Ajax On-Demand

Mission-critical business applications need immediate and guaranteed data delivery. Attempts have been made (like Comet) to adapt AJAX techniques to support such applications. These attempts have been based on using unreliable push techniques or resource-greedy polling techniques. However, Ajax On-Demand is an adaptation of AJAX for such applications that doesn't suffer from these disadvantages.

A White Paper from Exadel, Inc.

Exadel, Inc.
1850 Gateway Blvd.
Suite 1080
Concord, CA 94520
925-602-5561
www.exadel.com
info@exadel.com

The concept of upside-down AJAX (a.k.a. Reverse Ajax, Comet, or AJAX Push) has been talked up quite a bit lately, so let's see what all of this interest is about. The business proposition is actually pretty clear. There are types of applications that require notification to a user about changes on the server as soon as possible. As a rule, such applications belong to the class of mission-critical business applications, which need immediate and guaranteed data delivery. Examples of this class of application are broker's software, security monitoring, and load balancing. In a pre-AJAX era, there was no doubt that the whole-page refreshing of a Web application was not suitable for these purposes. When AJAX came on the scene, people started to look for a way to adapt it for use in this area where the thick client was still king.

Historically, Web technology assumed that only the client initiated the connection for communication with the server and that this connection was closed as soon as the request was completed with s corresponding response. The AJAX technique itself did not break this rule. It just enabled the partial update of a page with communication still initiated by the client and then dropped when an AJAX response is received.

The first approach to adapting AJAX for mission-critical applications (and actually the only one that it is popular now) is to emulate the situation where the response is not complete for a long period of time. Then, data can be transferred to the client again and again using this once opened connection.

Michael Mahemoff uses an interesting comparison to summarize this important point:[1] "this technique allows the server to start answering the browser's request for information very slowly. Extremely slowly. Actually in the same way I used to answer my French teachers at school, it starts the reply but never actually finishes."

Comet Overview

Comet is one version of this approach. In an article introducing Comet,[2] Alex Russell says that "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". He also provides an illustration of how Comet differs from the regular AJAX model.

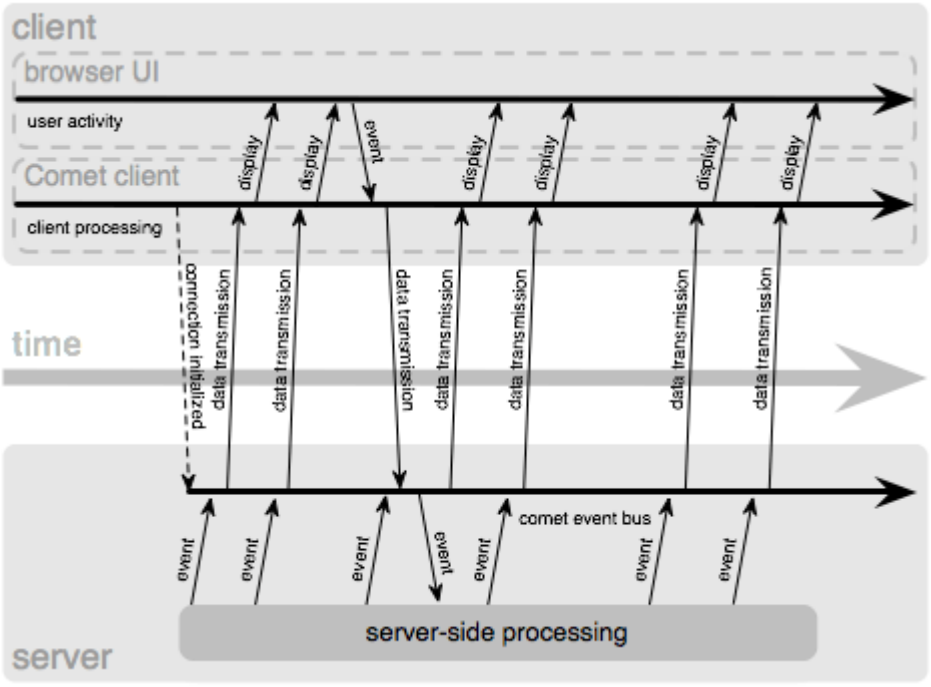

Figure 1: The Comet Web application model

As we can see, instead of short term processing of Ajax requests initiated by the client, Comet assumes the long-term server-side processing that uses a single connection named the Comet Event Bus, to deliver responses based on an event onto the server right after the event occurs. The client also might send data to the server that is delivered to the server as an event.

The entry for "Reverse Ajax" in Wikipedia[3] declares that this technique makes web pages feel more responsive by providing real-time information in a Web page. This is meant to increase Web page interactivity, speed, and usability. Therefore, it seems to match the criteria for the mission-critical applications we mentioned at the beginning of this article. Is the technique scalable enough, though? Alex Russell dedicates an entire paragraph to this topic. We will quote it here to avoid any misreading:

> 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.

Further in his article, Alex emphasizes that the technique relies on the server side for scalability including both hardware and software.

Weaknesses of the Existing Technique

Anybody who is reading this has probably had the experience where the connection is suddenly closed for various reasons (or without any reason at all) even during regular whole-page updates. Thus, the suitability of a long-term connection for mission-critical, real-time applications might be questioned.

The reliability issue itself is just the tip of the iceberg. There is also the fact that keeping a connection alive with each connected client consumes a significant amount of server resources. It requires memory to be allocated for each connection. Even though the resource management might be controlled by Java NIO, this does not solve all of the problems. These problems include proxy servers and firewalls closing non-active connections or rebooting because the long-term connections are recognized as hung-up.

Additionally, the HTTP 1.1 specification[4] says: "A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy." Therefore, the number of possible connections is limited and exceeding this limit might also be a signal to drop the connections.

Let's keep in mind the specific character of the TCP stack (per the "Nagle" algorithm[5]). When the sending data size is less than the package size (usually 1460 bites), to avoid data fragmentation, the server will put the data into the buffer and wait a certain period of time to collect the rest of the data. Server can set a TCP_NODELAY option on your own Unix system (Windows TCP/IP stack does not support it), but you cannot do it on each of the servers located between your server and your client computers

So, the weakness of the technique cannot be resolved just by using NIO. The weakness is between the server and client, inside the existing proxies and firewall that are not under your own control. A technique, that sounds good from a developer's point of view, sounds inappropriate from a network engineer's point of view.

Here is another example of this. Squid is one of the popular proxies used in hardware network devices. It has a timeout of 15 minutes. The Squid's manual says "The read_timeout is applied on server-side connections. After each successful read(), the timeout will be extended by this amount. If no data is read again after this amount of time, the request is aborted and logged with ERR_READ_TIMEOUT. The default is 15 minutes".

We will make one final point about the server side. Keeping long-term connections alive directly contradicts the actual Java EE specifications for clusterization.

In conclusion, the whole AJAX Push approach, based on a long-lived HTTP connection, is not reliable and does not guarantee data delivery, so it cannot be used for mission-critical applications. (In the future, as the new Java Servlet 3.0[6] spec is developed and implemented, this may not be as much of a problem.)

Other Alternatives

At first glance, the real alternative to push is polling. However, sending a periodic request to the server has a huge disadvantage. If the polling becomes too frequent, most requests will be made for nothing because the server has no data for update. Such unproductive round-trips to the server still carry costs absorbing server and network resources and taking time to establish the connection, authorization, session restoring and saving, and so on. On the other hand, if you set a period for polling that is too long to avoid such problems, you risk missing the exact time when data changes. Then, the poll has useless resources utilization on one side and latency on the other side.

DWR (Direct Web Routing: "Easy Ajax for Java") uses a piggyback technique which is something in the middle between push and poll. The client first sends an AJAX request similar to the poll request. However, the server does not respond immediately, but waits until the data is ready. After that, the server will push data to the client using this open connection. Instead of pure push, the connection is closed and client sends a new request again for the next possible data changes.

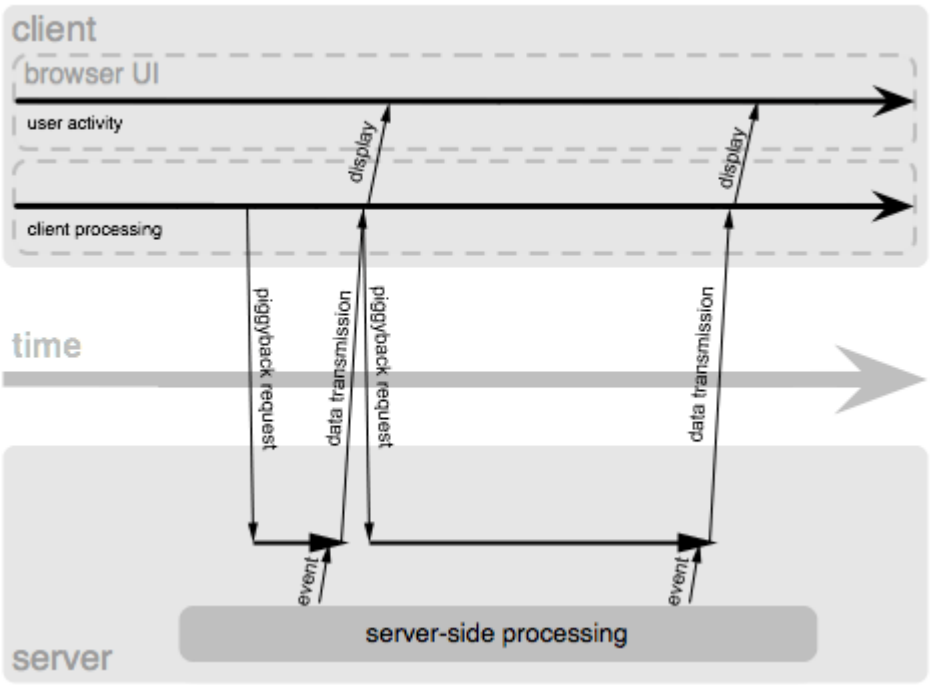

Figure 2: The piggyback Web application model

Piggyback solves the problem of latency. However, another set of problems associated with the push approach still exists. Although a piggyback connection is shorter than the one used with push, it is still at risk of being closed because of a protracted idle period. To use piggyback, the server needs to authorize a request and allocate resources for keeping the connection alive for an unpredictable period of time.

Introducing "Ajax On-Demand"

UNIX has a utility called "biff."[7] When a new mail message is delivered, **biff** alerts the recipient so he can read it immediately. Originally, the utility was named after a dog who barked at the mailman, so the host would only go to the mailbox when mail had really arrived. Avoiding the technical details of the utility implementation, let's look only at the main idea here: the use of a resource-conserving way to check for an update along with a regular and reliable way to receive this update.

Now, let's move to the Java EE world. Java Messaging with Message Driven Beans[8] is the only legitimate approach for working with long-term processes in EJB in a multi-threaded environment. From *Message Driven Beans:*

> The JMS clients send messages to message queues managed by the server (e.g., an email inbox can be a message queue). The message queues are monitored by a special kind of EJBs -- Message Driven Beans (MDBs)…

What we call the "Ajax On-Demand" implementation uses these MDBs to support a two-phase model. In the first phase, the client sends short requests to a special servlet that gathers information about available message IDs. Those requests come by HTTP and contain only the HEAD part of a request. This way does not require authorization, restoring the session, or other resource-related operations. (The goal of these short requests is to verify that the update has arrived.) In the second phase, a regular AJAX request takes the data and updates the client page in accordance with this data.

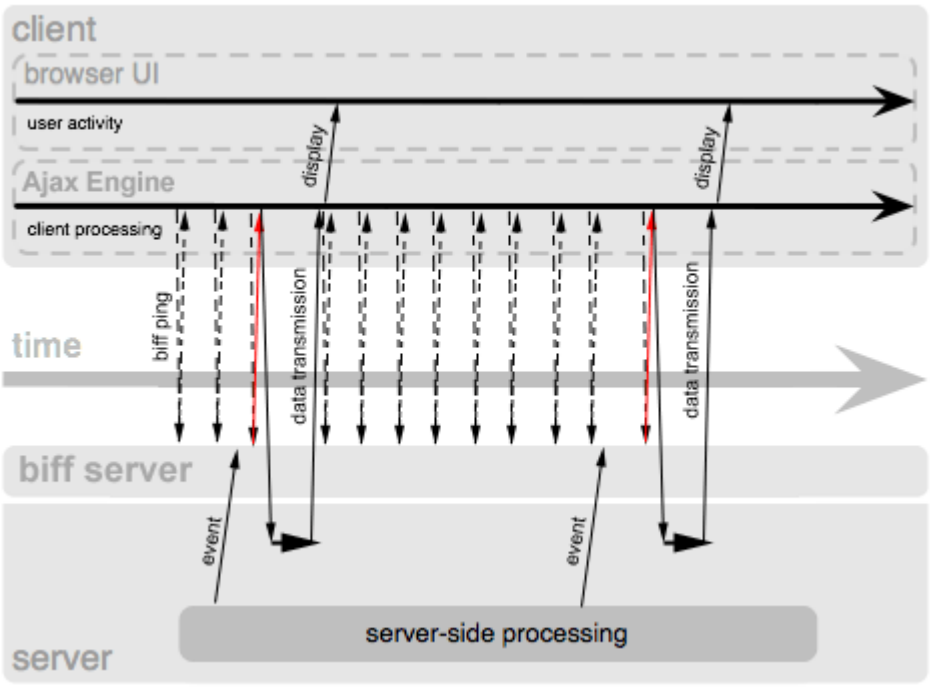

Figure 3: The Ajax On-Demand Web application model

The first-phase requests are extremely light-weight in terms of time and resources. The estimated response time for a first-phase request will be equal to the response time of a TCP/IP ping. For example, even a single Tomcat 5.5 server on a 3 Ghz processor with a 10 Mbits channel is able to process up to 10,000 such requests.

Conclusion

The Ajax On-Demand approach seems the best alternative to both unreliable push techniques and resource-greedy polling techniques. This approach can be very useful for real-life applications that demand "live" page updates based on asynchronous events on the server. Ajax On-Demand does not require extensive resources on the server and does not put any additional restrictions on client location or network topology.

---

[1]Michael Mahemoff. "Reverse Ajax with DWR." *ajaxian.* May 24, 2006. <http://ajaxian.com/archives/reverse-ajax-with-dwr>.

[2]Alex Russell. "Comet: Low Latency Data for the Browser." *Continuing Intermittent Coherency (Blog).* March 3, 2006. <http://alex.dojotoolkit.org/?p=545>.

[3]"Reverse Ajax." *Wikipedia, the free encyclopedia.* <http://en.wikipedia.org/wiki/Reverse_Ajax>.

[4] "Connections." *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1.* <http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html>.

[5] *RFC 896: Congestion Control in IP/TCP Internetworks* . <http://www.ietf.org/rfc/rfc896.txt>.

[6] *JSR 315: Java™ Servlet 3.0 Specification.* <http://jcp.org/en/jsr/detail?id=315>

[7]"biff." *Wikipedia, the free encyclopedia.* <http://en.wikipedia.org/wiki/Biff>.

[8]"Message Driven Beans." *JBoss EJB 3.0 and Extensions.* <http://trailblazer.demo.jboss.com/EJB3Trail/serviceobjects/mdb/>.