The Impact of AJAX on the Network

Overview In the beginning there was HTML, and it was good. Users filled out a form and hit the "submit" button, then waited for the server to respond. As applications increasingly became webified, this form-based submission model served us well and the request-reply model remained the de facto standard for building web-based applications.

But users complained about application responsiveness, or the lack thereof, and desired more interactive applications with less wait time between form submissions. Web acceleration became a key component of architecting an application infrastructure capable of scaling and performing to meet user expectations.

Challenge But you know users; they still weren't satisfied. They missed the interactive responsiveness of their fat clients…and while they liked web-based applications, they just weren't robust enough to keep users happy for long. Enter Web 2.0 and, specifically, Asynchronous JavaScript and XML (AJAX).

AJAX allows developers to build applications with rich interfaces that act more like their fat-client ancestors than their web-based cousins. Applications built on AJAX are often also referred to as Rich Internet Applications (RIA), those built upon Adobe's Flash platform are referred to as AFLAX (Asynchronous Flash and XML) or just FLAX (Flash and XML). What they all have in common is their ability to communicate with the server asynchronously and a set of fat-client client "widgets" (components, gadgets) that spice up the user-interface and make users happy. Not only are these interfaces sexier, but they're also capable of functioning like a fat-client—they can communicate with the server at will, with or without a specific user action initiating the exchange, and they can update objects on the page without requiring the entire page to be refreshed from the server.

Solution Basically, AJAX moves web-based applications from a page model to a true application model, based on events and user actions rather than pages.

In the quest to deliver fat-client functionality on a thin-client platform (i.e., the browser), many developers have turned to using toolkits. These toolkits (Dojo, Zimbra, Laszlo, Google) are JavaScript libraries that present an API and/or a set of markup tags that make it easy to develop rich interfaces and communicate with the server over HTTP. Unfortunately AJAX and similar technologies introduce significant delivery challenges that are not addressed by these toolkits or by the technology itself. Also not addressed, as is typical, are issues surrounding basic security and access rights to resources used by AJAX applications.

Why AJAX?

Before we jump into the technology, it's important to first understand why people are all agog over AJAX. AJAX toolkits generally comprise two components: a set of DHTML widgets and a communication library. Core AJAX functionality is focused on the communication channel, which provides the means to make HTTP requests to back-end servers from within JavaScript. Toolkits combine the ability to rapidly develop slick interfaces with these communication libraries. This makes it possible for the developer to build rich, interactive user interfaces that are browser agnostic. Well, they're supposed to be browser agnostic—experience says that many applications built on AJAX toolkits still break when used in Internet Explorer or Firefox, depending on the developer's biases.

Despite the fact that it's cool and sometimes developers jump on a technology bandwagon for the sake of doing something new and different, there are real benefits that can be achieved through the use of AJAX and similar technologies.

The primary benefits of using AJAX-based technologies are:

- Provide a common set of standard, DHTML-based "widgets" to improve the user-interface's look and feel.

  1. Tab components
  2. Editable combo boxes
  3. Rich text editing components

- Reduce the refresh rate of the "page" by providing a mechanism for developers to dynamically update objects based on a variety of variables.

  1. Keep the page updated with the number of "guests" and "members" online
  2. Load customer names into a combo box after the user chooses a state or province to limit the amount of data transferred
  3. Parameterized configuration of options for products; only show the colors, size, ports, modules, etc. applicable to the selected product—in real-time.

- Pre-fetching data to increase client responsiveness.

  1. Google Maps
  2. Images and data to be displayed in tabs or tree nodes

An example of a real-time, dynamic AJAX-based application is WebBoggle. If you play a game you'll notice how different pieces of the page update without the page refreshing. Every time you enter a word the data is sent to the server, checked against a dictionary, scored and then returned to the browser where it is parsed and entered into the list of words you've found, along with your new score—all without the page being reloaded. That's AJAX at work. It's great for the users, but they can't see what it's doing to the network and the server upon which the application is running. The server running this game crashes, and crashes often, and connections being attempted "under the covers" often time out, leaving the player disconnected with no explanation as to why.

The enabler of all this AJAX technology is the XMLHTTPRequest object. It's an object that Microsoft implemented in Internet Explorer back in 1998 and is probably one of the few "add-ons" that Microsoft put into its browser that other browsers picked up and adopted fairly quickly. The latest working draft of the W3C specification describes the XMLHTTPRequest object as an API that provides additional HTTP client functionality for transferring data between a client and a server. This specification aims to standardize the object in order to remove issues of browser interoperability currently caused by differences in implementation.

HTTP is emphasized throughout this discussion because that's really important to understand. It's just HTTP and this is a well-understood and implemented protocol. Nearly everything available in HTTP is available through the XMLHTTPRequest object. Authentication/authorization, cookies, and cache-control are among the functions available through the XMLHTTPRequest object that should be recognizable to those familiar with HTTP.

While the underlying transport may be HTTP, and is therefore well-understood, the way in which AJAX-based applications exercise HTTP is very different than that found in traditional web-based applications.

AJAX Differences and Impact on Applications

AJAX stands for Asynchronous JavaScript and XML. Each of these composite pieces has an impact on the network, for varying reasons.

Asynchronous

The sender (client) and receiver (server) aren't taking turns anymore. While traditional browser-based applications, primarily enabled through form submission, follow the request-reply paradigm just like its underlying transport (HTTP), AJAX-based applications do not. Both the browser and the client can be doing things independent of one another. While the user is interacting with the UI, the browser can be pre-fetching images and validating data at the same time.

> Impact on the client: More requests. A good example of this is Google Maps. While you're looking at the first snippet of the map presented, the client is quietly retrieving other large chunks of image data under the covers. This is what makes panning and zooming around in Google Maps appear so seamless—the data was loaded behind your back, whether you wanted it or not.
>
> Impact on the communication path: More data. Sure, it's smaller data in general, but there's actually more—it's just spread out across many more requests, which increases the overall traffic on the wire.
>
> Impact on the server: More responses. The server has to reply to the additional requests and serve up the additional data whether the user will view it or not. There are also connection issues, as real-time functionality requires a nearly constant stream of data flowing between the client and the server, using up resources on the server.

JavaScript

Code is executed on the client as well as the server. The browser is now a platform, just like the server, and is responsible for more than just rendering of markup languages. It's responsible for managing connections, interpreting data, and executing code.

> Impact on the client: Code executed blindly on the client raises myriad security concerns, especially if the code being executed is transported in clear-text from the server.

Data is not cached by the XMLHTTPObject as traditional web objects, despite the inclusion of cache-control headers in the latest W3C specification. Data retrieved once may be requested again in the same session, regardless of whether it has changed or not. This is an extension of the current "all content is dynamic whether it truly is or not" problem.

JavaScript code is easily readable on the client, making it a simple task to discover the functions and associated URLs necessary to craft a number of malicious attacks against the server.

> Impact on the communication path: Additional JavaScript code must be included in the initial load of the AJAX application to handle communications, meaning the size of the first page will increase as well, resulting in more data on the wire. Implementations that use JSON (JavaScript Object Notation) instead of XML to exchange data will not substantially affect the amount of data exchanged dramatically, as both formats are verbose.
>
> Impact on the server: The potential exists to manipulate the JavaScript functions providing AJAX capabilities or data being exchanged to launch malicious attacks against the server,

potentially causing service disruption on the server. A DoS (Denial of Service) attack can easily be crafted using the JavaScript functions on the client, sometimes unintentionally.

XML

The XML data transferred between the client and the server is generally smaller with AJAX than a full HTTP form, but it isn't always XML. Sometimes it's XHTML, other times it's just text-based such as comma-separated data, and sometimes it's just plain old text that only a developer would understand.

Impact on the client: Parsing of any data, especially XML, is compute-intensive, as is the process of updated existing objects in the page with new data. JavaScript is truly an interpreted language, and relies heavily on the capabilities of its host (the browser) to be performant. Client performance may be adversely impacted in general, but this is especially true if the data is complex XML.

Impact on the communication path: The amount of data may be increased overall. While AJAX strives to reduce the amount of data transferred in a single request, the reality is that many applications are making additional requests and loading data the user may never need which increases the overall amount of data on the wire. And while the request and responses strive to be small, if they are XML then they are always almost 2 times the size they need to be.

x-www-url-encoded (traditional HTTP GET/POST)	name=fred
XML	<name>fred</name>

The x-www-url-encoded field requires exactly 9 bytes of data, but in XML the same request requires at a minimum 17 bytes, nearly 2 times the data. For small requests the impact is negligible, but for larger data sets the increase in size begins to have a negative impact on the communication path (network).

Every intermediary in the network path adds latency to the transaction. Remember that XML is human-readable and that it must be converted into something an application or device can understand before it can be processed. That means that every application or device that needs to handle XML before it reaches the back-end server will need to parse the XML—and to parse the XML it has to be in memory. That means a lot of memory and a lot of CPU resources dedicated just to parsing XML before any processing actually happens. So it's far better for one intermediary to parse the XML and perform as much processing as possible—such as security, authentication, and validation of data – before sending it on to the back-end server. Not only does this simplify the architecture of a SOA (Service Oriented Architecture) and WOA (Web Oriented Architecture) implementations, but it can increase the overall responsiveness of the entire infrastructure by reducing the number of times the XML must be parsed and only requiring back-end servers to parse and process valid requests.

Impact on the server: The potential exists to manipulate the JavaScript functions providing AJAX capabilities to launch malicious attacks, potentially causing service disruption on the server. If the application actually is using XML then parsing will be required by the server-side application, consuming resources and increasing latency.

The biggest difference between AJAX-based applications and traditional web applications is that HTTP requests can be triggered by user-actions and not constrained by traditional web-based transactional boundaries, such as form submission. User-actions such as the input of a single key can trigger behind-the scenes HTTP requests. If you were to visit one of Google Labs' tools, you'd notice that as you type each letter, Google "suggests" possible completion values, including the

number of matching pages in its extensive index. Each keystroke in this application generates a new HTTP request to Google's auto-completion server and requires JavaScript execution on the client to read, parse, and modify the page to display the data. That's a lot of requests and a lot of processing in a very short period of time.

What is the impact on my infrastructure?

Understanding what AJAX is and how it impacts the browser, network, and servers leads to the quest for a solution. Each of the negative impacts on the overall infrastructure and delivery channels for AJAX applications has a corresponding, existing solution today, with almost all of the solutions being easily implemented by the addition of an application delivery controller to the existing infrastructure.

Verbose data

Data being exchanged is often fatter than its form-encoded cousins. Compression can be used to reduce the size of data being transferred to optimize the communication path. Caching can provide relief for larger “first pages”, which must include all the necessary JavaScript to enable the rich interfaces and underlying communication.

Dynamic data isn’t

Much of the data being pre-loaded under the covers is static text and images. Just as many objects in traditional web-based applications are treated as dynamic when they are not (the online bank logo), so the same is true for AJAX-based applications. Intelligent caching can reduce the burden on servers that are already seeing an increased burden due to increased request rates. Some of the data being loaded in an application is actually dynamic (retrieved at real-time from a database query) but has already been requested by the client earlier in the session. Again, intelligent caching can alleviate many of the issues arising from developers and servers that treat rarely changing dynamic data as truly dynamic content.

Asynchronous requests

At first glance this doesn’t appear to be problematic, but AJAX-based applications using a single connection “under the covers” to pre-fetch data or perform validation functions can cause requests to queue up on the server while the client waits for a response. The ability to distribute those requests to multiple servers (in situations where persistence is not required) and respond more quickly keeps the request queues from filling up and overburdening the server, and also clears connections as soon as possible to allow servers to serve more efficiently.

Higher frequency of requests / the Polling Paradigm

AJAX encourages real-time communication between the client and the server. This means a higher frequency rate of requests, which can quickly bury a server and begin impeding performance. This high arrival rate of requests can also mask more nefarious behavior such as a denial of service attack. Traffic management is necessary to prevent both unintentional and intentional denial of service situations caused by high request rates. TCP optimization, connection management, and security should be high priorities for deployments of AJAX-based applications to reduce the noise on the wire and lower the bandwidth necessary to support a high volume of users.

Because AJAX encourages real-time communication, developers often use AJAX to automatically update information on the page using a polling mechanism similar to that of enterprise portal

systems. A function sits quietly and executes a request to see if "anything is new" on a specified interval. This polling functionality is often implemented as a separate connection, which can be become problematic for browsers that restrict the number of connections to the server. If connections fail, application functionality breaks or becomes so sluggish that the productivity benefits thought to be achieved by an AJAX-based interface are completely lost.

This demo (http://www.unwieldy.net/ajaxim/) of an AJAX-based IM client is a good example of a polling AJAX application at work. A sniffer trace of this AJAX application shows it is polling the server approximately every 2.5 seconds, sending 63 bytes of application data to the server and receiving 421 bytes of data—that's without any communication actually occurring between users. That's approximately 11KB per minute per client being exchanged. It's easy to see that as the number of users increases (which would increase the 421 bytes a set amount per user) that the amount of traffic being exchanged will increase—just to keep the clients updated as to who is available.

Not only is the amount of base traffic problematic, but the connection to the server must also be kept open lest performance degrade due to TCP session management, reducing the available connections on the server for other applications and clients.

Load balancing, connection management, compression, security, and TCP optimization all provide relief from the increasing burdens placed on the client, network, and server by AJAX-based applications. Using an application delivery controller to provide load-balancing, connection management, and TCP optimization, HTTP compression and acceleration, as well addressing security concerns can reduce the negative impacts of deploying these types of applications.

Longer TCP time-out needed

Because requests are more frequent, one of the tuning options used by developers is to increase the TCP time-out values on their web-server. This reduces the amount of time the server spends dealing with TCP session setup/tear down, but as a result reduces the number of users the server can handle because each client can require a nearly continuous connection to the server during the client session, chewing up valuable server resources. And of course there exists the possibility that the increased time-out value will result in connections sitting idle, wasting resources that could be used to serve other clients. Connection and traffic management that provides optimization of TCP and is capable of handling large volumes of connections is necessary to ensuring scalability of the servers providing AJAX-based back-end functionality.

Lack of security considerations

AJAX and SOA share common traits in that both are very function (operation) oriented. Many developers will implement one script on the server per function, which means many more opportunities for security vulnerabilities to be introduced. There is often a one-to-one mapping between AJAX controls on a page and a script deployed on the back-end server. These scripts can also often be directly invoked, with the URI easily extracted from the very human-readable JavaScript on the client. Direct invocation circumvents limited and rarely implemented client-side security.

Many of the functions called upon by AJAX-based applications will directly interact with the database, giving rise to the opportunity for malicious minded – or just sometimes curious – folk to tamper with data being transferred, resulting in a wide variety of application-level attacks. These attacks are no different than those found in traditional page-based web applications.

As the underlying requests generally invoke server-side scripts via traditional HTTP mechanisms, an application firewall can provide nearly immediate relief for many security issues, especially those involve cookies, URI-based attacks, and command-injection attacks hidden in the payload.

False sense of security

Because many AJAX developers use AJAX and JavaScript to perform data validation on the client, they ignore the need to revalidate the data on the server side, which presents an opportunity for malicious attacks to be introduced despite the client-side verification. Developers are lulled into a false sense of security by client-side validation and are lax about security on the server-side, where it matters most.

AJAX developers also fail to implement security in every script accessed by their AJAX-based applications. While authentication may be required for the application, AJAX-based components rarely take advantage of even the limited HTTP Basic-AUTH capabilities of the XMLHTTPObject nor do their myriad scripts supporting the application reaffirm the client's access rights to the script and to the data being returned by the script. And because the URI and parameters required for these scripts can be easily gleaned by reading the JavaScript it is a simple task for an attacker to manipulate and exploit the scripts.

Because an application firewall understands this model and can protect against attacks like parameter and cookie tampering, it can prevent attempts to exploit the back-end scripts serving AJAX-based applications. The centralized client authentication capabilities of an application delivery controller can assist in further securing access to scripts by bridging authentication systems and ensuring that only authorized applications and/or users are accessing these scripts, or by enforcing authentication in general to prevent unauthorized access. These products can also play a part in securing AJAX scripts by scrubbing both the request and response, and by hiding server errors that may contain stack traces or other language-specific error messages that might provide insight into internal infrastructure and architecture.

Conclusion

AJAX is built upon existing technologies. Though it has a unique impact on the client, server, and the network due to its use, these challenges are easily addressed by existing, proven application delivery methods that address similar challenges arising from other technologies. An application delivery controller combined with an application firewall address the network, server, and security issues that will certainly arise with increased deployment of these new web-oriented rich applications.

About F5

F5 Networks is the global leader in Application Delivery Networking. F5 provides solutions that make applications secure, fast and available for everyone, helping organizations get the most out of their investment. By adding intelligence and manageability into the network to offload applications, F5 optimizes applications and allows them to work faster and consume fewer resources. F5's extensible architecture intelligently integrates application optimization, protects the application and the network, and delivers application reliability—all on one universal platform. Over 10,000 organizations and service providers worldwide trust F5 to keep their applications running. The company is headquartered in Seattle, Washington with offices worldwide. For more information, go to www.f5.com.