Chapter 4

Comet architecture for web applications

Sergi Baila and Vicenç Beltran

Abstract

The last two years have seen a revolution on the way web applications are developed. The popularization of new techniques under the acronym AJAX (Asynchronous Javascript and XML) has made web applications a lot more interactive and closer to desktop applications. At the core of this new approach is the ability of a web page script to send requests to the server without user prior action. This breaks one of the limitations of web applications and the HTTP protocol, and now a web application can trigger an asynchronous partial page update which makes applications a lot more responsive and interactive, and also hides latency effects.

This technology has evolved and has become quickly a new foundation for developing web applications which are closer to desktop applications. Web mail, calendars, instant messaging... Also, common bussines software is starting to be developed as a web application, even on intranet scenarios. However, AJAX is still limited by the underlying HTTP protocol and it's request/response cycle. On this known

client-server architecture the browser is the one which always initiates actions (send requests). Desktop application frameworks, based mainly on the MVC software pattern, implement GUIs which are based on an event-response model. Events can be fired on the client side but also on the server side. Web applications face a significant problem here. Perhaps even bigger than common desktop applications which tend to be single user whereas web applications are starting to be designed from the beginning as multi user applications. So the need for a server propagated event model is more necessary.

Comet is a new approach which uses an open idle connection, mainly unused, until there's a need for the server to push information to the client. This allows the push of events from the server to the client, so the gap between desktop applications and web applications is further reduced. But keeping an open connection per client breaks classic servers' scalability where you have one thread per connection. New server implementations based on asynchronous I/O are already available, which can handle thousands of connections with just a pool of threads.

This chapter introduces AJAX and Comet architectures, the new frameworks, and the servers which implements them on top of asynchronous I/O. We also analyze the new problems introduced by these technologies. AJAX relies completely on JavaScript, the DOM model, CSS... all web technologies which are now starting to see standards compliant products. Portability is one of the first problems encountered by an AJAX developer even between minor revisions of the same browser. Also, usability of web applications suffers from the AJAX approach as existing mechanisms for disable people aren't prepared yet for this new technique.

4.1 Introduction

On February 18th 2005 Jesse James Garret published a short article[1] on his company website coining a new buzzword on the internet world. No one suspected that essay would be seen later as the first milestone of a revolution on the way we understand web applications. There was no new technology, because the ingredients were present for some time, nor there was no new product. Instead he pointed to existing products

like Google Suggest or Google Maps. But that short name, AJAX, was rapidly spread among technological publications, blogs and sites.

But that was just the name. Most people, me included, had the first encounter with the new technology and a glimpse of the possibilities behind with a sub project from Google called Google Suggest (back in 2004). It was a simple product, just the google page with a twist added: as soon as you start typing the web page started to show suggestions of searches along with estimated result count. So you typed "car re" and google suggests "car rentals" but also "car reviews" and so on. Most non technical people saw it's speed and ease of use. We technical people were amazed as how it broke the classic HTTP request response and full page reload mechanism.

The magic behind relates to the XMLHttpRequest object, which was created by Microsoft (as the ActiveX object XMLHTTP) and later (2002 and beyond) was implemented on Mozilla and some other browsers. This object allows to send an HTTP request and retrieve the response as an asynchronous javascript method call. This is what Google Suggest uses to send a request to a server each time there's a keystroke and then parsing the HTTP response.

So the evolution of the usage and impact of this technology has been somehow exponential. It took nearly four years to reach a side project on google, then some other sites started using similar effects (GMail, Google Maps, Flickr, ...) and a name was adopted on 2005. That same year saw the explosion of the technology. This can be considered the first milestone of the future web applications.

There has been always a clear gap between web applications and desktop applications. Before 2005 the answer to the question "do we need a web application or a desktop application?" was easily answered because web applications were very poor and the only advantage was that they are distributed and easily available applications with a very thin and common client. Then came AJAX and applications like GMail which broke the limitation of the full page reload model based on the HTTP request response model. It was not the first step nor the last, but an important one. The HTTP protocol was not designed as a foundation of general purpose applications. Actually, it was not designed with any application on mind, even classic web applications. One of the first important limitations resolved in the past was the stateless property of the protocol. Today with the use

of cookies or URL rewriting, and with every web developer framework supporting sessions, this seems an easy task. We are here to take a look at the next step to narrow the gap between desktop applications and web applications: Comet or how can you build a event-based web application.

4.2 Background

In this section we provide the necessary background information for those not familiarized with web technologies. Some concepts of the HTTP request response model are presented. Then we introduce the JavaScript environment available on web browsers and we dive into AJAX as the precursor technology for Comet.

4.2.1 HTTP model

The Hypertext Transfer Protocol is a communications protocol designed mainly for the retrieving of HTML pages and accesory elements (CSS pages, images, etc.). It is a request/response client/server protocol. That means that the model is clearly and strictly defined[2]: the client (browser) sends a request to the server which reads the whole request, processes it and returns back a response (see figure 4.1). An HTTP client (known as the user agent) establishes a TCP connection on port 80 (the standard one, but could be any) of the web server in order to send the request and retrieve the response. The server listens to that port and can serve multiple clients simultaneously.

The HTTP model has several limitations for developing a web application. It is a stateless protocol, bonded to a strict request/response cycle. The stateless problem was solved with the use of cookies or URL rewriting to keep a session between the client and server. Until recently, that was the foundation for developing web application, and is what we call here the classic model (figure 4.1). On this classic model each time there was an action from the user the browser sent a request to the server which resulted on a new page loaded. This is what we call the full page reload model. Given current network latency, even on a local area network, is very difficult to develop a web application with the same funcionality as a desktop application. We

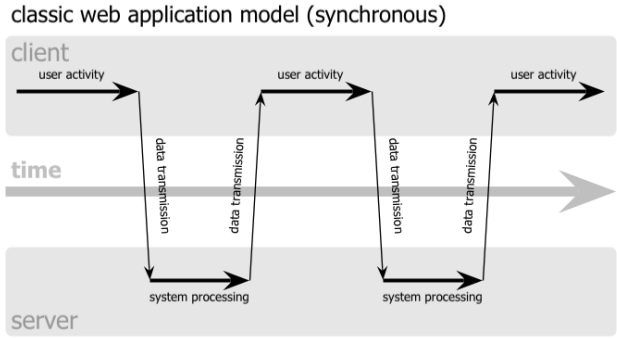

Figure 4.1: Classic HTTP model

will see how AJAX solves this and brings us the next model.

Another problem arises with a Comet architecture that we will explore on a next section and is introduced by a HTTP protocol limitation. The protocol [2] limits (by suggestion) the number of simultaneos connections from a user agent (browser) to the server to just two. Using new techniques like HTTP pipelining and a classic or AJAX model this is not of much concern. But with a Comet model using a permanent connection there's just one left.

4.2.2 JavaScript

JavaScript is nowadays a real distributed execution environment, being the standard language for script execution inside web pages. Its real name is ECMAScript[3] and its evolution is tightly close to that of the web browsers. This scripting language allows, within a browser, to manipulate most of the components of the web page (the document structure via a DOM(Document Object Model)[4] interface). It is a quite powerful language which not only can manipulate the Document Object Model but also can be used to listen on events, use it asynchronously, parse XML and even send HTTP request from within a web page without triggering a complete reload (this is the base for AJAX).

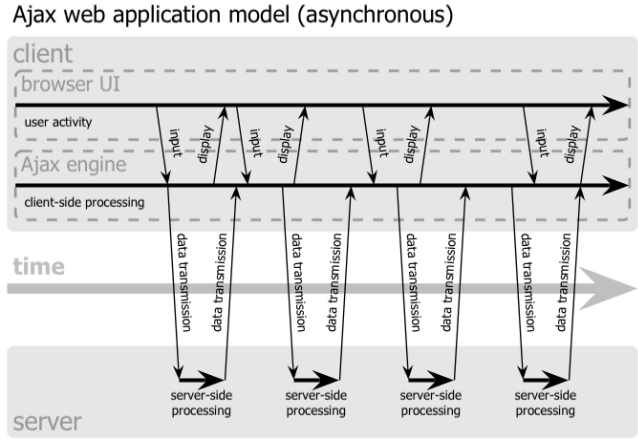

Figure 4.2: AJAX HTTP model

4.2.3 The AJAX model

For years web developers had faced the problem of having a full page reload every time they wanted to get or set new data to or from the server. But the advent of the XMLHttpRequest object and it's easy asynchronous usage led quite rapidly to a new breed of web applications. There was no more a synchronous and closed request and response cycle. We can now have a request sent on the background which response triggers a partial change (thanks to the ability of JavaScript to manipulate the page through the DOM). We can have then partial page updates, background server communication and requests made by programming logic and not subject to user interaction. We will call this the AJAX application model as seen in figure 4.2.

The AJAX acronym [1] stands for Asynchronous JavaScript And XML. The original concept was suposed to use XML as the language to encapsulate the response where the JavaScript has the ability to parse it and modify the page state and contents via the DOM interface. Some applications use that model, but most of the time developers use a simpler and stripped down model where the response is just HTML (partial page) and the action to do is just replace some part of

the page. Also a common usage is encapsulating JavaScript code on the response so the server can trigger any event on the page. All of this work has been greatly simplified with the development of JavaScript frameworks like Prototype or Dojo.

4.3 Introduction to Comet

As J.J. Garret coined the word AJAX there was also a blog post from Alex Russell [5] where he tried to follow the same path coining the Comet term to refer to the possibility of the server to send events to the client without having to wait for a request from the browser to arrive. Also as with the AJAX term, there were prior works on the area to solve the problem of a web application being unable to receive asynchronous events from the server. We're just referring to this milestone as a signal of the maturity of the idea.

The reason for this need was actually a consequence of the exit of the AJAX architecture. A great number of new highly functional web applications were developed with AJAX and both developers and users wanted to push developments further [6]. But as interactive it was an AJAX application it lacked a core mechanism from desktop application: real time updates. Developers notice that it was necessary to propagate events from the server to the client in order to have an event-driven web application.

The problem again was the HTTP protocol. It's a client-server protocol, without option to the server to contact the client. Also, given the diversity of networks and connections between browsers and servers, building any mechanism for the server to open a connection to the client is out of the equation.

So there's only one solution possible (without severe modifications of the underlying protocol). To have an open connection idle just waiting for an event on the server (any Comet client should have one then). So when the server has to send an event to one, some or all of the clients, it just uses the open connection (which is a standard HTTP connection). You can see figure 4.3 for a diagram of the Comet model with an HTTP streaming connection technique. The beauty of the solution is that it works. And works without modification of clients, servers, protocols, etc. Unfortunately, the problem introduced

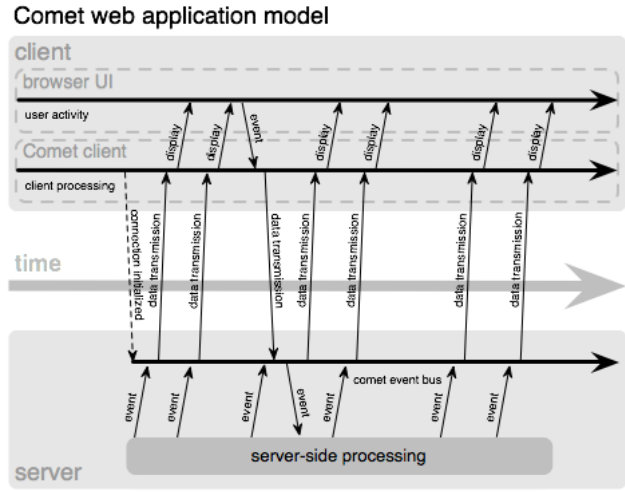

Figure 4.3: HTTP streaming Comet model

is of a different nature. The servers suffer from a scalability problem with this architecture. [5] [7] [8] [9]

4.3.1 Comet architecture

The tradicional implementation of web servers, specially an application server (like a JavaEE server) uses a pool of threads to manage incoming connections. When a new connection arrives it's established and a thread from the pool is assigned to the connection. Request is read, code is executed and a response is generated and sent. Then, the thread returns to the pool. This is designed under the assumption that requests are short in duration but intensive in computing resources. But a Comet connection is established and is expected to be long in duration (can be several minutes) and very low CPU or memory intensive. The connection is only required for sending events to the client and just keep the connection open.

A classic web application can handle easily on the order of tens of thousands of simultaneous users, because users are not sending re-

quests all the time. So the number of active connections on the server is always a fraction of the users of the application at any given time. With a Comet architecture each user on the system is an open connection on the server and a thread (with a classic model) which is locked to the opened connection. Even if it's not doing something, any server has problems managing tens of thousands of threads.

The servers need to be redesigned around this new problem. The solution comes from a know mechanism, asynchronous I/O, which has existed in modern operating systems for a log time. C programmers know it as the select() or poll() system call. Java, for example, has support for it since version 1.4 with the introduction of the java.nio packages. [10] [11]

The new design decouples the one to one relationship between connection and thread. There's also a thread pool, but threads are also used to process active connections, not connections which are not handling data. Of course, developers of server side components need to do some modifications, but they're only needed on the comet handlers.

Besides scalability on the server the Comet architecture introduces another subtle problem on the client side. The HTTP protocol [2] limits on 2 the number of simultaneous connections to a server. Using at least one for a Comet connection leaves the whole page with just one connection. As the page is probably using the AJAX model it's obvious that a complex or simply slow response would block all the other connection and leave the page unable to send any other request as we have the two connections busy: one for the comet connection and another waiting for the slow reponse to an AJAX call. So this is nearly impossible to circumvent but it can be alleviated. One necessary step is to stream all Comet communication to the same and only connection as no page can afford to have the two connections busy on different components.

We've seen that the Comet architecture posses a series of challenges both on the server and the client. We are now presenting the internal details and work done on the model.

4.3.2 Bayeux protocol

As a non standarized architecture Comet faces significant interoperability problems. Actually there are as protocols as implementations.

Some of the major names behind certain libraries and servers are pushing for a standard protocol of communication between a Comet client (JavaScript library) and a Comet server component. The result of this is the Bayeux protocol [12] with the Dojo Foundation behind it. There's also work in progress from the authoritative source W3C for HTML 5 server sent event listeners [13] but without any real work impact yet.

The lead person behind Bayeux is Alex Russell from Dojo which guarantees a certain level of notoriety for the protocol. As he states in his first post [**?**] about Bayeux: "One of the biggest problems facing the adoption of Comet is that it's, by definition, not as simple. It's usually not possible to take ye-old-RESTian HTTP endpoint and suddenly imbue your app with realtime event delivery using it unless you want your servers to fall over. The thread and process pooling models common to most web serving environments usually guarantees this will be true. Add to that the complexity of figuring out what browsers will support what kinds of janky hacks to make event delivery work and you've got a recipe for abysmal adoption. That complexity is why we started work on Bayeux."

As they define it: "Bayeux is a protocol for transporting asynchronous messages over HTTP. The messages are routed via named channels and can be delivered: server to client, client to server and client to client (via the server)". The protocol specification is in a very initial stage but has seen some support from the community which see it as a good way to push the architecture support and ease of development further.

The protocols tries to address the main problems associated with the Comet architecture. It uses JSON (JavaScript Object Notation) as the data interchange format to define the messages. Those messages are clearly defined on the specification and cover all the low level technical details needed as the handshake, connection negotiation, channel subscription, reconnection, etc. The standarization of the messages allow the development of interoperable client libraries and server components. Further, it ensures that key concepts like negotiation and reconnection are taken into account even for simple developments. The protocol also introduces a versioning system which allows to negotiate between client and server for a preferred protocol level in the same way as the HTTP negotiation works.

A key concept on the protocol is the multiplexing of different endpoints for comet components via a mechanism of channels. Each message sent with the protocol has a channel destination, which helps alleviate the two connection limit problem of HTTP. So having different server components accessed via Comet no longer wastes multiple connections but just one. It also helps server components to clean up and separate things. Another helpful introduction is the identification of each client with an autogenerated id, much the same way as an HTTP session id.

Another advantatge of a standard protocol is the support for multiple connection models and a negotiation protocol. The Comet architecture is really a hack over the limitations imposed by the HTTP protocol, so different connection methods are not only necessary but desirable to support as many clients as possible. We're going to take a look at the different Comet connection models.

4.3.3 Comet connection models

We've seen that a Comet architecture needs somehow a permanent connection to the server in order to be able to receive server generated events. But the handling of this connection can be different on the way is managed mainly in the client side but also affecting the server side. Choosing the right one is not an easy answer [9].

The first one and the first used before the advent of Comet or even AJAX is the polling connection model (figure 4.4). This can't really be considered a Comet model because it's not receiving the event when it happens but we're incluing it here as a base idea which has been used in the past and is a perfect example of an scenario where Comet can really help.

Of course this model has a clear scalability problem. As it has not the problem of keeping an idle connection, the number of polling request received on the server can be extremely high with a high frequency value which will be desirable in order to make the application responsive. This model can work with a small number of users even with an update every 2 or 3 seconds. This model also has other drawbacks. There is an overhead of a new request and response. Also, probably the main problem, depending on the application usage some or many of the connections could be empty, just the client asking the

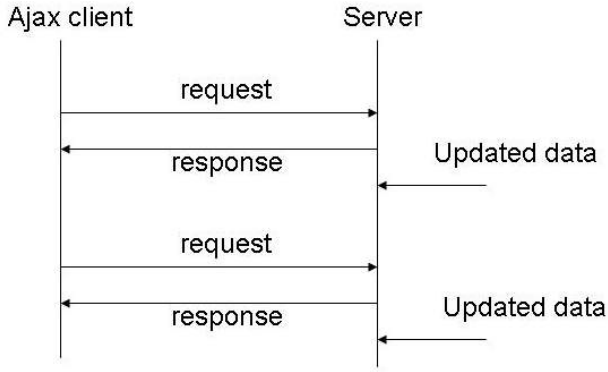

Figure 4.4: Polling connection model

server for events and getting a negative response. This overloads the server and the network for nothing.

Long polling is an evolution of this technique which solves the problem of the void requests because the server only responds when there are data to. Meanwhile, the connection is just waiting. You can see on figure 4.5 that the model just sends a request waiting for data on the server which also waits until there's some event. So the requests are not returning void never, but you have on average as many connections as clients on that page.

As this model solves the void response problem it may introduce an scalability problem on those servers which block on the request and have a classic 1:1 mapping between threads and connections. That's because if you want your application to be able to scale to tens of thousands (or more) simultaneous users you will have at least as many connections on the web application. So, for example, having 10.000 users on your application means you will have 10.000 AJAX connections because of your long polling model. That on a classic server translates to 10.000 threads just waiting (sleeping) on each connection doing nothing but wasting resources. Even if your OS, TCP/IP stack and so supports that it's unlikely your application server do. Fortunately, new servers (Grizzly [14]) and revisions on old ones (Jetty [8],

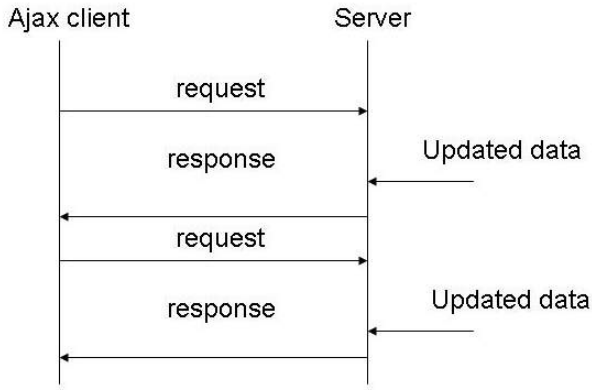

Figure 4.5: Long-polling connection model

Tomcat, Apache) implement what's called Asynchronous Request Processing [14] which is based on non blocking I/O, a mechanism found on recent revisions of OS and libraries [15] [10].

Of course this isn't the perfect solution. If the server is pushing events fast enought you will find yourself in a similar scenario as with the polling model where you have several connections and a big overhead for each request/response cycle. Also both models suffer from the network latency specially as they need to send a new request for each response (event) received. There's also some bandwith wasted on the multiple requests.

This leads us to the third model called HTTP streaming [16], a model similar to long polling but without closing the connection even after getting a response (see figure 4.6). The trick here is to use a transfer mechanism from HTTP [2] called chunked transfer encoding, which allows to send a response build up of blocks of data (chunks) without knowing the amount of data and lenght of each chunk in advance. This fits exactly to a series of events on the server which need to be propagated to the client without knowing in advance the number of events, the lenght or most important when they will happen. This model greatly helps leveraging the network usage as eliminates the overhead of multiple requests and reduces the latency because the

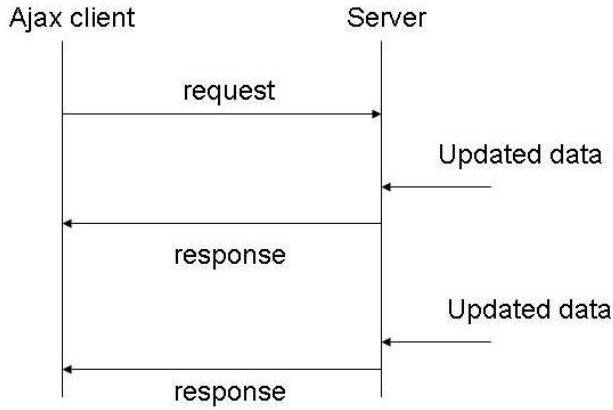

Figure 4.6: HTTP streaming connection model

response can be sent without waiting for a request to end.

Even HTTP streaming is of course not exempt from caveats. Not only the server should support thousands of connections with a limited number of threads on the pool, on big scenarios you can find yourself with too many events which can't be correctly propagated to the clients because of network congestion. So some kind of event throttling should be considered.

Surely there are also some challenges that need to be addressed. For example even with the HTTP streaming connection model there's no way that the client can send events to the server on the opened channel. A bit ironic that the standard way of communication doesn't work, but in this case is more a limitation of the XMLHttpRequest JavaScript construction that needs a complete request before starting the transaction.

4.4 Scalability issues

There's no extensive work on the AJAX and Comet impact on performance in web environments, and existing work has very preliminary results [17]. Even without extensive experimental evidences the one

to one mapping between connections and threads doesn't seem the best idea. And not only Comet HTTP streaming or long polling connection models benefit from it, some testing indicates that certainly most kinds of web application can benefit from it. J.F. Arcand, one of the engineers behind Sun's Grizzly server, has done [18] some syntethic test and real benchmarks over Grizzly asynchronous request processing module based on Java no blocking library java.nio. The throughtput of static files and simple JSP and servlets is quite the same (see figures 4.7, 4.8 and 4.9) but a classic connector (Tomcat Catalina) needs a 500 threads pool to match a 10 threads pool on a ARP connector. Testing the maximum number of users that a website can handle (figure 4.10) with a maximum response time of 2 seconds on 90% of request and an average think time of 8 seconds show a clear winner of the non blocking model because there is less context switching and more available memory with the far lesser number of threads of the second model.

4.5 Comet frameworks

As the AJAX and Comet technologies evolve and popularize we see an increasing number of frameworks appearing. Actually, the number of AJAX frameworks is growing very quickly [19], perhaps because it's quite new technology and the market hasn't done the natural cleaning for the best ones. Anyway just a very small subset of this frameworks support Comet so we're centering on the most popular ones. We will first take a look at some of the developer libraries to implement Comet solutions. We will introduce Pushlets, a combined library which uses a client JavaScript component and a Java servlet for the other side. As a different example, Dojo is a more general purpose framework written completely in JavaScript without server side components. The Comet part is solved implementing the Bayeux protocol.

We will then introduce three of the server which implement some kind of asynchronous request processing using non blocking I/O. Grizzly from Sun is the web container for their JavaEE server GlassFish and is built from the ground thinking on asynchronous request processing. Jetty is a very popular servlet and JSP container which was one of the first (if not the first) to implement a solution with its Continuations mechanism. The newest Apache Tomcat version 6 includes

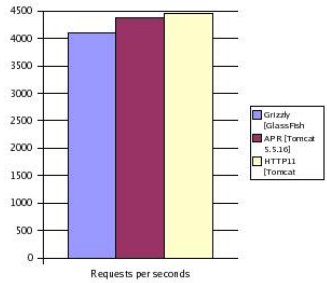

Figure 4.7: ARP 2k file static performance

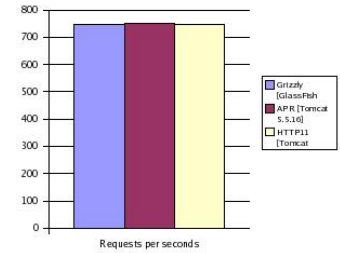

Figure 4.8: ARP 14k file static performance

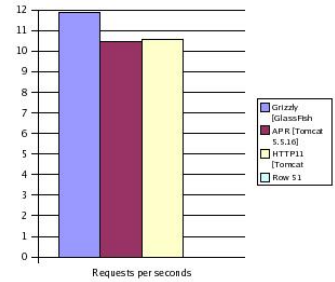

Figure 4.9: ARP 954k file static performance

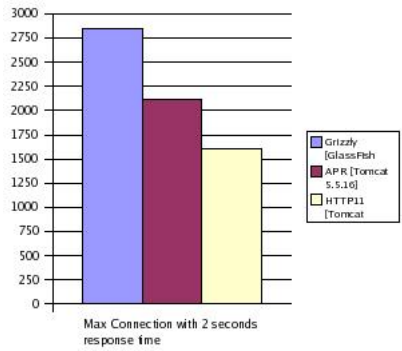

Figure 4.10: ARP maximum number of simultaneous connections with 2s response time

an ARP connector.

4.5.1 Client libraries

"Pushlets are a servlet-based mechanism where data is pushed directly from server-side Java objects to (Dynamic) HTML pages within a client-browser without using Java applets or plug-ins. This allows a web page to be periodically updated by the server. The browser client uses JavaScript/Dynamic HTML features available in type 4+ browsers like NS and MSIE. The underlying mechanism uses a servlet HTTP connection over which JavaScript code is pushed to the browser. Through a single generic servlet (the Pushlet), browser clients can subscribe to subjects from which they like to receive events. Whenever the server pushes an event, the clients subscribed to the related subject are notified. Event objects can be sent as either JavaScript (DHTML clients), serialized Java objects (Java clients), or as XML (DHTML or Java Clients)." [20]

The Dojo toolkit is a modular open source JavaScript toolkit (or library), designed to ease the rapid development of JavaScript- or Ajax-based applications and web sites. It was started by Alex Russell in 2004 and is dual-licensed under the BSD License and the Academic Free License. The Dojo Foundation is a non-profit organization de-

signed to promote the adoption of the toolkit. [21]. Alex Rusell is the responsible for the word Comet [5] and one of the people behind the Bayeux Protocol [22] [12]

4.5.2 Server solutions

Grizzly is the HTTP server component for the new reference JavaEE server Glassfish from Sun. A description of Grizzly from one of his creators: "Grizzly has been designed to work on top of the Apache Tomcat Coyote HTTP Connector. The Coyote Connector is used in Tomcat 3/4/5 and has proven to be a highly performant HTTP Connector when it is time to measure raw throughput. But as other Java based HTTP Connector, scalability is always limited to the number of available threads, and when keep-alive is required, suffer the one thread per connection paradigm. Because of this, scalability is most of the time limited by the platform's maximum thread number. To solve this problem, people usually put Apache in front of Java, or use a cluster to distribute requests among multiple Java server. Grizzly differ from Coyote in two areas. First, Grizzly allow the pluggability of any kind of thread pool (three are currently available in the workspace). Second, Grizzly supports two modes: traditional IO and non blocking IO." [15]

Jetty is a 100% pure Java based HTTP Server and Servlet Container. Jetty is released as an open source project under the Apache 2.0 License. Jetty is used by several other popular projects including the JBoss and Geronimo Application Servers. This server was probably the first breaking the one thread per request mapping with it's Continuations [8] and provide a sort of Comet server framework before even the concept was clear.

Apache Tomcat is a web container developed at the Apache Software Foundation (ASF). Tomcat implements the servlet and the Java Server Pages (JSP) specifications from Sun Microsystems, providing an environment for Java code to run in cooperation with a web server. It adds tools for configuration and management but can also be configured by editing configuration files that are normally XML-formatted. Tomcat includes its own internal HTTP server. Since version 6, Tomcat supports a NIO HTTP Connector and has native Comet support.

4.6 Conclusions

The Comet architecture allows to develop web applications based on server sent events. Because of the nature of the HTTP specification the only way to really have near real time event propagation from client to server is keeping an open connection. We've seen this introduces serious scalability problems on servers but they can be and are being adressed using new models for processing requests based on non blocking I/O systems.

There are currently production ready servers with support for asynchronous request processing. There are multiple libraries supporting Comet models and even a standard protocol (Bayeux) with some support behind it. So it's safe to say Comet is ready for production and actually it's being actually used in several public web applications.

The Comet architecture represents another step into the evolution of web application like AJAX has been on the last two years. In the following years we will see a proliferation of AJAX and Comet enabled web applications that will implement funcionality only available to desktop applications today.

4.7 Future Trends

The gap between desktop applications and web applications is getting small. Not also because there are the technical mechanism available but also because people has started to think about web applications and browser as the ultimate application framework. Is not unlikely a future were most of the applications are web based and built upon web standards [23]. Probably not the ones the current ones but an evolution. That road would bring several challenges which will need to be addressed.

In the middle of the nineties there was a boom coming from the hardware and software major vendors about the thin clients, net clients or NetPCs. It was a vision of things to come, but as many vision it was too much ahead of time. Nowadays we can start talking about the WebOS again, and think of the true mobility where you will have all your desktop computing environment anywere there's a Internet connection. Most of us have already a web based email system which

we can read from anywhere in the world (who hasn't read email on holiday on a very far and remote computer?). Google is one of the pioneering companies behind products like GMail, Google Calendar and Google Docs. Today you can have on the web the email, a calendar, a word processor, a spreadsheet, an instant messenger, a music player, a company files repository... all of the applications most company computers execute at the end of the day. Mobility is a demanded requirement today as sales for laptop systems exced desktop systems. The next step could be simplifying the laptops, making it smaller, more durable, more usable and rely on the network for bringing the applications.

This of course is still years ahead but there's an important wind of change on the industry and companies like Microsoft and Apple who mostly rely on selling an operating system should start thinking in other terms. The software bussines is also changing, and subscription models are starting to become interesting on a world where someone cares about the software, updates, storage of data, etc. Will the WindowsOS be hosted on Microsoft server and billed for usage or monthly rates?

What is clear is that the revolution is starting at the web application level and the Comet architecture is just a single step on that direction.

4.8 References / Further Reading

We're listing some references with some examples and further readings work which could be useful to complement this chapter. On [24] AJAX is applied at the middleware level. Mesbah and Deurse [25] define an architectural style for a single page AJAX model while Khare and Taylor [26] propose an extension to the REST architectural style for decentralized systems. Jacobi and Fallows [27] explore on a single article the Comet architecture and Bayeux protocol.

Bibliography

[1] Jesse James Garrett. Ajax: A new approach to web applications, 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. *Internet RFCs*, 1999. http://tools.ietf.org/html/rfc2616.

[3] E.L. Specification. Standard ecma-262. *ECMA Standardizing Information and Communication Systems*, 3, 1999.

[4] A. Le Hors, P. Le Hegaret, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (dom) level 2 core specification version 1.0. *W3C Recommendation*, 13, 2000.

[5] Alex Russell. Comet: low latency data for the browser, 2006. http://alex.dojotoolkit.org/?p=545.

[6] Rohit Khare. Beyond ajax: Accelerating web applications with real-time event notification, 8 2005. http://www.knownow.com/products/docs/whitepapers/KN-Beyond-AJAX.pdf.

[7] Wikipedia page for comet. http://en.wikipedia.org/wiki/Comet_%28programming%29.

[8] Greg Wilkins. Jetty 6.0 continuations - ajax ready!, 2005. http://web.archive.org/web/20060425031613/http://www.mortbay.com/MB/log/gregw/?permalink=Jetty6Continuations.html.

[9] Jean-Francois Arcand. New adventures in comet: polling, long polling or http streaming with ajax. which one to choose?, 2007. http://weblogs.java.net/blog/jfarcand/archive/2007/05/new_adventures.html.

[10] Giuseppe Naccarato. Introducing nonblocking sockets, 2002. http://www.onjava.com/pub/a/onjava/2002/09/04/nio.html.

[11] Nuno Santos. Building highly scalable servers with java nio, 2004. http://www.onjava.com/pub/a/onjava/2004/09/01/nio.html.

[12] Greg Wilkins Alex Russel, David Davis and Mark Nesbitt. Bayeux: A json protocol for publish/subscribe event delivery, 2007. http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html.

[13] Web apps 1 / html 5, 2007. Server sent events specification http://www.whatwg.org/specs/web-apps/current-work/#server-sent-events.

[14] Jean-Francois Arcand. Grizzly part iii: Asynchronous request processing (arp), 2006. http://weblogs.java.net/blog/jfarcand/archive/2006/02/grizzly_part_ii.html.

[15] Jean-Francois Arcand. Grizzly: An http listener using java technology nio, 2005. http://weblogs.java.net/blog/jfarcand/archive/2005/06/grizzly_an_http.html.

[16] Http streaming. http://ajaxpatterns.org/HTTP_Streaming.

[17] Youri op't Roodt. The effect of ajax on performance and usability in web environments, 8 2006. http://homepages.cwi.nl/~paulk/thesesMasterSoftwareEngineering/2006/YouriOpTRoodt.pdf.

[18] Jean-Francois Arcand. Can a grizzly run faster than a coyote?, 2006. http://weblogs.java.net/blog/jfarcand/archive/2006/03/can_a_grizzly_r.html.

[19] Michael Mahemoff. 210 ajax frameworks and counting. *ajaxian.com*, 2007. http://ajaxian.com/archives/210-ajax-frameworks-and-counting.

[20] Just van den Broecke. Pushlets - whitepaper, 8 2002. `http://www.pushlets.com/doc/whitepaper-all.html`.

[21] Dojo toolkit. `http://en.wikipedia.org/wiki/Dojo_Toolkit`.

[22] Alex Russell. Cometd, bayeux, and why they're different, 2006. `http://alex.dojotoolkit.org/?p=573`.

[23] Aaron Weiss. Webos: say goodbye to desktop applications, networker 9, 4 (dec. 2005). *netWorker*, 9(4):18–26, 2005.

[24] John Stamey and Trent Richardson. Middleware development with ajax. *J. Comput. Small Coll.*, 22(2):281–287, 2006.

[25] Ali Mesbah and Arie van Deursen. An architectural style for ajax. *wicsa*, 0:9, 2007.

[26] R. Khare and RN Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 428–437, 2004.

[27] Jonas Jacobi and John Fallows. Enterprise comet: Awaken the grizzly!, 2006. `http://java.sys-con.com/read/327914_1.htm`.