've even heard that Java's NIO packages will start to take advantage of them in a forthcoming release. These tools are quietly making the event-driven future a reality. This stuff will scale, and most of the tools are in place already.

I'll be giving a more on this topic at ETech and describing the various techniques that Comet applications can employ to push data from the server to the client. As always, I'll post the slides here as well.

The future of the read-write web is multi-user. There is life after Ajax.

Endnotes

First, a word on terminology and its importance. "Ajax" was coined to describe background request/response data transfer. Many of us had worked on solutions to do exactly this, but it wasn't until a simple name and accompanying description were provided that it was possible for people not directly building applications to describe what it was they liked about it. Common terminology acts not only as a shortcut in discussions between technical folks, but also as a bridge for those who may not be able to give a technical rundown of exactly how it works.

As with Ajax, those of us who build technology are now faced with another communication challenge. We have a hard problem for which solutions are available (and have been for some time) but no way to communicate about them. Terminology is again the missing link. Today, keeping an HTTP connection open for doing low-latency data transfer to the browser has no digestible name. When I describe a cool new hack, there's nothing to associate it with. When people say "how the hell did they do that?", we don't have a compact answer. Therefore, in the spirit of improved communication (and not technology invention), I'm proposing a new name for this stuff.

Next, for those who are network-level programmers or are familiar with sockets and/or basic TCP/IP programming, you will probably scoff at the concept of web applications finally getting this kind of datagram packet support. Fair enough. It is however interesting to note that while more responsive UIs have been available on a variety of platforms to date, the Web has "won" the broad majority of market share for most classes of applications in which the browser provides enough native (non-plugin) support to make the performance and/or UI feasible. Comet may be a new name for an old set of concepts wrapped in some pretty grotty hacks, but that in no way diminishes the market impact it will have (and is already having).

Lastly, as current Dojo users might expect, Dojo already supports Comet via dojo.io.bind(). More than a year ago we designed the API with Comet in mind. In the next couple of weeks I'll be showing how bind's pluggable

transport layer can be combined with Dojo's event topic mechanism to provide message delivery on top of a message bus.

This entry was written by alex, posted on March 3, 2006 at 10:37 pm, filed under javascript, programming, webdev. Bookmark the permalink. Follow any comments here with the RSS feed for this post. Post a comment or leave a trackback: Trackback URL.