Push solutions for AJAX technology

Master's thesis

Research report

Engin Bozdag

Push solutions for AJAX technology

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Engin Bozdag
born in Malatya,Turkey

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Backbase BV
Stephensonstraat 19
Amsterdam, the Netherlands
www.backbase.com

Push solutions for AJAX technology

Author: Engin Bozdag
Student id: 1119869
Email: v.e.bozdag@student.tudelft.nl

Abstract

A new breed of web application, dubbed AJAX, is emerging in response to a limited degree of interactivity in large-grain stateless Web interactions. However, AJAX still suffers from the limitations of the Web's request/response architecture. This prevents servers from "pushing" real-time alerts such as market data, news headlines or auction updates. It is possible to overcome this limitation by using several techniques. This paper first summarizes two different types of architectures to accomplish a push solution. It later discusses several solutions and examples from the software industry and evaluates these solutions from the perspective of several software quality attributes.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	ir. Ali Mesbah, Faculty EEMCS, TU Delft
Company supervisor:	Mark Schiefelbein, Backbase

Contents

List of Figures

Chapter 1

Introduction

Recently, there has been a shift in the direction of web development. A new breed of web application, dubbed AJAX (Asynchronous JavaScript and XML) is emerging in response to the limited degree of interactivity in large-grain stateless Web interactions. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes and making changes to individual user interface components. This way, the entire web page does not have to be reloaded each time the user makes a change. This *Single Page Interface* (SPI) approach makes AJAX a serious option not only for newly developed applications, but also for existing web sites if their user friendliness is inadequate [48].

The term AJAX spread rapidly from a Weblog to the Wall Street Journal within weeks. The new web applications under the AJAX banner have redefined end users' expectations of what is possible within a Web browser. However so-called 'fat clients' that run on PC's retain one fundamental advantage over the Web: Real-time event notification. AJAX alone does not address instant messengers, stock tickers, or other collaborative applications that require 'push' data streaming [38].

The web model, for security reasons, requires all browser/web server communication to be initiated by the client; i.e., the end user clicks on a button or link and thereby requests a new page from the server. There are, however, many use cases where it is important to update the client-side interface in response to server-side changes. For example:

- An auction web site, where the users need to be averted that another bidder has made a higher bid.
- A stock ticker, where stock prices are updated.
- A chat application, where new sent messages are delivered to all the subscribers.
- A news portal, where news are pushed to the subscriber's browser when they are published.

Today, these types of applications are usually implemented using a client-pull style, and the client component actively requests the state changes using client-side timeouts. These solutions have many drawbacks; i.e., if the clients pull too often, this might lead to high server load. If the updates are not frequent, clients will make unnecessary requests. On the other hand, if the clients pull infrequently, they might miss

some updates. An alternative to this is push-based style, where the server broadcasts the state changes to the clients asynchronously every time its state changes. Most AJAX technology implementations use the pull-based style. AJAX applications are designed to have a high user interactivity and a low user-perceived latency. Using a push-based style can further improve these properties.

This research explores the fundamental limits of browser-based applications and explains how real-time event notification can be added to today's Web and AJAX technology. Chapter 2 shows the focus and the goal of this research. Chapter 3 introduces an architecture for the AJAX technology. Chapter 4 discusses push and event-based models from the literature. Chapter 5 compares these models with the implementations from the software industry. Chapter 6 introduces current web based push solutions that are used in the software industry. Chapter 7 illustrates several examples that use these solutions. Chapter 8 evaluates the current solutions based on several software quality attributes. Chapter 9 ends this paper with a conclusion.

Chapter 2

Problem Statement

As we stated in the introduction, most AJAX applications do not implement a server-initiated data transfer model, which is also known as ”push”. Providing this push functionality is very important, because the client-initiated model is very inefficient in situations where frequent updates are needed and it might lead to network congestions. However implementing such a ”push” solution is not trivial. Most studies that discuss the push approach [32, 1, 43, 30, 68, 53, 37] focus on the streaming of large data (such as media) and introduce a broadcasting model. However, most AJAX based push solutions rather deal with frequent events with small amount of data and use an event-based model since they have to deal with the limitations of HTTP. As we will see in Chapter 6, although HTTP/1.1 [24] brought improvements over HTTP/1.0 [13] such as persistent connections and pipelining, it still has limitations on a full server-initiated data transfer model.

Backbase is an Amsterdam-based company that provided one of the first commercial AJAX frameworks [11]. The framework is still in continuous development, and in use by numerous customers world wide. The addition of a push implementation will improve the current Backbase framework. There are already AJAX solutions available that use the push technology and the demand for such a solution is high. Backbase would like to keep up with the changes that are already happening in the market and the technology.

In this paper we compare the push-based models from the literature and the web-based push solutions used in the software industry. We discuss the limitations of the HTTP protocol and compare the HTTP based solutions with the alternatives. We also evaluate push model based on software quality attributes taken from [48, 12]. Based on the results of the research, a suitable solution will be implemented for the Backbase AJAX framework.

Chapter 3

AJAX Architecture

AJAX is a combination of several web application development technologies, previously known as Dynamic HTML (DHTML) and remote scripting, to provide a more interactive web-based user interface [48]. It is composed of the following technologies [31] :

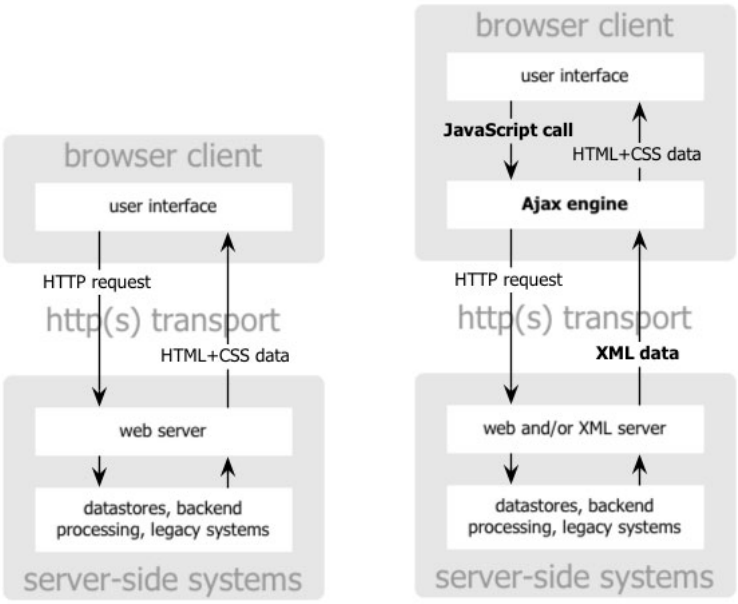
- Standards-based presentation using XHTML and CSS;
- Dynamic display and interaction using the Document Object Model (DOM);
- Data interchange and manipulation using XML and XSLT;
- Asynchronous data retrieval using XMLHttpRequest;
- JavaScript to bind everything together.

In the following subsection we compare AJAX approach with the classical web application model. Later we mention SPIAR [49], an architectural style for AJAX applications.

3.1 Classical web vs. AJAX

In the classical web application model (Figure 3.1a), the user performs an action that causes the browser to make an HTTP request to the web server. After receiving the request, the server does some processing (i.e., retrieving data, doing some calculations, connecting to various legacy systems) and then returns an HTML page to the client. However between making the request and receiving the response (Figure 3.2a), the user has to wait and a new page is needed to be reloaded in every request. This *Multi Page Interface* (MPI) approach has generally exhibited problems such as slow performance and limited interactivity, particularly when compared to typical desktop applications [51].

AJAX [31] tries to solve this problem by introducing an intermediary AJAX engine between the user and the server (Figure 3.1b). The engine is loaded at the start of the session and it handles the events initiated by the user, communicates with the server, and has the ability to perform client-side processing [48]. The user actions now do not generate an HTTP request to the server, but takes the form of a JavaScript to the AJAX



(a) Classical Web Application Model

(b) AJAX Web Application Model

Figure 3.1: The traditional model for web applications compared to the AJAX approach. Images are taken from [31].

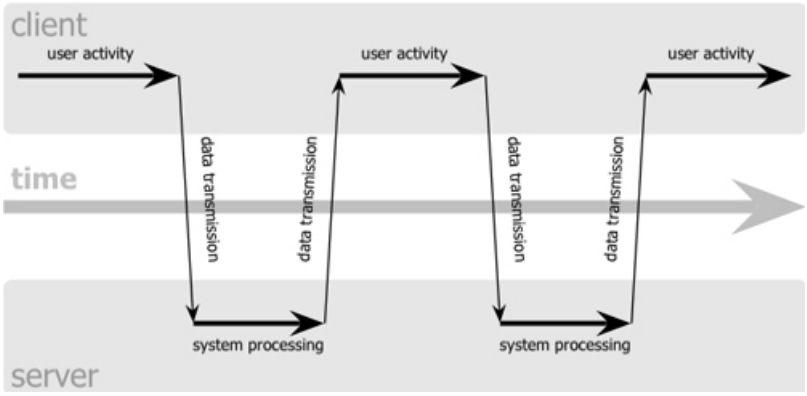
engine instead (Figure 3.2b). If the user action does not need data from the server (i.e., simple data validation, editing data in memory, and even some navigation), the engine handles it on its own. If the engine needs something from the server in order to respond, it makes those requests asynchronously, without stalling the user's interaction with the application. The user stays in the same page, and the page does not need to be reloaded for every request.

This *Single Page Interface* approach has been used in the software industry, even before the term AJAX was coined. It is widely used now by web applications such as Google Map, Gtalk, Flickr, and the new versions of Yahoo Mail and Hotmail.

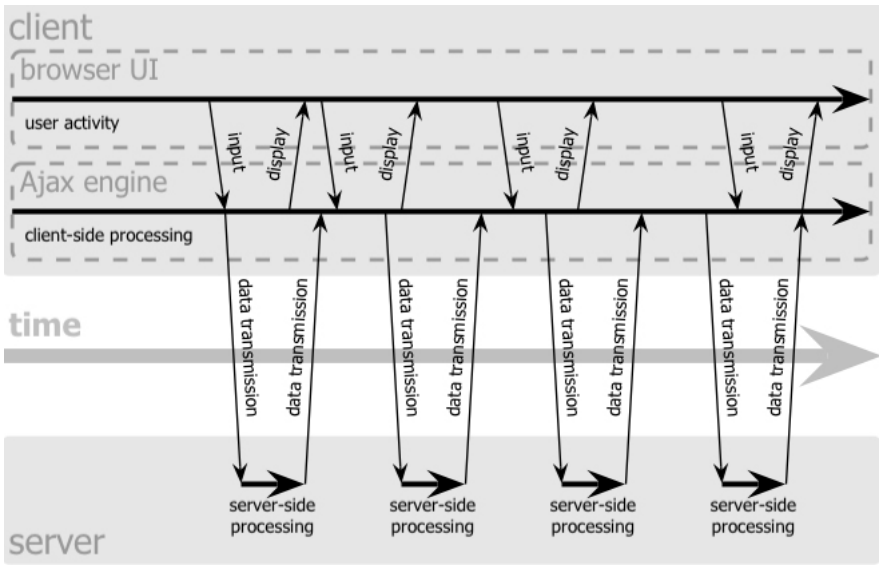
3.2 An Architectural Style for AJAX

In the interest of scalability, the world-wide web adopts a *stateless* approach to client-server communication. This style of application is called REpresentational State Transfer (REST, [25]). In this style, clients request resources from servers (or proxy servers) using the resource's name and location, specified as a Uniform Resource Locator (URL,[14]). All interactions for obtaining a resource's representation are performed by synchronous request-response messages over an open, standard network protocol (HTTP, [23]). In this scheme, each interaction between the client and the server is independent of the other interactions. No "permanent" connection is established between the client and the server and the server maintains no state information about the clients.

AJAX architectures on the other hand are not so easily captured in REST, due to



(a) Classical Web Application Model



(b) AJAX Web Application Model

Figure 3.2: The sequence of events in different web application models. Images are taken from [31].

the following reasons [48]:

- REST is suited for the transfer of large data, but AJAX works with small data exchanges.
- In REST, a client requests a specific resource. In AJAX, a response is required to a specific action.
- In REST, all interactions in order to obtain a resource are performed in a synchronous request-response scheme. With this scheme the user has to wait until a response is returned from the server. AJAX applications, however, require a model for asynchronous communication.
- No permanent connections are established in REST, so the server has to be stateless. This increases scalability, but the trade-offs with respect to network perfor-

mance and user interactivity are of greater importance in AJAX.

In order to deal with these limitations of REST, SPIAR [49, 48] identifies the following architectural elements and shows their interaction in a processing view, where data flow and communication can be observed (See Figure 3.3).

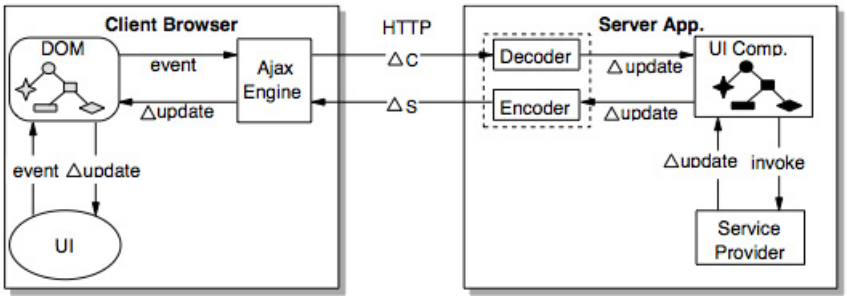


Figure 3.3: Processing View of a SPIAR-based architecture. Taken from [48]

- Processing Elements: These are the components that supply the transformation on the data elements. They are formed by the Ajax Engine and User Interface (UI) widgets on the client side, Delta Encoder/Decoder, UI Components and Service Provider on the server side.

- Data Elements: They contain the information that is used and transformed by the processing elements. They are formed by DOM on the client side and Delta communication messages, which are exchanges between the client and the server.

- Connecting Elements: They hold the components together and enabling them to communicate. Events, which are initiated by a user actions, Delta connectors which are light-weight communication media connecting the engine and the server, and Delta updates, which reflect state changes all belong to connecting elements.

According to Figure 3.3, user activity on the user interface fires off an event which is delegated to the AJAX engine. If a listener on the server side has registered itself with the event, the engine sends a DELTA-CLIENT messages of the current state changes and send it to the server, if needed, asynchronously. On the server, the message is decoded, and relevant components are identified in the component tree. The changed components invoke the event listeners of the service provider. The service provider will handle the action, update the corresponding components with the new state, which will be rendered by the encoder. a DELTA-SERVER message is sent back to AJAX engine on the client side, and the engine will update the DOM and the interface. If no request to the server is needed, the engine may also update the DOM directly.

The asynchronous interaction and client side processing of AJAX improve user interactivity and user-perceived latency. Delta-communication style only interchanges state changes, instead of full-page retrievals, reducing data redundancy. UI components improve simplicity because developers can use existing components.

Chapter 4

The Push Technology

As stated in Section 3.2, REST has many limitations for representing AJAX architectures. REST also poses the following limitations for server-initiated events [39, 32]:

- Every request must be initiated by a client, and every response must be generated immediately, precluding servers from sending asynchronous notifications. REST cannot model peer-to-peer relationships well.
- It is difficult to maintain a state: the client, the server, or both must maintain the state and ensure its coherence
- Every request can only generate a single response; and if that response is an error message (or lost), there are no normative rules for proceeding.
- Every request proceeds from one client to one server. Without a way to transfer information to a group of components, this leads to nested ”proxy chains”.

The alternative to this client-server model is peer-to-peer communication. This model may be divided further into two subclasses: the push-based and the event-based models [32].

4.1 Push-based model

4.1.1 Push by Continuous Pulling

In order to ”simulate” a server push, most of the available push systems actually pull at the dissemination infrastructure level; i.e., at regular, user-definable, intervals the receiver checks with the broadcaster whether the receiver's view of the channel is still consistent or needs to be updated. However, with such a scheme complete data accuracy and data freshness cannot be achieved. In order to achieve this, the frequencies of pulling should be high, which in turn will lead to high network traffic and high number of unnecessary messages if the channel data does not change frequently. According to the research of Acharya et al. [1], If the server is lightly loaded, client-pull seems to be the best strategy. In this case, all requests get queued and are serviced much faster than the average latency of broadcasting. However if the server is saturated, the probability of a request being dropped is extremely high. This makes full client-pull scheme very ineffective in such conditions.

4.1.2 Pure Push Approach

The communication model for the push-based systems is tightly coupled and asymmetric. In contrast to event-based model, push-based systems scale to fewer producers, but more customers. They may be viewed as a specialization of the event-based systems with designated producers and customers, and channels to connect each producer with interested consumers. Dissemination is done on the basis of particular channels rather than event classes. Hauswirth and Jazayeri [32] analyzed existing push systems and identified the following components (See also Figure 4.1):

- Information source
- Broadcaster
- Channel
- Repeater
- Receiver

In this model, a consumer (receiver) subscribes to a channel and receives any information that is sent on the channel. A producer (information source) sends the data to the broadcaster with rules on how and where (which channel) to distribute the data. The broadcaster may apply filters to the date and then disseminate it via channels to a set of repeaters (Repeaters are needed to provide scalability to high number of users). The repeaters will then redistribute the data to the subscribed receivers. For higher scalability, additional levels of repeaters may be necessary. The transportation system is transparent to the broadcasters, receivers and channel. Receivers think that they receive the data directly from the source, and vice versa.

For large-scale applications that provide channels to thousands of subscribers , the broadcaster cannot be a single component. For scalability, a specialized, distributed, broadcasting infrastructure is necessary. In that case data management schemes such as primary copy replication [65] or data partitioning [65] can be used.

The key underlying design goal for any distributed system is scalability [32]. In the push model, the broadcaster actively sends content to its subscribed receivers. This solves the freshness problem, however contacting the subscribers sequentially does not scale even for medium numbers of subscribers. It would leave receivers with different views of channel information depending on their ordinal number in the pushing process [39]. As a consequence of worldwide geographical dispersion, it becomes necessary to address relativistic issues in multiple observations of the same event. Additionally, an application requesting a notification about an event at roughly the same time as, but prior to, the occurrence of the event of interest may or may not be notified about the event [54]. Server push broadcasting also requires a directory of subscribers. This brings additional administration costs, since it must be maintained, kept consistent and is a single point of failure. Additionally, if the receivers are not online all the time, rebroadcasts might be needed, which will effect the load and complexity of the system.

Khare and Taylor [39] propose a push approach called ARRESTED. Their asynchronous extension of REST, called A+REST, allows the server to broadcast notifications of its state changes. In this scheme, clients issue a long-running “watch” request

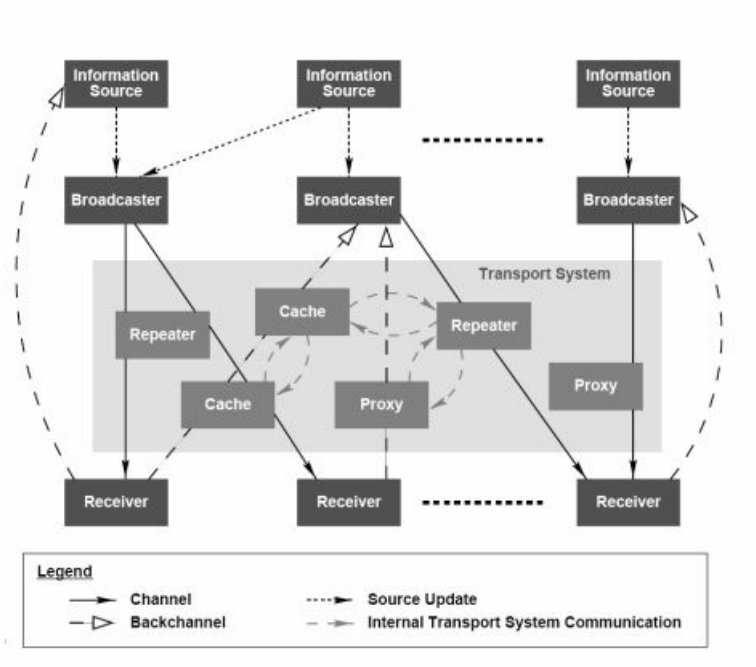


Figure 4.1: Components of the push model. Taken from [32]

rather than a one "get", and receive multiple "notify" replies. Authors note that this is a significant implementation challenge across the public Internet.

4.1.3 Hybrid Approach

Deolasee et al. [15] argues that it is difficult to determine whether a push- or pull-based approach should be employed for a particular data item. They developed two methods in order to deal with this issue. In the first method, which is called *Push and Pull* (PaP), the client actively pulls the data, and the server pushes data when there is a sudden surge in the rate of change that is yet to be detected by the client. The *Push or Pull* (PoP) allows the server to adaptively choose between a push- or pull-based approach on a per-connection basis, depending on observed rate of change of the data item or the coherency requirements. Note that PaP can not be implemented in REST, since the server can not push updates to a client which is pulling.

Similar hybrid approach is also mentioned by Hauswirth et al, and Acharya et al. [1, 32]. Such approaches combine the advantages of server push (freshness, consistency) and client pull (scalability). Consumers are notified about the availability of new data via a push mechanism (small messages) while the client pulls to transfer the actual data (possibly large data). By chopping off the large data, the available bandwidth for pull is increased. However if there is not enough bandwidth dedicated to client-pull, performance can degrade severely since clients will not be able to pull non-broadcast data [1]. Using a threshold value here can help to reserve more of the backchannel capability for pull data. Acharya et al. [1] define a parameter *PullBW*, in order to denote the bandwidth available for pull transfer. Here a 100% *PullBW* denotes

”Pure-Pull”, and a 0% *PullBW* denotes ”Pure-Push”. According to the experimental results, a lower value of Pull (e.g., 30%) does not produce the best results when the system is lightly loaded, but neither does it produce the worst results when the system is heavily loaded, and thus, are suited for systems with dynamic loads [1]. Note that according to Hauswirth and Jazayeri [32], analyzing the scalability of a push system is far from straightforward because it depends on many criteria and many design goals: number of broadcasters, receivers, channels; amount of data on channels; frequency of updates; network latency and bandwidth; and the amount of common subscriptions to certain channels.

IP Multicasting instead of Proxy/Repeater component is also an option, but this is mostly limited to intranets, since not all routers will allow such packets to pass through [27]. Therefore using Internet service provider sites as repeaters seems to be a promising choice. But this, of course, will bring the question of payment for services, which might require further research.

4.2 Event-based model

In this model, nodes are loosely-connected and behave symmetrically: any node may produce events and any node may consume events. Components declare interest in receiving specific events and are notified of occurrence of those events. This approach is targeted at possibly very high event-rates, but since events are happening frequently, the payload sizes must be minimal. This model scales to many producers and many consumers because there is no coupling between them [16]. In an event-based system, component interactions are modeled as asynchronous occurrences of, and responses to, events. To inform other components about the occurrences of internal events (such as state changes), components emit notifications containing information about the events. Upon receiving notifications, other components can react by performing actions that, in turn, may result in the occurrence of other events and the generation of additional notifications.

Welsh et al. [71, 72] define the key property of a well-conditioned service as *graceful degradation*. In graceful degradation, as offered load exceeds capacity, the service maintains high throughput with a linear response-time penalty that impacts all clients equally, or at least predictably according to some service-specific policy. Welsh et al. argue that the traditional web servers such as Apache [29], use thread-based concurrency, in which each accepted request consumes a thread to process it (See Figure 4.2a). Although relatively easy to program, such a model has the following problems (Quality attributes are taken from [57]) :

- Efficiency: Threaded model can not achieve *graceful degradation*, because of the overheads associated with threading. Scheduling overhead, lock contention, cache misses, etc. can lead to serious performance degradation when the number of threads is large [71]. Increasing the thread limit does not solve the problem, but rather postpone it. As we see from Figure 4.4a, the throughput decreases substantially, when the number of threads reach a certain number. The latency also increases, leading to large response times. It is also possible that a small number of threads may cause the bottleneck in the system. Because the system

can not inspect the internal request system, it will arbitrarily reject new connections.

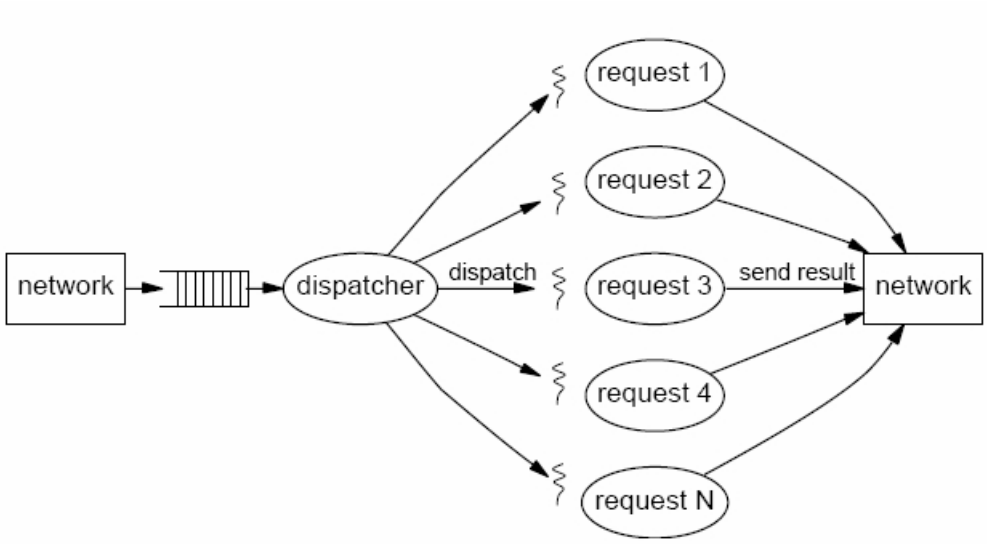
- Programming simplicity: Threading may require complex concurrency control schemes. In the lack of a scalable multi-threading platform, programmers usually have to implement their own thread pool and thread control. Without a standard here, it will be difficult for a mediator to understand the interactions between system components, therefore the system itself will be less visible.

- Portability: Threading is not available on all OS platforms.

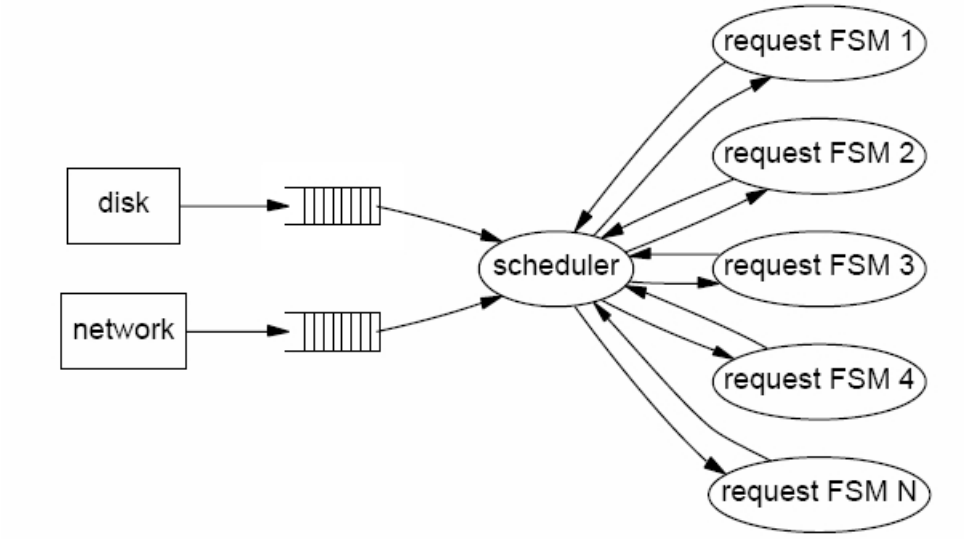
The event-driven design intends to solve these limits of the threaded model. In this approach [71], shown in Figure 4.2b, a small number of threads (usually one per CPU) loops continuously, processing events of different types from a queue. The event-driven approach implements the processing of each task as a finite state machine (FSM), and the transition between the FSM are triggered by events. In this way the server maintains its own continuation state for each task, rather than relying upon a threaded context. Figure 4.4b shows the throughput achieved with an event-driven implementation of the service. As the number of tasks reach a certain limit, the excess tasks are absorbed in the server's event queue. The throughput remains constant and the latency shows a linear increase, which is a huge improvement comparing to threaded design.

An important limitation of an event-based model is that it assumes that event-handling threads do not block. However operating system primitives, such as page faults, garbage collection or interrupts may block those threads. This model also brings additional challenges for the application developer. In order to balance fairness with low response time, the application must carefully multiplex the execution of multiple FSM's [71]. This requires careful decision on scheduling and ordering of events.

Welsh's *Staged Event-Driven Architecture* (SEDA,[71]) is a good example of event-based model. In SEDA, a service is decomposed into a graph of stages, where each stage is an event-driven service component that performs some aspect of request processing (See Figure 4.3). Each stage contains a small, dynamically sized *thread pool* to execute the given task. Stages are connected with explicit queues that act as the execution boundary between stages, as well as a mechanism for controlling the flow of requests through the service. In this model, each stage is responsible only for a subset of request processing, and stages are isolated from each other through composition with queues. Each stage has an *admission controller* that accepts or rejects new request for the stage. Additionally, stages also controlled by a *resource controller*, which might adjust the number of threads executing within each stage, based on an observation of the stage's offered load and performance. SEDA has an overload management that prevents service performance degrading in an uncontrolled fashion under heavy load. This is accomplished by admission control, service degrading, and class based differentiation. Welsh and Culler [72] emphasize that traditional overload protection (used by Apache[29]) just stops accepting new connections when all server threads are busy. This is problematic, since it is based on a static thread or connection limit, which does not directly correspond to user-perceived performance and does not apply a load-conditioning policy. Also not accepting new connections does not give the user any indication about what is going on. SEDA carefully controls the flow of



(a) Threaded



(b) Event-driven

Figure 4.2: Two different server designs. Image is taken from [71].

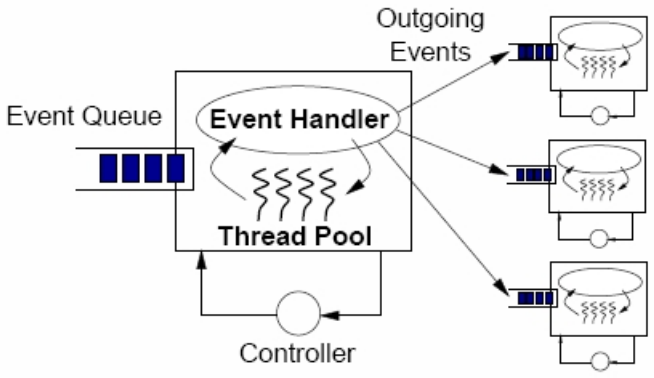
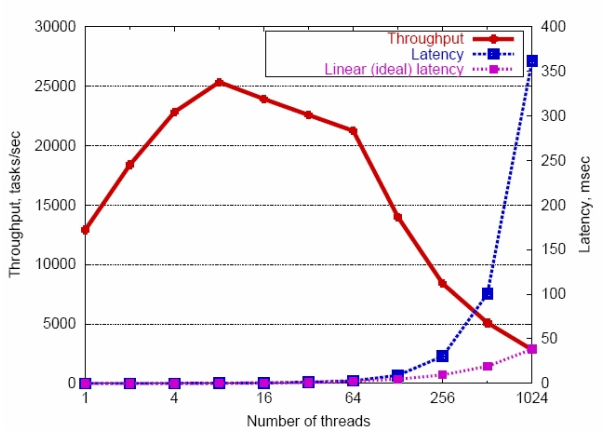


Figure 4.3: A SEDA Stage.Taken from [32]

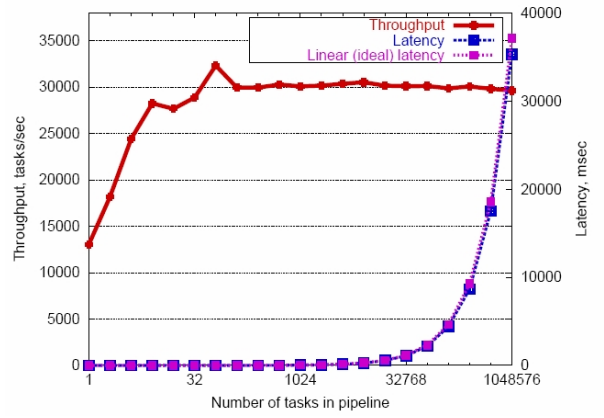
requests through the system; i.e., by only rejecting those requests that lead to a bottleneck resource. It also defines an action to be taken when the system is under heavy load, for example forwarding the request to another node, instead of simply rejecting the connection at all.

4.3 Discussion

A push-based model can offer a better synchronization between the client and the server with low communication overhead, comparing to a pure pull approach. However, implementing an HTTP based push approach is not trivial. If the system is not designed properly, the server will be saturated quickly, and will not be able to serve new clients properly. There are several hybrid approaches that combine pull and push techniques [15, 1, 32], which produce better results than a pure push approach. Further research is needed in order to find and test adaptive push-pull algorithms, that lead to optimum results. An event-driven architecture such as SEDA [72, 71], is necessary on the server side. This architecture avoids the high overhead associated with thread-based concurrency models, and decouples event and thread scheduling from application logic. As we will see in Section 6.2.3, a naive push/subscribe model will suffer from high load, leading to service degradation. By measuring and controlling the resources efficiently, scalability can be achieved. Jetty web server, which we discuss in Section 7.1, can decrease the number of needed threads dramatically by using non/blocking Java NIO package [62] and using an event-driven mechanism.



(a) Threaded



(b) Event-driven

Figure 4.4: The comparison of server performances. In threaded design, the throughput begins to decrease substantially after the number of threads grows large. In event-driven design, the throughput remains constant, as the load is increases to a very large number of tasks. Image is taken from [71].

Chapter 5

Push and Event models in practice

The push model we mentioned in Section 4.1 can be recognized in most of the push systems in the industry. Most of them however lack the *Repeater* component and the *Broadcaster* is usually more complex, usually because of scalability reasons. For example Grizzly [7], which is an HTTP Listener included in GlassFish project [1] [6], uses SelectorThread object to accept incoming connections and other components (such as Tasks, Pipelines, Handlers) to deal with requests internally. In Pushlets [28] the broadcaster component is Publisher. The list of subscribers is handled by PushletSubscriber object. In Jetty [20], SelectChannelConnector component is used for asynchronous handling of AJAX requests.

Jetty and Grizzly use Java NIO package [62], which allows asynchronous IO to be used and threads to be allocated to connections only when requests are being processed. Java NIO is based on Reactor pattern [57]. In this pattern (See Figure 5.1), Reactor object responds to IO events by dispatching the appropriate handler. Event Handlers, (which are given as "read, decode, compute, encode, send" on the figure) perform non blocking actions. Reactor object is similar to a java AWT [63] thread, Event Handlers are similar to AWT *ActionListener*, and binding the handlers to events is similar to AWT *addActionListener*.

In Java NIO a *Selector* object performs the task of *Reactor* and tells which of a set of channels have I/O events. *SelectionKey* represents the binding between the handler and event, and maintains I/O event status. NIO uses the concept of channels, *Selector* object demultiplexes events on the channel, when an event occurs.

There are several quality attributes that an effective event based server mechanism must fulfill [57]. We summarize them below and analyze to what extend Java NIO supports them.

- **Availability:** The server must be available to handle incoming requests even if it is waiting for other requests to arrive. Asynchronous I/O in NIO does not block. Instead, it listens for I/O events on an arbitrary number of channels, without polling and without extra threads. It can listen multiple ports at the same time, and deal with connections from all of those ports in a single thread.

- **Efficiency:** A server must minimize latency and maximize throughput.

[1] code base for the Sun Java System Application Server

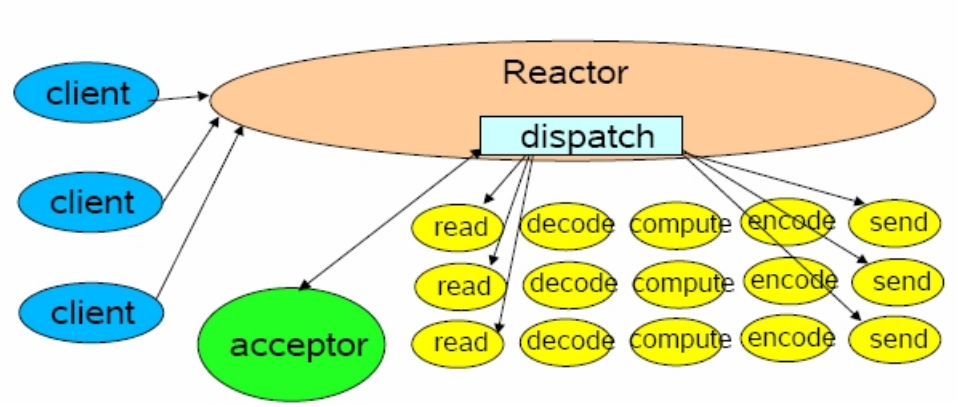


Figure 5.1: Reactor Pattern. Taken from [41].

- **Programming simplicity:** The design of a server should simplify the use of suitable concurrency strategies. NIO achieves this by separating handlers from event dispatching, so that a new service does not need to bother with event demultiplexing and dispatching.
- **Modifiability:** Integrating new or improved services should incur minimal modifications and maintenance. NIO decouples application functionality into separate classes using separate handlers. Therefore, a change in a particular functionality only affects the implementation of that handler class.
- **Portability:** Porting a server to a new OS platform should not require significant effort. Since Java is portable, NIO can be used, independently of the OS system calls that perform event demultiplexing.

Chapter 6

Web Based Push Solutions

As we have mentioned in Section 4, in REST style, each request can only generate a single response and must be initiated by a client. This makes a server-initiated request impossible. There are several solutions used in the practice that still allows the server to send updates to the client when they occur. In this section we will analyze them.

6.1 HTTP Pull

Most AJAX applications check with the server at regular user-definable intervals known as *Time to Refresh* (TTR). This check occurs whether the application's data is still consistent or needs to be updated. This pattern is actually the opposite of push approach. In order to achieve high data accuracy and data freshness, the pulling frequency has to be high. This will induce high network traffic and possibly unnecessary number of messages. The application will waste a lot of time querying for the completion of the event, thereby directly impacting the responsiveness to the user. In addition, a lot of network bandwidth will be wasted. However, this scheme is frequently used in web push systems, since it is robust, simple to implement, allows for offline operation, and scales well to high number of subscribers [32]. Mechanisms such as Adaptive TTR [15] allow the server to change the TTR, so that the client can pull on different frequencies, depending on the change rate of the data. This dynamic TTR approach in turn provides better results than a static TTR model [58].

6.2 COMET - HTTP Based AJAX Push

HTTP Streaming is a basic and old method that is used first in 1992 by Netscape, in the product Navigator 1.1. In this approach the server sends data which is displayed by the browser but the connection between server and client remains open. Later the server may continue to send other pieces of data to the client. The connection needs to be open, because HTTP requests can only emerge from the client. When a state change occurs, there is no way for a server to open connections to interested clients. The application of this scheme under AJAX applications is now known as *Comet* [19, 67]. Comet tries to achieve the server to send a message to the client when the event occurs, without the client having to ask for it. As we can see from Figure 6.1, when an event occurs at the server side, the client is directly notified. The data is delivered over a

single, previously-opened connection. This is not the case with the general AJAX approach (See Figure 3.2b). Thanks to Comet, the client can continue with other work and work on the data generated by the event when it has been *pushed* by the server. Comet does not suffer from the problems mentioned in 6.1.

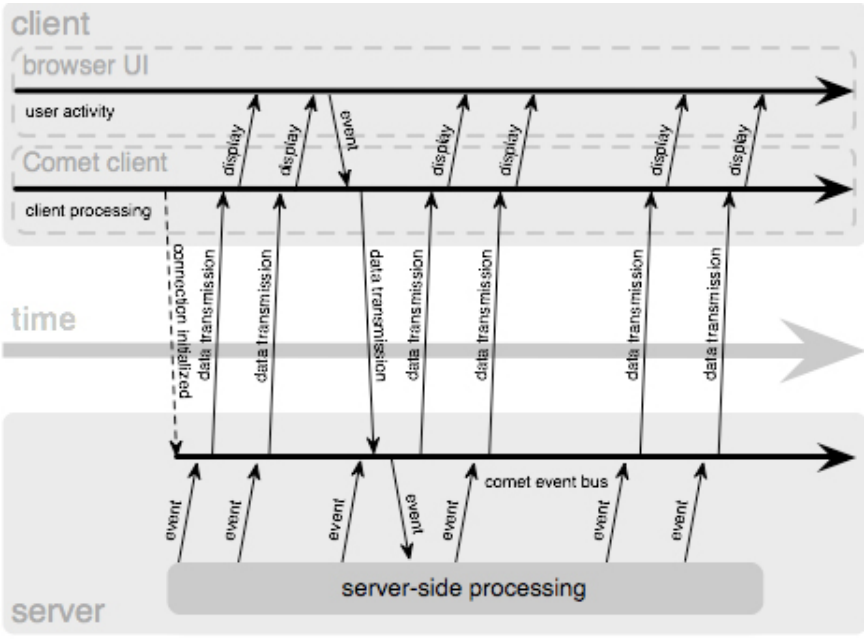


Figure 6.1: Comet web application model. Taken from [67].

Comet scheme is available thanks to the ”persistent connection” feature brought by HTTP/1.1. With HTTP/1.1, unless specified otherwise, the TCP connection between the server and the browser is kept alive, until an explicit ”close connection” message is sent by one of the parties, or a timeout/network error occurs. Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. With persistent connections, fewer TCP connections are opened and closed, leading to a save of CPU time in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches). Also, memory used for TCP protocol control blocks can be saved in hosts. According to the Hypertext Transfer Protocol (HTTP/1.1) [24], HTTP implementations SHOULD implement persistent connections.

Since HTTP/1.1 [24] clearly states that the protocol must follow the request/response scheme, this is the only way of implementing a push solution under HTTP. Once the connection is closed, there is no way for the server to start a connection back to the browser.

Comet approach has been implemented and included in several web servers including Jetty [20] and IBM Websphere [33]. Both these solutions are following the Bayeux protocol [56]. Bayeux protocol currently supports the following connection types for HTTP push:

1. IFrame: IFrame is an HTML element which makes it possible to embed another HTML document inside the main document [74]. The content of the embedded document can be replaced without reloading the main document, by using JavaScript. Many AJAX applications use this feature by implementing an invisible IFrame to communicate with the server asynchronously, instead of XMLHttpRequest [70].

2. Long Polling: Long polling is a mix of pure push approach and client pull. After a subscription to a channel, the connection between the client and the server is kept, for a defined amount of time (by default 45 seconds). If no event occurs on the server side, a timeout occurs, and the server asks the client to reconnect. If an event occurs, server sends the data to the client, and the client reconnects.

Bayeux protocol defines the following phases in order to establish a Comet connection:

1. Client performs a handshake with the server, receives a client id and list of supported connection types (IFrame, long-polling, etc.).

2. Client sends a connection request with its id and its preferred connection type.

3. Client later subscribes to a channel and receives updates

The Bayeux message format is defined as JSON [22]. The protocol also intends to deal with heavy load by giving different ”advices” to different clients, such as ”Reconnect after 30 seconds”. The server can also send “ping” messages to check the status of the client. See Section 7.1.1 for Jetty’s Bayeux implementation.

In the following subsections we summarize different forms of Comet and discuss possible problems.

6.2.1 Page Streaming

This is simply streaming server data in the response of a long-lived HTTP connection. Most web services do some processing, send back a response, and immediately exit. But in this pattern, the connection kept open by running a long loop. The server script uses event registration or some other technique to detect any state changes. Immediately after a state change occurs, it pushes new data to the outgoing stream and flushes it, but does not actually close it. Meanwhile, the browser must ensure the user-interface reflects the new data.

”Page Streaming” involves streaming the original page response. Here, the server immediately outputs an initial page and flushes the stream, but keeps it open. It then proceeds to alter it over time by outputting embedded scripts that manipulate the DOM. The browser is still officially writing the initial page out, so when it encounters a complete <script> tag, it will execute the script.

This solution has an important advantage. It will work on any browser with JavaScript capability.

6.2.2 Service Streaming

The XMLHttpRequest object is an interface exposed by a scripting engine that allows scripts to perform HTTP client functionality, such as submitting form data or loading data from a server [70]. Service Streaming relies on XMLHttpRequest object. This time, it's an XMLHttpRequest connection that's long-lived, instead of the initial page load. This brings some flexibility regarding the length and frequency of connections. The page will be loaded normally (one time), and streaming can be performed with a predefined lifetime for connection. The server will loop indefinitely just like in page streaming, but the output can no longer be HTML script tags, because the web browser will not automatically execute them. The browser has to poll the latest response (*responseText*) to update its content.

6.2.3 Problems Facing HTTP Based AJAX Push

As mentioned in Section 4, according to the HTTP protocol, the server can not initiate a connection, therefore it has to hold on to the connection started by the client. This however causes the following problems:

- **Scalability**: Comet uses persistent connections, so a TCP connection between the server and the client is kept alive until an explicit disconnect, timeout or network error. So the server has to cope with many connections if the event occurs infrequently, since it needs to have one or more threads for every client. This will bring problems on scaling to thousands of simultaneous users. There is a need for better event-based tools. Operating Systems have an edge-triggered event mechanism that supports non blocking I/O. However, not all the servers support this. OS's have edge-triggered event IO mechanism (epoll [55] on Linux and kqueue [42] on FreeBSD). For the web server, there are several high-level event-driven IO abstractions available such as Perl's PEO [18], Python's Twisted [40], Jetty's Continuations [21] based on Java NIO [62] and Apache's MPM event [26]. AsyncWeb [10], a high-throughput non blocking Java HTTP engine claims that it can handle 9000 concurrent connections by using an event-driven mechanism. However, most of these solutions are still experimental.

- **Connection lifetime:** Most servers define a lifetime for an HTTP connection. The connection lifetime should be carefully chosen during implementation. If the chosen lifetime is too small, it will cause many connection re-setups, which is a costly process. On the other hand a big value may cause the server to push to clients who are no longer interested in a piece of information, thereby incurring unnecessary message overheads.

- **Maximum number of connections:** According to W3C's HTTP 1.1 spec [69] a single-user client should not maintain more than 2 connections with any server or proxy. Some browsers (e.g. IE 6.0) therefore allow a maximum of 2 connections per client. The intention here is to reduce the load on servers, but this might lead to performance issues in AJAX applications and different pages/applications in the same server might not be available to the client. For example, if a series of images is being loaded into the browser, other updates can not be loaded. There

are workarounds for this such as creating a separate hostname for the push connection (It can also be a physical server). Jetty (See Section 7.1) currently deals with this by counting the number of push connections established by the client. If multiple push connections are detected, the client is asked to switch the second connection to pull. Note however this solution is experimental.

- **Proxies:** These are servers that are located between the client and the server. If their content is not up to date (mostly not), real-time data will not be flowing into the browser.

- **Firewalls:** Firewalls are typically configured to drop connections that have been open for too long (as a security heuristic). Most frameworks [52] that are implementing push technology recreates the ”push” connection periodically. A “ping” message can be a possible solution here, to check if the connection is still alive and to prevent the firewall closing the connection.

Other than the problems above, *Page Streaming* has a big problem with memory limitations. In this scheme, JavaScript keeps accumulating, so the browser must retain all of that in its page model. In a rich application with a lot of updates, the model is going to grow quickly, rising the need for a page refresh to avoid hard drive swapping. *Service Streaming* is more flexible in this issue, because arbitrary content (other than JavaScript commands) can be sent to the client. However, according to user reviews, it does not work on all browsers [3]. It also suffers from the issues mentioned above. It performs better on memory thanks to XMLHttpRequest, the JavaScript Engine will not be filled quickly in this case.

6.3 Flash XML Sockets

The XMLSocket object of Flash Technology implements client sockets that allow the computer running the Flash Player to communicate with a server computer identified by an IP address or domain name. To use the XMLSocket object, the server computer must run a daemon that understands the protocol used by the XMLSocket object. Note that the connectivity is not provided with HTTP, Even though Flash has HTTP-based XML server connectivity, this solution uses its own protocol [45].

The protocol is as follows:

- XML messages are sent over a full-duplex TCP/IP stream socket connection.

- Each XML message is a complete XML document, terminated by a zero byte.

- An unlimited number of XML messages can be sent and received over a single XMLSocket connection.

The XMLSocket object is useful for client-server applications that require low latency, such as real-time chat systems. Just like HTTP Streaming, XMLSocket maintains an open connection to the server, which allows the server to immediately send incoming messages without a request from the client.

The advantages of using a Flash solution is as follows:

- Cross Browser Support: in AJAX technology, different browsers have different names and usages for JavaScript objects (XMLHttpConnection for example). Flash, as a standalone plugin, does not have this issue

- Efficiency and speed: It is possible to produce robust and efficient solutions with Flash. XML parser of Flash is easy to use. See Section 7.6 for Fjax, which uses 65lines of code, with a 4K small footprint.

- Scalability: Now the connection is not kept open, since the server can issue a request and contact the socket on the browser side at any time. This results in less threads on the server side, leading to better scalability.

The main disadvantage of Flash is its availability. Even though most browsers are released including the Flash plugin, support of legacy systems might be a problem. It might also be the case that several systems won't allow the execution of the plugin.

A problem that occurs within sockets, not Flash sockets in particular, is the issue with the firewalls. As Martin-Flatin points out [46], repeated socket creations and time-outs cause network and CPU overhead, but it also requires the client to create a new connection if the previous times out. Secondly, most firewalls filter out UDP, let only a few TCP ports go through. If the server initiates the connection, the firewall on the client side can block it. Therefore, to make this socket-based solution work, the firewall systems need to be modified. This may well be a problem for small and medium-sized enterprises [46].

6.4 Java RMI

Applets on browsers allows Java Remote Method Invocation (RMI) [61] to be used. RMI provides the possibility of invoking the methods of remote Java objects from other Java virtual machines, possibly on different hosts. Once an RMI client has bound to an RMI server, both of them can send data to the other. RMI is focused on Java, and uses Object Oriented technology. It allows passing complex types and using design patterns. This makes it easier to design complex applications. RMI will face the same problem with firewalls as the Flash solution, which was mentioned in Section [2]. This is because RMI is actually based on sockets, which are transparent to applications [46].

6.5 Server Sent Protocol

Another push/subscription solution is "'server-sent event"' protocol, mentioned in the draft of WHAT-WG group [73]. To use server-sent events in a web application, an "element" is added to the document, with an "src" attribute pointing to an event source URL (the server). This URL should provide a persistent HTTP connection that sends a data stream containing the events. The connection must use the content type "application/x-dom-event-stream". Opera 9 browser already supports this feature (See [9]), however, it will probably take some time until the work is finalized, and a universal browser support is established.

6.6 Discussion

Comet [19, 67] is a generic solution that has no problems with firewalls. It does not require any plugin download with the modern browsers, as it only uses JavaScript engine of the browser. However, this does not directly result in high portability, since different browsers work with different JavaScript engines. This results in additional code for different browsers, adding to the complexity of AJAX approach itself. Flash sockets offer a solution that can be used in any browser that contains the plugin. It has a positive effect on simplicity of the code, since the development effort that is needed to understand, design, and implement the system will be less than Comet approach. However not all computer environments might have the plugin, and problems with the firewalls might become an issue. The overhead that comes with socket creation should also be remembered. Both solutions come with their own advantages/disadvantages. The decision on which solution to choose depends on which trade-off we are willing to take.

Chapter 7

Current Push Implementations

This section introduces a couple of examples from the software industry that use the solutions mentioned in the previous section.

7.1 Jetty Server

Jetty [20] is an open-source web server implemented entirely in java. Jetty can be used as a stand-alone traditional web server for static and dynamic content, as a dynamic content server behind a dedicated HTTP server such as Apache , or as an embedded component within a java application. Jetty contains a Comet module which implements the Bayeux protocol [56], which we have mentioned in Section 6.2. In the next subsection, the Bayeux implementation of Jetty will be discussed. Later, Continuations, a mechanism which allows a scalable Comet solution will be mentioned.

7.1.1 Bayeux Implementation

Figure 7.1a shows the handshaking,connection and subscription phases of a Jetty Comet setup. The tasks on the server side are divided into several components:

- CometdServlet: A servlet that handles the client's request and calls the specific handle, by checking the type of the request.
- Handlers: Each type of handler performs a different task. HandshakeHandler takes care of connection handshaking. SubscribeHandler takes care of subscription to a particular channel, etc.
- Transport: This is the component that sends the actual response to the client. It sends specific output, depending on the client's connection type

The initial handshake and subscription phase works as follows:

1. Handshake: The Client (Browser) sends a handshake request to the server. CometdServlet calls the HandshakeHandler. The handler creates a random id for the client, sends the type of connections it supports, by using the Transport component.

2. Connect: The Client sends a connection request to the server, with the client id it obtained and the connection type it prefers. CometdServlet calls ConnectHandler. Connecthandler sends a connection id and time stamp via the Transport component.

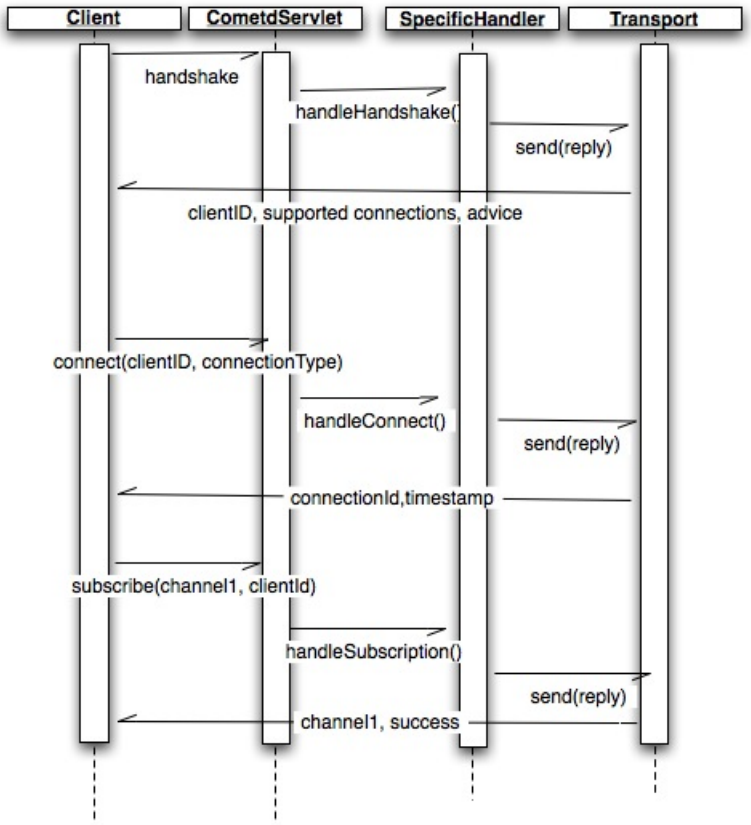
3. Subscription: The client sends a subscription request, specifying its clientID and also channel it wants to subscribe. CometdServlet calls the SubscribeHandler. SubscribeHandler sends a confirmation if the request is accepted.

Figure 7.1b shows the reconnection phase with ”long polling”. After every response from the server, the client will send a reconnect with its clientID and the timestamp of the last message it has received. This way the server can compare the timestamp of the client and timestamp of the other messages, and resend them to the client if necessary. This way the client does not miss an update. If an event occurs on the server side (database update or a message published by another user), it will simply be returned to the client. If no event occurs during a predefined interval (by default this is 45 seconds in Bayeux), a timeout will occur and the server will send just the timestamp, asking the client to update its own timestamp and to reconnect. Since the users reconnect every time they receive data, the system acts as a mix of push and pull. Therefore less threads are occupied at a given time and this help scalability. However, under heavy load a delay can occur, if the client can not reconnect. This is not acceptable in certain applications, where the time of the delivery is critical.

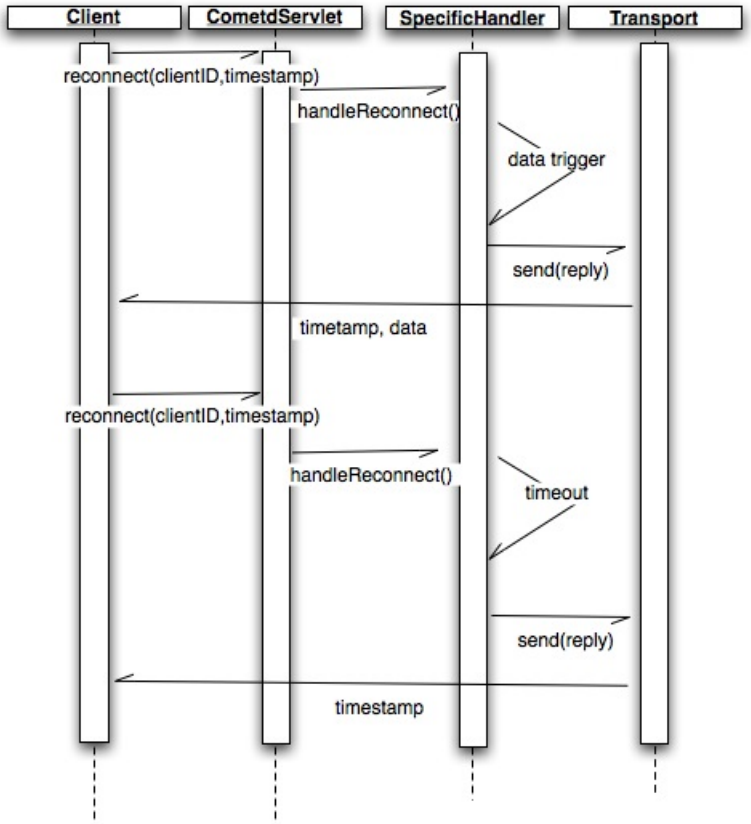
7.1.2 Continuations

The traditional Java IO model associates a thread with every TCP/IP connection. Even though modern JVM's [59] are much better at handling large numbers of threads, this is not very scalable if HTTP or page streaming is used, as the memory usage of every thread stack and buffer is not insignificant. As mentioned in Section 5, Java NIO package [62] allows the threads to be allocated to connections only when requests are being processed, instead of during the whole connection. When the connection is idle, associated thread and buffers can be returned to pools for reuse and the connection can be added to an NIO select set to wait for new activity without requiring significant resources. Once a request is received on a connection, a thread and buffer are allocated and the synchronous servlet API invoked to handle the request. This thread-per-request scheme works on the traditional web model, but an AJAX application using the push (COMET) model will have long standing requests that are kept open by the server to allow asynchronous responses to be sent by the server to the client. This is a great technique, but it breaks the thread-per-request model, because now every client will have a request outstanding in the server. Thus, the server again needs to have one or more threads for every client, and again there are problems scaling to thousands of simultaneous users.

Java does not provide a mechanism to really suspend a thread and then resume it later. Jetty implements a method called Continuation [21]. A continuation is a mechanism by which a HTTP Request can be suspended and restarted after a timeout or an asynchronous event has occurred. Continuation allows both a blocking and non-blocking mechanism to handle requests. Blocking continuations blocks the process of the request. Non-blocking continuation can abort the current request and arrange for



(a) Connection handshake and subscription



(b) Reconnects after data events and timeouts

Figure 7.1: Jetty's Comet event subscription and event delivery based on Bayeux protocol [56]

it to be retried later or the timeout expires. With the appropriate HTTP Connector, this allows threadless waiting for events.

For example, SelectChannelConnector is a connector that receives requests (normally from a socket) and calls the handle method of the Handler (typically the server) object. When a filter or servlet invokes a Continuation, SelectChannelConnector throws a runtime exception that allows the thread to exit the current request handling. Jetty catches this exception and will not send a response to the client. Instead, the thread is released, and the Continuation is placed on the timer queue. If the Continuation timeout expires, or it's resume method is called, then the request is again allocated a thread and the request is retried.

Figure 7.2 below shows the results of this Continuation approach. Continuations allow more concurrent requests (10700 versus 10600 of Comet with simple HTTP Streaming). The real effect can be seen on the minimum number of required threads and stack memory. Continuation solution reduces the minimum number of threads from 10600 to 875. Required stack memory is also reduced from 694MB to 57MB. It is obvious that Continuations bring a huge improvement over the scalability, comparing to Comet solution without continuations.

	Formula	Web 1.0	Web 2.0 + Comet	Web 2.0 + Comet + Continuations
Users	u	10000	10000	10000
Requests/Burst	b	5	2	2
Burst period (s)	p	20	5	5
Request Duration (s)	d	0.200	0.150	0.175
Poll Duration (s)	D	0	10	10
Request rate (req/s)	rr=u*b/20	2500	4000	4000
Poll rate (req/s)	pr=u/d	0	1000	1000
Total (req/s)	r=rr+pr	2500	5000	5000
Concurrent requests	c=rr*d+pr*D	500	10600	10700
Min Threads	T=c T=r*d	500 -	10600 -	- 875
Stack memory	S=64*1024*T	32MB	694MB	57MB

Figure 7.2: Performance and requirements of a Jetty web server using different schemes. Taken from [21]

7.2 Grizzly

Grizzly [7] is an HTTP Listener using Java's NIO technology [62] and implemented entirely in Java. It can be used for any HTTP related operations (HTTP Listener/Connector) as well as non-HTTP operations.

To support Comet functionality, Grizzly uses the following components [8]:

- **SelectorThread:** used to accept incoming connections. This fulfills the task of *Reactor* from the Reactor pattern (See Section 5).
- **CometContext:** A shareable space that is either subscribed by the application or created if it doesn't exist. When the context is updated, the application is also updated.
- **CometEvent:** An object containing the state of the CometContext (content updated, client connected/disconnected, etc.).
- **CometHandler:** The interface an application must implement in order to be part of one or several CometContext.

Comet in Grizzly works as follows [8]:

1. The application creates a new Comet context (or registers to an existing one)
2. Clients open connections to the SelectorThread, and a CometHandler is created per connection.

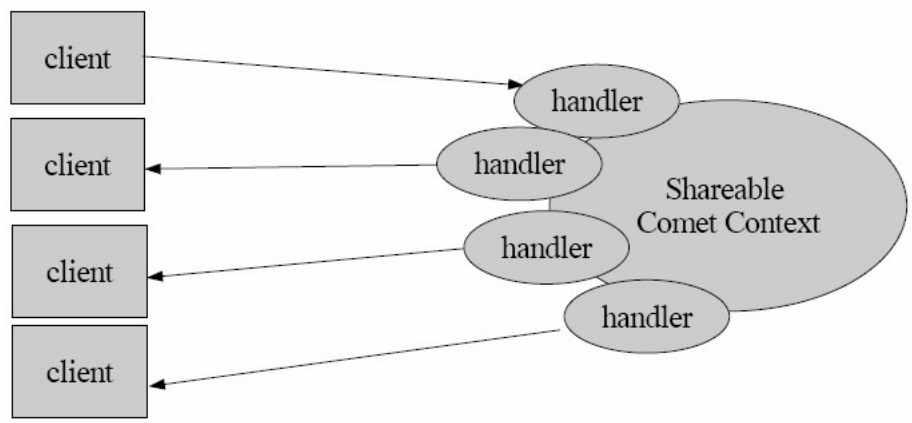
Now, whenever a client pushes a data, all the others will also be updated (Figure 7.3a). If a server-side event occurs, such as a database update, all subscribers are also notified (Figure 7.3b).

7.3 ICEfaces

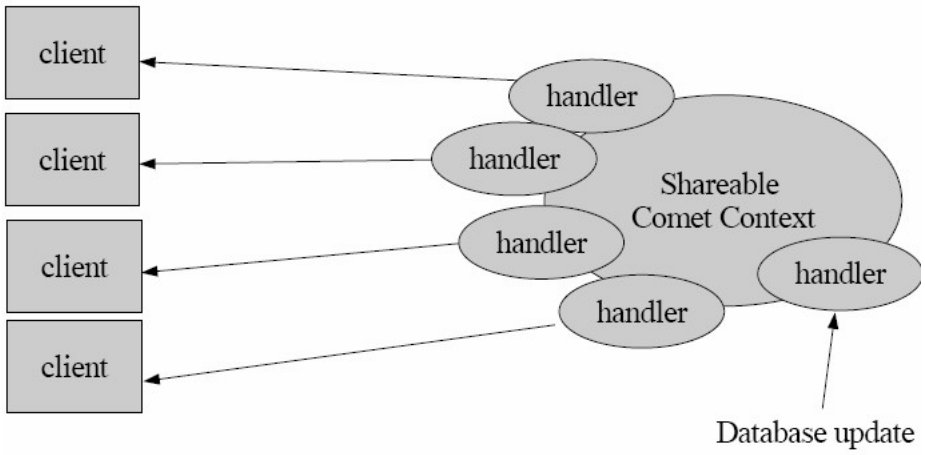
ICEfaces [34] is an integrated AJAX application framework that is developed by ICEsoft Technologies. It enables Java EE application developers to create and deploy thin-client Rich Internet applications (RIA) in Java. ICEfaces comes in different variants, which include a free "community" version, but also an "Enterprise Production Solution" (EPS). Both versions support HTTP push mode by using Jetty Continuations (See Section 7.1).

EPS version also supports features such as "Connection Management", "Asynchronous HTTP Server" and "Cluster Support". Asynchronous HTTP Server [35] is an HTTP server that handles long-lived asynchronous XMLHttpRequests by using Java NIO [62] and Java Messaging Service (JMS) [60]. It is deployed as a separate application in the application server (Such as Apache Tomcat [4] or IBM Websphere [33]). As we can see from Figure 7.4, individual ICEfaces application deployments use JMS to communicate with the Asynchronous HTTP Server. Requests are queued at the Asynchronous HTTP Server on a per client basis, and as responses become available, they are communicated to the Asynchronous HTTP Server, again via JMS. A response is matched to its associated blocking request, the request is unblocked, and the response is returned to the client. The deployment architecture is fronted with an HTTP web server that filters blocking (push) request and directs them to the Asynchronous HTTP Server for processing. Non-blocking (non-push) requests are passed directly to the ICEfaces application via an application server-specific plug-in.

ICEfaces employs several techniques to deal with problems mentioned in Section 6.2.3. In order to deal with scalability, Java NIO and a configurable thread pool is used. Problems with connection lifetime, such as response timeout, expired session,



(a) Client sends an update and others receive it



(b) Server starts an event and all subscribers are notified

Figure 7.3: Two different type of updates in Grizzly. Images are taken from [8].

or empty response are handled by so-called ”Connection Management”. The state of the client (Idle, waiting, lost, etc.) is kept in a component. The status is checked with ”Asynchronous Heartbeating” mechanism, which is similar to Bayeux [56] protocol’s ping messages. This mechanism closely monitors the connection status based on heartbeat responses (by default every 20 seconds). If this component is present in the page, it will be updated as connection status changes. The possible client states are:

- idle
- waiting
- caution
- lost

The ”caution” state occurs if heartbeats go missing, but retries are in progress. If the retries fail, the connection state will transition to ”lost”, and if a retry succeeds the connection state will return to ”idle”.

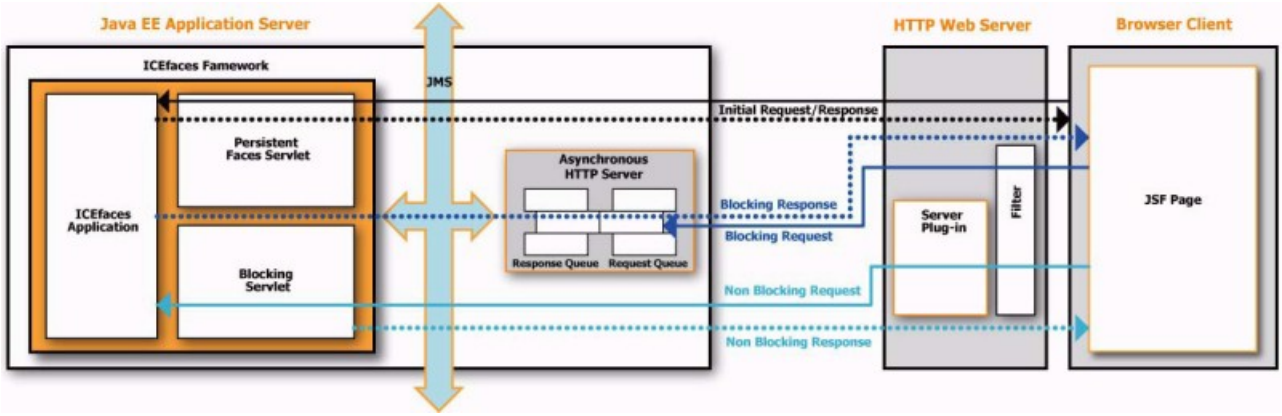


Figure 7.4: ICEfaces EPS Deployment Architecture. Taken from [35]

7.4 Pushlets

Pushlets [28] 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. Through a single generic servlet (the Pushlet), browser clients can subscribe to subjects from which they like to receive events.

Pushlets include two libraries, one with iframes and one with an AJAX pushlet client in JavaScript, which uses XMLHttpRequest and XML. Iframes library uses *Page Streaming*, which we have mentioned in Section 6.2.1. AJAX library uses the XMLHttpRequest object and employs the push model, which was mentioned in Section 4.1.2. The *Broadcaster* component from the push model [32] is realized by *PushletSubscriber* and *Pushlet* classes. By invoking the Pushlet object through an HTTP request, clients subscribe to receive *Events*. A basic event subscription works as follows (See Figure 7.5).

1. Browser invokes the Pushlet with the method *doGet()*
2. Because multiple clients may invoke the same Pushlet object, it does not handle subscription itself. It delegates all subscription (and subsequent Event handling) by creating a PushletSubscriber object for each *doGet()*.
3. The PushletSubscriber creates an Adapter (ClientAdapter in Figure 7.5) to deal with different client formats and protocols. It later registers itself with the Publisher object, which deals with events.

Sending and dispatching events works as follows (See Figure 7.6.

1. EventGenerator object creates an event and notifies the Publisher
2. The Publisher checks with every PushletSubscriber, to see if the subscribed client is interested in the event. If the event ”matches”, then it sends the event.

3. PushletSubscriber sends the event to the Adapter (BrowserPushletAdapter in Figure 7.6).

4. Adapter sends the event details in JavaScript to the browser.

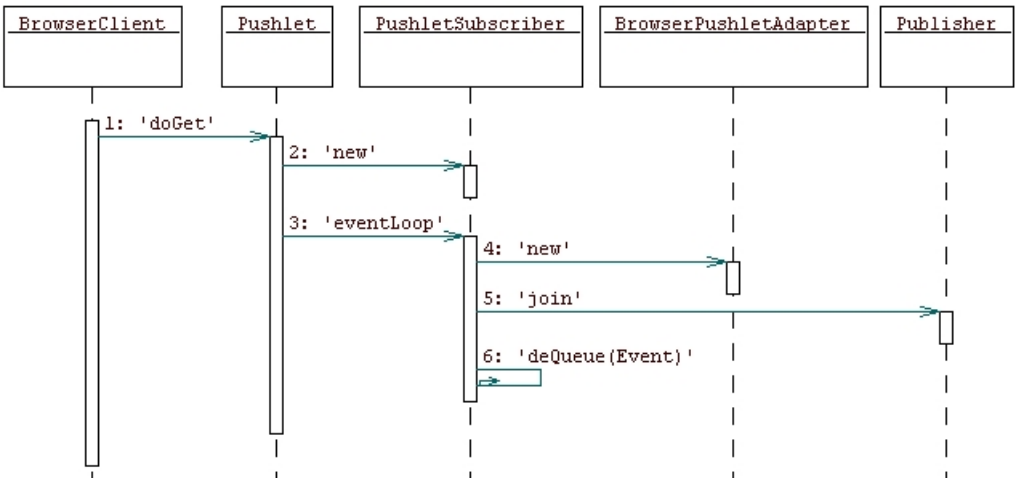


Figure 7.5: Subscription mechanism of the Pushlets, taken from [28]

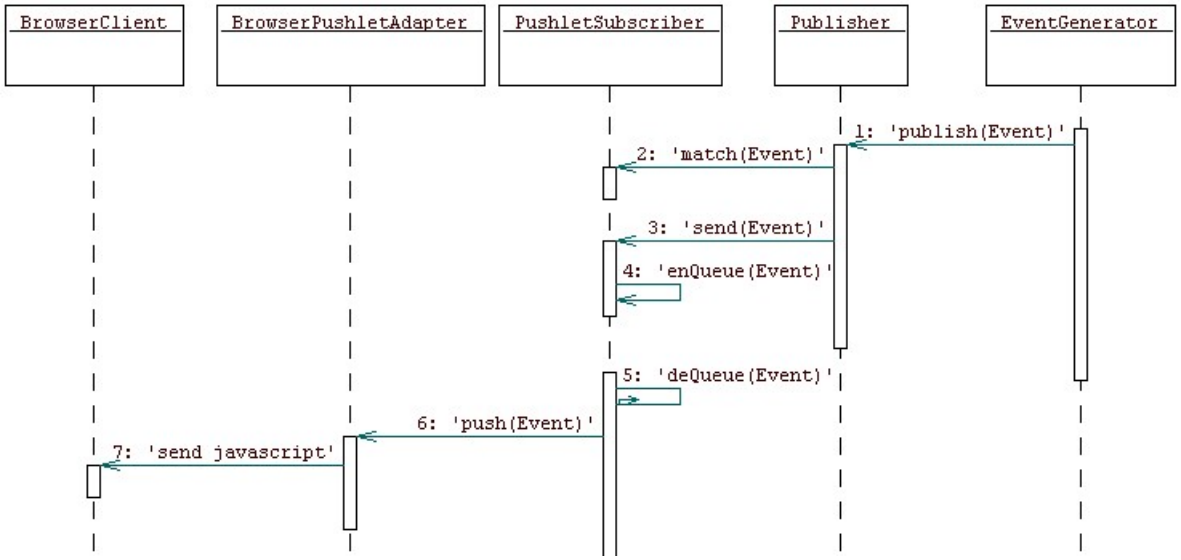


Figure 7.6: Event dispatch mechanism of the Pushlets, taken from [28]

Publisher uses PushletSubscriber objects to send events, instead of sending the event directly to the adapter. The reason here is to notify multiple clients; i.e., if one client is very slow and is standing in front of the queue, then all the fast clients will have to wait for the slow client to finish. However, this design decision needs a thread

per client. As we have discussed before, threads are costly resources, and handling 100's of clients in a single server can be a problem.

Pushlets are a good first step in implementing a HTTP push solution. It supports clients that are written in Java J2SE, J2ME or Flash. Actually any client that can receive XML over HTTP can be used.

7.5 Lightstreamer

Lightstreamer Server is a commercial push/streaming engine based on a staged event-driven architecture, that handles the physical connections with the clients. It streams data through HTTP and HTTPS connections to any type of client (Web clients, application clients, etc.).

Lightstreamer Server is a stand-alone process, which does not rely on web servers or application servers. This approach was taken in order to have direct control over the operating system's TCP/IP stack and to optimize the data transmission. The Kernel of Lightstreamer Server has the task to distribute the data to the clients in a reliable and efficient way.

Lightstreamer also contains a plug-in module Data Adapter which interfaces Lightstreamer with the data source (or "feed") to be integrated. The Data Adapter receives a flow of data from the back-end systems (information provider, data feed, database, etc.) and makes it available to Lightstreamer Server for controlled delivery to individual users. The Data Adapter can use any technology to integrate with the data feed. It is also possible to use some technology that allows an asynchronous paradigm (such as message-oriented middleware: JMS, TIB/RV, MQ, etc.) in order to avoid a break with the asynchronous chain that goes from the feed to the user's browser. Lightstreamer also supports polling techniques to refresh the data (e.g. reading from file or from database).

This approach deploys a server along a web server, in which static pages are sent to the user via web server, and dynamic data is sent by Lightstreamer server. It is able to update the data contained in an HTML page with push Ajax approach (with a JavaScript library)

According to the company, thanks to staged event-driven architecture, the solution is very scalable and efficient for handling very high loads. The number of threads does not grow with the number of concurrent users. It also uses Java NIO with advanced management of network sockets, with an increase on performance and scalability. The company claims that a LightStreamer server can handle many thousands of concurrent users for each CPU. See [44] for detailed information.

7.6 Fjax

Fjax is a lightweight, cross-browser, rapid-development methodology for AJAX-style web 2.0 development that puts a Flash engine (for XML parsing) under the hood to handle realtime XML/HTML content updating [47].

Fjax operates as follows:

1. The (x)HTML webpage loads into the browser on the visitor's computer, bring-

ing with it the included JavaScript file. It is also what pulls in the Flash swf file.

2. The JavaScript tells the swf which XML file it wants, and the swf goes to the server to get it.

3. The XML file containing the (X)HTML content you want brought into the page gets pulled through the swf which hands it back to the JavaScript which then pops it into the DIV tag where you want it.

7.7 iPush Server

iPush server is a commercial server operating as a front-end to a web-server, handling incoming data from the clients and also stream data towards them. It requires the clients to have either a Flash Player plug-in or Java's JRE [64]. According to the company, the throughput is 30,000 messages per second on an average network usage of 26% and an average CPU usage of 50%. The throughput increases to 105,000 messages per second if the cpu load decreases to 45% and network usage to 13%. For more information see [66].

7.8 Discussion

All Java web applications that implement COMET approach make use of an event-driven architecture on the server side, to avoid the high overhead associated with thread-based concurrency models. The need to have an open standing connection for every push subscriber, causes scalability problems in thread-based systems. Therefore, asynchronous IO tools such as Java NIO is needed on any web server that is implementing the COMET approach. Frameworks such as ICEfaces also support load balancing, clustering and connection management to support even more clients with high availability. Such features should be considered during the design of a new web server or the modification of a current one.

Chapter 8

Discussion

AJAX technology has been a shift in the direction of web development. Thanks to the momentum of AJAX, *Single Page Interfaces* (SPI) have attracted a strong interest in the web application development community. The *Multi Page Interface* (MPI) model is slowly evolving into SPI. The same will happen for the Push technology, such a model will be very interesting to implement for certain types of applications. A software engineer adopting this model, however, will be wondering about the effects of this to software quality attributes. In this section we will be discussing architectural properties and software quality attributes, taken from Bass et al. [12] and Mesbah and van Deursen [48]

8.1 Scalability

In Web engineering, a system's scalability is determined, for instance, by the degree in which a client can be served by different servers without affecting the results [48]. Scalability is the main issue in a Push model. Web servers usually create a thread per client (connection). Libraries such as Java NIO [62] can achieve a thread per request, instead of per connection. However, an HTTP based AJAX push solution will have an outstanding request waiting on the server which will be used to send a response to the client the instant an asynchronous event occurs. This however will break the "thread per request" model. Better technologies are needed on the server side, in order to serve a growing number of client requests.

8.2 Availability

A pure "push" server is less resilient to failures, due to its stateful nature and its list of subscribers. The server has to keep the state, so when the state changes, it will broadcast the necessary updates. The amount of state that needs to be maintained can be large, especially for popular data items [15]. The state information is lost if the server fails and requires the client to detect the failure and re-register itself. Note that this extra cost of maintaining a state and a list of subscribers will also have a negative effect on scalability.

8.3 Visibility

Visibility is determined by the degree in which an external mediator is able to understand the interactions between two components. This is already poor in current AJAX frameworks, as they are based on proprietary protocols [48]. If we look at the current push approaches, we see different techniques on achieving the push solution itself, but also different measures to deal with scalability. Without a standard here, it will be difficult for a mediator to understand the interactions between system components, therefore the system itself will be less visible. Bayeux protocol [56] is quite simple and if more developers implement it, this will have a positive effect on visibility.

8.4 User-perceived Latency

User-perceived latency is defined as the period between the moment a user issues a request and the first indication of a response from the system [48]. AJAX improves the responsiveness of a UI for a single user, but changes to data from others users are lost until a user refreshes the whole page. The user has to either wait until he/she performs some action which would kick off a request to see the updated state from other users, or ”poll” the latest data in pre-determined intervals. Comet solves this by improving application responsiveness without the performance problems associated with polling. Now the user does not have to keep refreshing the site to get the latest data. The data is presented to the user asynchronously, eliminating the latency caused by the request/response approach of REST [39] . Getting fresh updates also has positive effect on user's perceived satisfaction and attitude towards a Web site.

8.5 Portability

Software that can be used in different environments is said to be portable [48]. As we have mentioned in Section 6.2.1, *Page Streaming* does not work in all the browsers. *Service Streaming* works on most modern browsers, but AJAX itself relies on Javascript [17], which may be implemented differently by different browsers or versions of a particular browser. Cross platform solutions such as Flash [2] is available, but they do require the need of an extra action from the user (downloading the plugin), therefore decreasing the portability.

8.6 Simplicity

Simplicity or development effort is defined as the effort that is needed to understand, design, implement, and re-engineer a web application [48]. Since AJAX relies on JavaScript [17], and different browsers have different JavaScript engines, it's not uncommon to see a JavaScript code written twice, a part for Internet Explorer [36], a part for Mozilla [50] compatibles, increasing amount of code and decreasing simplicity. So far Comet approach shows different behaviors on different browsers [3]. More code will be needed to support multiple browsers, leading to code complexity and less simplicity.

8.7 Network Performance

The *Push* approach has small overhead because only values of interest to a client are transferred over the network, and the transfer is done when there is actually an update. This results in less network activity, because clients do not produce requests periodically by pulling for the latest data. However because of the scalability problems mentioned in Section 8.1, a pure push server will not be feasible to implement for very big amount of users. Deolasee et al. [15] argues that it is difficult to determine whether a push- or pull-based approach should be employed for a particular data item. Hybrid push-pull approaches[15, 1, 32] might lead to better results, but a case study testing adaptive push/pull in Comet is needed.

Chapter 9

Conclusion

In this paper we have discussed push technologies for AJAX, allowing the server to send real-time event notification. We have summarized several push models from the literature and shown their applications in the software industry. As we have discussed before, the HTTP Push solution Comet is quite portable since it does not require any plug-in download and has no problems with firewalls. Comet can offer a better synchronization between the client and the server with low communication overhead, better network performance and low user-perceived latency, compared to a pure pull approach. However implementing such a solution is not trivial. Comet requires the server to hold on to the connection which was started by the client, since the REST model does not allow the server to initiate a request. Most web servers implement a thread-concurrency model, which creates a thread per connection. This will cause scalability problems with Comet applications, if the number of clients grow. The server will be saturated quickly, and will not be able to serve new clients properly. According to our research, servers which implement an event-based model scale better, maintain high throughput, and employ a linear response-time penalty that impacts all clients equally, or at least predictably. Thread-concurrency model can not satisfy this "graceful degradation", because of overheads associated with threading. Therefore, an event-based model on the server side is necessary. Open source solutions such as Jetty [20] and Grizzly [7] are already implementing this model by using Java NIO [62]. Tomcat 6.0 [5] also supports NIO with a separate module.

Another important problem with Comet is browser compatibility. The approach does not work in all browsers, and multiple code is needed to handle different browsers, which has a negative effect on simplicity. Cross-browser solutions such as Flash and Java RMI are available, however they have problems with firewalls. They also require plug-ins to be installed, which has a negative effect on portability.

Comet approach is quite popular and its protocol Bayeux is already implemented by Jetty, Grizzly and Websphere [33]. Such a protocol is important in order to make a standard for the interactions between the components. This will have a positive effect on visibility. Future work includes a case study that compares push under thread-concurrency and event-based models. We will also implement a push solution for Backbase AJAX Development Framework [11].

Bibliography

[1] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 183–194. ACM Press, 1997.

[2] Adobe flash player. http://www.adobe.com/products/flashplayer/, 2007.

[3] AjaxPatterns. Http streaming. http://ajaxpatterns.org/HTTP_Streaming.

[4] The apache tomcat 5.5 servlet/jsp container - documentation index. http://tomcat.apache.org/tomcat-5.5-doc/index.html, 2006.

[5] The apache tomcat 6.0 - advanced io and tomcat. http://tomcat.apache.org/tomcat-6.0-doc/aio.html, 2006.

[6] Jean-Francois Arcand. Glassfish project - webtier home page. https://glassfish.dev.java.net/javaee5/webtier/webtierhome.html#HTTP_Connector, 2005.

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

[8] Jean-Francois Arcand. Implementing asynchronous web application using grizzly's comet. http://www.glassfishwiki.org/gfwiki/Wiki.jsp?page=Presentations, 2006.

[9] Opera Software ASA. Opera 9 streaming support demo. http://oxzone.opera.com/webchat/.

[10] Asyncweb. http://docs.safehaus.org/display/ASYNCWEB/Home, 2006.

[11] Backbase enterprise ajax framework. http://www.backbase.com/, 2007.

[12] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[13] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – http/1.0, 1996.

[14] T. Bernes-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). Internet Request for Comment RFC 1738, Internet Engineering Task Force, December 1994.

[15] Manish Bhide, Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.

[16] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[17] Steve Champeon. Javascript: How did we get here? `http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html`, 2001.

[18] Tom Christiansen and Nathan Torkington. *Perl cookbook*. O'Reilly & Associates, Inc., 1998.

[19] Cometd. Cometd,the scalable comet framework. `http://www.cometd.com/`.

[20] Mortbay Consulting. Jetty 6 architecture. `http://docs.codehaus.org/display/JETTY/Architecture`, 2006.

[21] Mortbay Consulting. Jetty webserver documentation - continuations. `http://docs.codehaus.org/display/JETTY/Continuations`, 2006.

[22] D. Crockford. Json (javascript object notation). `http://www.json.org/`, 2006.

[23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999.

[25] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[26] R. Finlayson. Apache mpm event. `http://httpd.apache.org/docs/2.2/mod/event.html`.

[27] R. Finlayson. Ip multicast and firewalls. `http://www.ietf.org/rfc/rfc2588.txt`, 1999.

[28] Simon Forge. Pushlets whitepaper. `http://www.pushlets.com/doc/whitepaper-all.html`, 2006.

[29] Apache Software Foundation. Http server project. `http://www.apache.org`.

[30] Michael Franklin and Stan Zdonik. ”data in your face”: push technology in perspective. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 516–519. ACM Press, 1998.

[31] Jesse Garrett. Ajax: A new approach to web applications. Adaptive Path, 2005.

[32] Manfred Hauswirth and Mehdi Jazayeri. A component and communication model for push systems. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag / ACM Press, 1999.

[33] Ibm websphere application server. `http://www.ibm.com/software/webservers/appserv/was/`, 2007.

[34] Icesoft technologies - icefaces 1.5.2 developers guide. `http://www.icesoft.com`, 2006.

[35] Icesoft technologies - icefaces 1.5.2 enterprise production suite developers guide. `http://www.icesoft.com`, 2006.

[36] Internet explorer 7 technology. `http://www.microsoft.com/windows/ie/ie7/technology/`, 2007.

[37] Kanaka Juvva and Raj Rajkumar. A real-time push-pull communications model for distributed real-time and multimedia systems. Technical Report CMU-CS-99-107, School of Computer Science, Carnegie Mellon University, January 1999.

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

[39] Rohit Khare and Richard N. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.

[40] Twisted Matrix Labs. Twisted documentation. `http://twistedmatrix.com/trac/wiki/Documentation`.

[41] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[42] Jonathan Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153. USENIX Association, 2001.

[43] Benedetto Lepori, Lorenzo Cantoni, and Riccardo Mazza. Push communication services: A short history, a concrete experience and some critical reflections. *SCOMS - Studies in Communication Sciences*, 2(1):149–164, 2001.

[44] Lighstreamer. Lightstreamer - true push streaming. `http://www.lightstreamer.com/technology.htm`.

[45] Macromedia. Macromedia flash - actionscript dictionary: Xmlsocket (object) - http://www.adobe.com.

[46] Jean-Philippe Martin-Flatin. The push model in web-based network management. `http://www.labunix.uqam.ca/~jpmf/papers/index.html`.

[47] Steve McDonald and Jay McDonald. Fjax. `www.fjax.net`.

[48] A. Mesbah and A. van Deursen. An architectural style for ajax. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53. IEEE Computer Society, 2007.

[49] Ali Mesbah, Kees Broenink, and Arie van Deursen. Spiar: An architectural style for single page internet applications. Technical report, Centrum voor Wiskunde en Informatica (CWI), 2006.

[50] Mozilla support. `http://www.mozilla.org/support/`, 2007.

[51] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.

[52] Donovan Preston. Livepage. `http://twistedmatrix.com/projects/web/documentation/howto/livepage.html`, 2006.

[53] S. Rao, H. Vin, and A. Tarafdar. Comparative evaluation of server-push and client-pull architectures for multimedia servers. Technical report, Department of Computer Sciences, University of Texas, Austin, 1996.

[54] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 344–360. Springer-Verlag New York, Inc., 1997.

[55] Frederic Rossi. An event mechanism for linux. *Linux J.*, 2003(111):7, 2003.

[56] Alex Russell, Greg Wilkins, and David Davis. Bayeux - a json protocol for publish/subscribe event delivery protocol 0.1draft3. `http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html`, 2007.

[57] Douglas C. Schmidt. Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching. *Pattern languages of program design*, pages 529–545, 1995.

[58] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 60. IEEE Computer Society, 1998.

[59] Sun microsystems - java virtual machine (jvm) specification. `http://java.sun.com/docs/books/vmspec/`.

[60] Sun microsystems - kjava message service (jms). http://java.sun.com/products/jms/.

[61] Sun microsystems - remote method invocation home. http://java.sun.com/javase/technologies/core/basic/rmi/.

[62] Sun microsystems java nio package. http://java.sun.com/j2se/1.4.2/docs/guide/nio/.

[63] Sun microsystems - abstract window toolkit. http://java.sun.com/products/jdk/awt/, 1999.

[64] Sun microsystems - java runtime environment (jre). http://www.java.com, 2004.

[65] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems: principles and paradigms*. Prentice Hall, 2002.

[66] Ice Technology. ipush server - benchmark reports. http://www.icetechnology.com/products/ipushv2/benchmark.shtml.

[67] Alex Russell Dojo Toolkit. Comet: Low latency data for the browser. http://alex.dojotoolkit.org/?p=545.

[68] Vittorio Trecordi and Giacomo Verticale. An architecture for effective push/pull web surfing. In *ICC (2)*, pages 1159–1163, 2000.

[69] W3C. Http 1.1 spec, section 8.1.4. http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html.

[70] W3C. The xmlhttprequest object. www.w3.org/TR/XMLHttpRequest, 2006.

[71] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[72] Matt Welsh and David E. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[73] WHAT-WG. Web applications 1.0 working draft. http://whatwg.org/specs/web-apps/current-work/\#scs-server-sent, December 2006.

[74] World Wide Web Consortium. *HTML 4.01 Specification*, December 1999.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AJAX: Asynchronous Javascript And XML

CSS: Cascading Style Sheets

DHTML: Dynamic Hyper Text Markup Language

DOM: Document Object Model

FSM: Finite State Machine

JRE: Java Runtime Environment

JVM: Java Virtual Machine

MPI: Multi Page Interface

PaP: Push and Pull

PoP: Push or Pull

REST: Representational State Transfer

RIA: Rich Internet Applications

RMI: Remote Method Invocation

SEDA: Staged Event Driven Architecture

SPI: Single Page Interface

TTR: Time to Refresh

URL: Uniform Resource Locator

W3C: The World Wide Web Consortium