Improving the model view controller paradigm in the web

Santiago Robles and
Juan Lautaro Fernández

LIFIA, Facultad de Informática,
UNLP,
Calle 50 y 115, La Plata,
Buenos Aires, CP 1900, Argentina
E-mail: srobles@lifia.info.unlp.edu.ar
E-mail: lfernandez@lifia.info.unlp.edu.ar

Andrés Fortier

CONICET and LIFIA, Facultad de Informática,
UNLP,
Calle 50 y 115, La Plata,
Buenos Aires, CP 1900, Argentina
E-mail: andres@lifia.info.unlp.edu.ar

Gustavo Rossi*

CONICET and LIFIA, Facultad de Informática,
UNLP,
Calle 50 y 115, La Plata,
Buenos Aires, CP 1900, Argentina
E-mail: gustavo@lifia.info.unlp.edu.ar
*Corresponding author

Silvia Gordillo

CIC and LIFIA, Facultad de Informática,
UNLP,
Calle 50 y 115, La Plata,
Buenos Aires, CP 1900, Argentina
E-mail: gordillo@lifia.info.unlp.edu.ar

Abstract: Nowadays, web applications have changed from static, information-based web pages to dynamic, service-oriented software. As they evolved, these web applications started to show features that used to appear only in desktops applications. We have reached a point in connection speed and browser maturity where there is only one thing preventing the next big step: the ability of web servers to send information to the client without a previous request. In this paper, we present Meteoroid, a framework that allows a system running in the server to send information to web browsers without a client request. On top of this functionality we have built a full model-view-controller protocol, allowing us to create web applications that completely mimic desktop ones.

Copyright © 2012 Inderscience Enterprises Ltd.

Keywords: comet; MVC; web framework; Seaside; Smalltalk.

Reference to this paper should be made as follows: Robles, S., Fernández, J.L., Fortier, A., Rossi, G. and Gordillo, S. (2012) 'Improving the model view controller paradigm in the web', *Int. J. Web Engineering and Technology*, Vol. 7, No. 1, pp.22–44.

Biographical notes: Andrés Fortier is a PhD student at the Facultad de Informática, Universidad Nacional de La Plata, Argentina where he also received his degree. He obtained two CONICET grants for his PhD studies and has taught different courses in the UNLP. His research interests include context-aware application development, web engineering and object-oriented languages.

Santiago Robles has recently finished his undergraduate studies at the Facultad de Informática, Universidad Nacional de La Plata while researching at LIFIA with a CICPBA grant. He is mainly interested in web development and object-oriented technologies.

Juan Lautaro Fernández has recently finished his undergraduate studies at the Facultad de Informática, Universidad Nacional de La Plata while researching at LIFIA with a CICPBA grant. He works on dynamic web development and architectures design.

Gustavo Rossi is a Full Professor at the Facultad de Informática, Universidad Nacional de La Plata and a Researcher of CONICET, Argentina. He holds a PhD from PUC-Rio, Brazil. His current research interests include agile approaches in web engineering and context-aware software development.

Silvia Gordillo is a Full Professor at the Facultad de Informática, Universidad Nacional de La Plata and a Researcher of CICBA, Argentina. She holds a PhD from the Université Claude Bernard-Lyon I, France. Her current research interests include geographic information systems and mobile computing.

1 Introduction

In the early days, the web was mostly used as a big repository of arbitrary static data, usually managed by a user who was in charge of the site and even the web server. The static data we are referring to was encoded in the form of HTML documents that were later accessed by web browsers around the world. The mechanism used to retrieve the HTML pages was implemented over the HTTP (1999) protocol, which states (among other things) that an explicit request must be made to the server for every resource that the web browser needs. Each request is independent of the others and once it is completed there is no further connection between the client and the server. As a result, showing a complete web page involved a set of stateless HTTP requests, one for the page itself and one for each resource in it (e.g., an image, a video, etc.).

However, the web has evolved since those days and it is no longer a repository of static data, but a place where information is constantly changing and where the end users are capable of generating the contents themselves. Also, part of the web has gone beyond the idea of web pages to become full-fledged applications (Java Servlets, 1999), where

the user can perform almost the same tasks as he would on a ‘standard’ desktop application.

It is clear that, for this revolution to happen, new technologies were needed in different areas: server side programming (PHP, JSP, etc.), web development frameworks (RubyOnRails, Struts, Seaside, etc.), client side languages and toolkits (JavaScript, Ajax, etc.) and browser plugins (Java Applets, Flash, Silverlight, etc.) are just some of the engines that powered this transition. However, despite all these advances, one thing still remains intact in the HTTP protocol: in order to obtain a resource, the client has to issue an HTTP request, which is closed once it has been completed, leaving no binding between the client and the server. While this last remark was not an issue 10 or 15 years ago, we have reached a point where stateful connections between the server and the client can make a difference.

The issues related to stateless HTTP connections have been long perceived by developers and users and have been mitigated with different techniques. Maybe the most obvious one was the need to reload a whole page when only a small portion of it changed. This problem was solved with Ajax (Garret, 2005), a technique used to retrieve only parts of a page from the server and replace those changing sections of a page without replacing the page itself. This technique helped us avoiding sending the same data over and over the network and reducing page flickering, but at the expense of some problems of its own (e.g., properly handling the browser’s back button). However, there is still a void when it comes to pushing data from the server to the web browser when no explicit request has been made. Thus, if the server detects a change in its data, there is no way of notifying its clients that their view is outdated; instead, it is the responsibility of the client to somehow poll the server for changes, generally by sending periodic requests.

This last issue is of great importance if we want to implement a real model-view-controller (MVC) (Krasner and Pope, 1998) architecture for the web, since one of its core components [the Observer pattern (Gamma et al., 1995)] can not be fully realised.

As we previously mentioned, there are ways of mitigating this problem. Some applications combine JavaScript timers with Ajax, creating a sort of busy waiting technique: after being idle for X milliseconds a request is sent to the server to ask for changes. If new data is available, it is returned in the server’s response and the client page is updated. However, this technique has a big drawback: the estimation of the polling rate. If the update interval is too long, the updates are deferred and the information may be obsolete by the time it reaches the client. On the other hand, if the update interval is too short, it can overload the network with unnecessary requests. Also, in both cases the overhead of opening and closing a connection must be paid.

As a result, even though this technique is very popular nowadays, it is just a workaround to overcome the stateless nature of the HTTP protocol. As a matter of fact, this technique would never be used in a desktop environment, where the observer pattern and different flavours of MVC frameworks have long been the de-facto standard. What we are really missing is a technique to send events from the server to the client without an explicit request from the latter.

While it is still in the early stages, a new trend to solve this problem started some years ago. Comet (Crane and McCarthy, 2008), which name was coined by Russell (2006), has the purpose of providing a bidirectional connection between servers and clients, enabling the former to push new events to the latter whenever it wants. By having Comet capabilities in a server, we would be able to set up a new environment for rich

web applications, where the client could be updated on-time without relying on complicated and inefficient refresh mechanisms. By using Comet, web developers can concentrate on the visual representation of the model in the browser, not worrying about setting up timers in the client. As a matter of fact, Comet gives web applications the chance to be as powerful as desktop ones, because it solves a relevant aspect that is currently missing in web applications: the capability of notifying changes in its model to the views (browsers).

However, having this new feature has en important cost: most of the Comet techniques rely on having a socket constantly open between the server and the web browser. This of course has an impact on the sever side, where the amount of clients that can be handled concurrently for long periods of time is smaller than in stateless requests. However, much has been done in the last years regarding server performance; servers like Migratory claim to achieve 1 million concurrent users while others can easily support 20.000 to 30.000 concurrent clients[1] (Comet Maturity Guide, 2009; Migratory Benchmarks, 2010). Nevertheless, it should be clear that Comet should not be used in every website or web application. For example, there would be no benefit at all in adding Comet capabilities to a static site (like a simple institutional site or a file-sharing repository). Comet-based web applications are those that require instant client notification when something changes in the server. A few examples of this kind of applications are:

- notifications systems, like Facebook and Gmail
- instant-messaging, like Gtalk and Meebo
- cooperative systems, including e-learning, collaborative authoring and editing, telemedicine, call-centre support, gaming, etc.
- telemetry in different areas, like F1 simulations[2], biking events[3] or spacecraft data[4].

In this paper, we present *Meteoroid*, a Comet-based framework built on top of Seaside (Ducasse et al., 2010), a web application framework developed in Smalltalk. The aim of this framework is twofold:

- to provide a bidirectional channel between the client and the server, hiding the complexities of dealing with browser-specific details for setting up the connection
- build a set of live web-widgets on top of the bidirectional channel, to allow developers to create web applications that can respond in the same way desktop applications do.

For this purpose Meteoroid uses a strict MVC view of web applications, where the model role is played by the application model running in the server and the view-controller role is played by the browser and the web page displayed in the client. By having a communication channel between the server and the web browser, Meteoroid can send events to update the view according to the changes in the model, just as any standard desktop MVC-based application would do. On top of that a set of web-widgets are constructed to ease the building process of live web applications.

The paper is structured as follows: in Section 2 a set of Comet frameworks are described, comparing their advantages and pitfalls. Later on, in Section 3, the Meteoroid framework is explained in detail showing how we factored it in three layers. For each layer we explain the rationale behind its design and an example of its use. As a final

example of this paper, in Section 4, we present an online auction application (which is one of the paradigmatic Comet examples). In Section 5, we conclude and discuss some further work we are pursuing. As a final note, we won't be discussing security issues related to Comet since that exceeds the scope of the paper [see for example WebSockets Disabled (2010)].

2 Related work

Although we have been referring to Comet as a technique, it is actually an umbrella for several connection techniques to enhance the HTTP protocol. Its main objective is to empower the web server with the capability of pushing data to the client, while choosing the connection technique that best fits each browser (since Comet is not a standard yet, each web browser has to use a specific connection type).

The requirements for setting up a Comet connection pose no big challenges: the web server should support streaming connections (Mahemoff, 2006), while the web browser should have JavaScript enabled. With this basic setup, Comet uses as much standards as possible while avoiding proprietary plugins like Flash or SilverLight. Those plugins are replaced by a browser-specific type of connection, taking the advantages of each browser implementation to exploit the use of the HTTP connection.

While the idea behind Comet is quite simple (sending events from the server to the browser), once a Comet connection has been established many things can be built on top of it (e.g., event-based notification systems, high-level communication channels, live web widgets, etc.). For this reason there are several frameworks and toolkits that offer different approaches towards building real web MVC applications. Each approach focuses on different aspects, like using the best Comet connection technique, providing a high level of abstraction to the developers, optimising the amount of users that are concurrently connected to the applications, etc.

In this section, we describe a group of frameworks that use Comet, highlighting their strengths and explaining their disadvantages. We chose these frameworks and toolkits since they are paradigmatic in some sense (e.g., IceFaces because of the high-level widgets, LightStreamer since it was one of the pioneers, etc.). As a result other servers exposing the same features had to be left out.

2.1 Ajax push engine (APE)

APE (2010) is an open source framework to create Comet applications, focusing on building systems that allow many concurrent users. This framework is divided in two components, the *APE server* and the client-side *APE JavaScript framework*.

The APE server is in charge of creating and managing Comet connections, choosing the best connection technique depending of the web browser. It was developed in C with speed in mind and it consumes few resources. The server uses a channel abstraction for its communication with the application model, which can be developed in any language.

The APE JavaScript framework, which executes in the web browser, provides an abstraction layer that helps to process the information that comes from the APE server. It basically exports an interface to send and receive events from and to the server.

The APE framework is open source and it does not need any plugins in the client-side. It chooses the best connection technique for the browser and it handles many users running concurrently. However, the framework does not provide any abstraction beyond the channel metaphor. The developer has to code JavaScript to update HTML fragments or form components when an event is triggered in the server. In other words, APE handles the low level details, but lacks support for higher abstraction elements.

2.2 IceFaces

IceFaces (2010) is an open source framework that allows creating Comet-like applications written in Java. It is 'Comet-like' and not exactly Comet because to build the applications IceFaces uses the long polling (Crane and McCarthy, 2008) connection technique which, depending on the application type, may not be as good as other Comet techniques. IceFaces was developed on top of the JavaServer Faces (2010) framework, which is used to create web applications in a very simple way.

This framework has a set of high level components, making it simple to update an HTML page. This is a really interesting feature, because it allows the developer to focus on its own problems and lets the framework handle the HTML updates and the JavaScript codification. As with APE, this framework does not require any plugins on the client side. The main drawback that we found in IceFaces was the use of a long polling connection that has some disadvantages as described by Crane and McCarthy (2008).

2.3 LightStreamer

LightStreamer (2004) is a Comet framework that allows the web developer to build a Comet application without requiring a plugin installed on the browser. This framework has some interesting features like bandwidth control (limiting the bandwidth assigned to each client to a specified value) or limiting bandwidth usage to avoid network congestion. As other frameworks, it chooses the best connection technique and provides a JavaScript API to simplify the data processing. Despite having this JavaScript API, the abstraction level is not as high as it could be, it has no HTML tags that can be automatically updated and the developer must explicitly code JavaScript to modify page fragments.

2.4 Meteor

Meteor (2010) is an open source framework, built in Perl, which provides a very simple Comet connection. It is composed of a *Meteor server* in the server-side and a *JavaScript API* in the client-side.

The Meteor server works using channels where the clients can subscribe to receive information or push new data for the others clients. It is also in charge of choosing the best Comet connection according to of the client browser and managing all the channels that the users subscribed to.

As others described frameworks, it has some nice features like being an open source project and choosing the best Comet connection, but the JavaScript API in the client-side is too simple, and does not provide an easy way of programming the updates, leaving all the responsibility to the application developer.

2.5 Orbited

Orbited (2008) is an open source Comet framework, written in Python, that provides a persistent connection to send information from the server to the clients that are interested. It is composed of a *daemon* and a *JavaScript API*.

On the server-side, the daemon receives information through message queues and updates all the clients that are interested in a specific event. On the client-side, the JavaScript API is in charge of opening the Comet connection using the best technique depending on the web browser. The framework is open source and, like APE, it can work with any language that has a queue implementation. However, the framework does not provide a set of high level abstractions to perform page updates. The JavaScript API provides a set of messages to use when an event is triggered, but the updates must be coded in JavaScript and there are neither predefined updates nor updatable HTML tags.

2.6 Pushlets

Pushlets (2007) is a framework built in Java conceived to be combined with Servlets to create Comet applications. Through a single class (Pushlet), all the clients are subscribed to events and whenever an event occurs, all the interested clients are notified.

Unlike others frameworks that choose the best connection technique, Pushlets only uses Forever IFrame (Russell, 2006), which is a generic technique that works in almost every browser but has some visual problems (Fernández et al., 2009). Pushlets has a JavaScript API in the web browser that simplifies a bit the processing of new data that arrives from the server, but is not as complete as it could be, forcing the developer to program the JavaScript updates.

2.7 Discussion

All the frameworks described in this section have similarities and differences, each one focusing its effort in different kind of problems to solve. We next summarise why we have chosen the following criteria to evaluate the frameworks:

1. Open source: Since there is no widely accepted standard when it comes to Comet, working with open source libraries can make a huge difference when it comes to debugging and adding new functionality.
2. Requires external web server: Having a framework integrated with a web server eases the configuration part of the development. On the other hand it may be restricting for legacy systems.
3. JavaScript API: If no JavaScript API is provided the programmer must deal with low-level details.
4. Streaming connections: A restriction to consider if the framework requires an external web server.
5. Choose best Comet technique: As we will show in the next section, not choosing the best Comet technique for each individual browser may cause glitches and even confuse the user.

6 Updatable HTML tags: We use this to differentiate those frameworks that only provide the communication channel from those that add one or more layers on top of it to easily build graphic interfaces.

7 Needs plugin: Whether or not we must install third-party software on the client for the web application to run properly.

Among the interesting features of the reviewed frameworks are the updateable HTML tags, provided by IceFaces, which lets the user define tags that are automatically updated by the framework as a result of a server event. This is really interesting because it moves the event processing and DOM updating burden to the framework, releasing the developer from this tedious task and allowing him to focus in the application domain problem. Most of the other frameworks only provide a JavaScript API that handles some kind of update channel where the developer has to manage the event processing and page update. On the bright side, all the frameworks presented here do not need external plugins, which makes them compatible with former and new web browsers. While this is the most common approach, other servers (e.g., juggernaut) use third-party plugins like Flash to achieve the bidirectional connection.

Table 1 Framework comparison

	APE	*IceFaces*	*LightStreamer*	*Meteor*	*Orbited*	*Pushlets*
Open source	Yes	Yes	No	Yes	Yes	Yes
Requires external web server	Yes	No	Yes	Yes	Yes	No
JavaScript API	Yes	Yes	Yes	Yes	Yes	Yes
Streaming connection	Yes	No	Yes	Yes	Yes	Yes
Choose best Comet technique	Yes	No	Yes	Yes	Yes	No
Updateable HTML tags	No	Yes	No	No	No	No
Needs plugin	No	No	No	No	No	No

As a final remark we should mention that in the future, those frameworks that only take care of handling the Comet connection are likely to disappear due to the implementation of new standards. Currently, the W3C is working on two standards that allow the creation of Comet connections. As a result Server-Sent Events (2009) and WebSockets (2009) were born with the aim of encompassing the different Comet techniques and defining a cross-browser standard for Comet. In the specification, the server-client connection is standardised and a format is given to push data from the server to the client.

Despite being recently-developed standards they are already implemented on several popular browsers like Chrome, Safari, Opera and the beta version of Firefox. However, due to recently found security issues, browsers such as Firefox 4.0 have disabled WebSockets (WebSockets Disabled, 2010) until the protocol vulnerabilities can be fixed.

3 Our approach to comet

In order to solve the previously mentioned problems, we decided to build a new framework, to allow the developer to easily create full fledged MVC applications. The aim of the framework (which we called Meteoroid) was to isolate the developer as much as possible from the low-level details (e.g., choosing the best Comet connection technique) and to provide a set of high-level widgets that would be automatically updated according to the server changes. We decided to build Meteoroid using Comet since it defines a variety of connection techniques, which in our implementation are completely encapsulated. Among the connection strategies are both WebSockets and Server-Sent Events, both of them being part of the html five drafts. As a result our framework can seamlessly transition the different stages until server-client communication is standardised across browsers. Also, Comet can be achieved without installing any third-party products in the client (e.g., Flash), making its use straightforward.

Figure 1 Meteoroid layers

Meteoroid was developed as a three-layered framework, with each layer providing new functionality on top of the previous one. The base layer, called PushScript, provides a raw Comet connection that is created using the best connection technique according to the client web browser. At the application programming interface (API) level it only provides a single function (*PushScript*, hence the name of the layer) which allows the developer to send arbitrary JavaScript code, through the Comet connection, to be executed in the web browser. The second layer (Observer) sits on top of the previous one, providing a higher level of abstraction by letting the developer configure updates based on events and coded in Smalltalk instead of JavaScript. The upper layer, called WebLiveWidgets, implements a set of live widgets (input fields, lists, etc.) that are bounded to an object’s aspect (e.g., a property) and are automatically updated whenever that aspect changes. In Figure 1, we summarise the framework layers.

3.1 PushScript layer

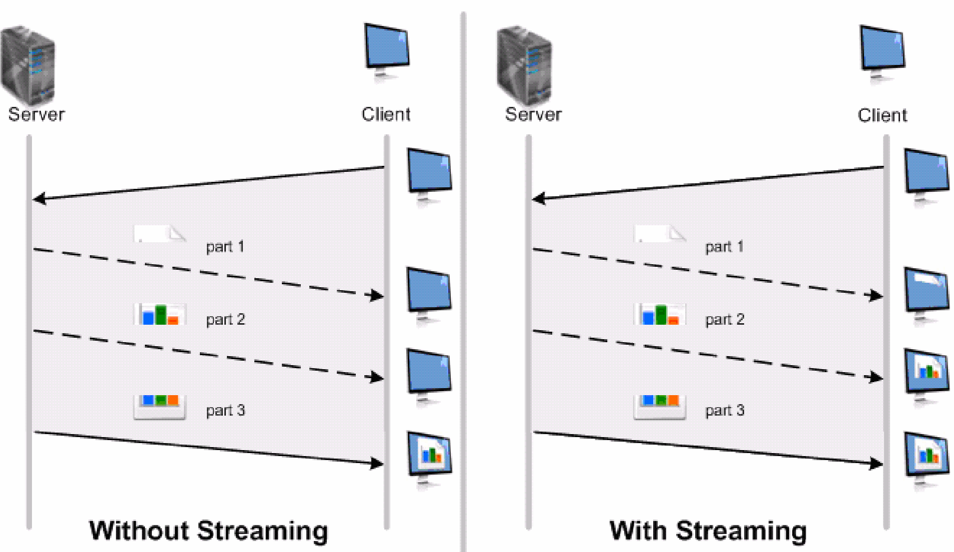
As explained before, this first layer is the core of Meteoroid and it implement the lowest abstraction level facilities. It is responsible of creating the Comet connection that best suits the web browser making the request. The Comet connection is then established and kept alive by using the underlying web server streaming capabilities. On top of that the

layer provides a very simple API to allow the web server to communicate with its views on the web browsers. This API basically lets the programmer send an arbitrary piece of JavaScript code from the server to be executed in the client. We next explain the most important features of this layer.

3.1.1 Streaming capabilities and Comet implementation

In the early days web servers created a response for each request they served, only sending it back to the client once it was completely build. This approach presented two important problems: 'heavy' resources (like big images) would give the user the feeling of slow page loading and real-time content could not be delivered (e.g., a live broadcasting). To solve these issues streaming was introduced, allowing servers to send the response to the client in chunks. Thus, a page can be incrementally rendered as the contents are moved from the server to the client, enabling for example to broadcast a live concert almost on real time. The above example is explained graphically in Figure 2, where a client requests a page with a big image. In the normal request (without streaming) the page is rendered after the whole content is in the client, while in the second scenario the page will render the image as its chunks are delivered.

Figure 2 Page loading with and without streaming (see online version for colours)



In the Comet case, the streaming capabilities can be exploited to implement server-client communication. The trick is done by forcing the server to keep the response always open (i.e., never closing the response) and use that response as a communication channel. When new data has to be sent to the client, it is encoded and appended to the open response. The client can in turn incrementally process the response with a JavaScript library that runs in the browser, updating its contents as needed. The later approach is the one used in Meteoroid, where a connection is kept alive between the server and the client.

3.1.2 Creating the connection

Any browser can establish a connection with a streaming server by requesting a URL with an 'endless' response. However, depending on the browser this may cause some glitches, like giving the user visual cues that the page has not finished loading. Examples of those cues are shown in Figure 3.

Figure 3 (A) Waiting cursor (B) looping throbber (C) status bar (see online version for colours)

The 'standard' way of creating a Comet connection is by using a technique called Forever IFrame, which has the advantage of working in any browser but the disadvantage of displaying misleading feedback to the clients (e.g., the browser will always indicate that the page is still loading). Others techniques solve these visual issues by using different means to establish the Comet connection; however these techniques are browser-dependent:

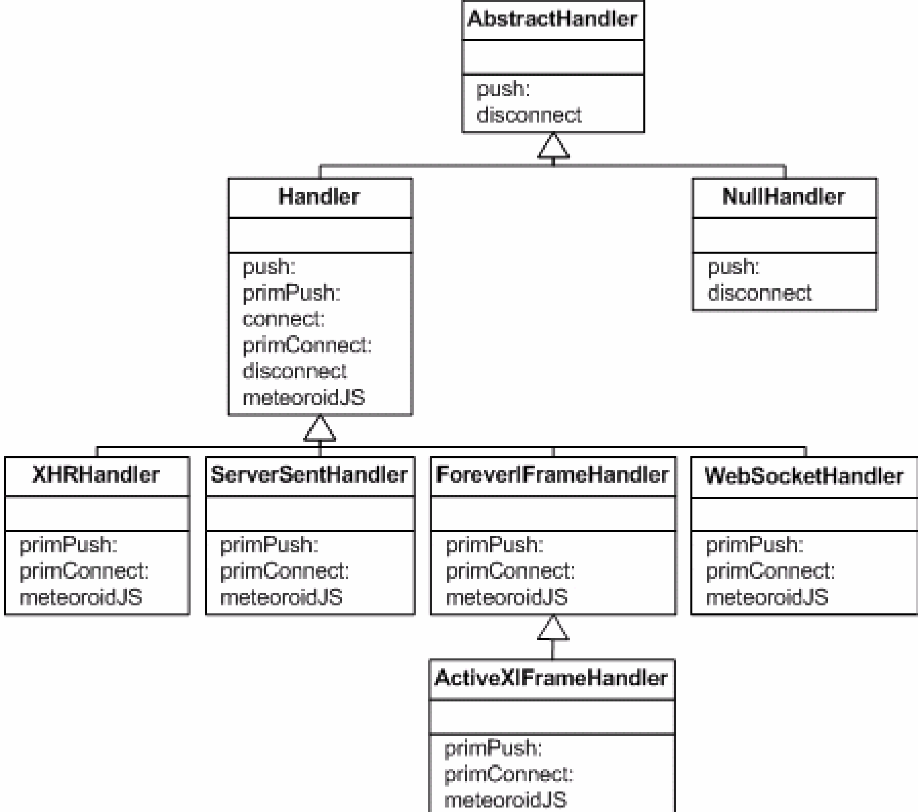
- *XMLHttpRequest (2010):* The connection is created by means of a XMLHttpReqest on the client. The browser uses the XHR interactive state to parse the upcoming data which came from the server in the browser. It is mostly used on Mozilla-based engines.
- *Server-Sent Events (2009):* Allows sending events from the server to the client. This was conceived as a first approach for later inclusion in the HTML5 standard. It is implemented by Opera, Safari 5 and beta versions of Firefox 4 and Chrome 6.
- *IFrame and ActiveX (2009):* The ActiveX object creates an in-memory page, which has an IFrame that dynamically loads the page. It only works in Internet Explorer.
- *WebSockets (2009):* A bidirectional channel is created by the browser, which can be later used to send and receive data from both sides. Available in new HTML5-compliant web browsers, like Chrome, Firefox beta 4 and Safari 5.

A summary of the different techniques and their application according to the web browser is shown in Figure 4. The advantage of using those techniques marked with green circles over the greyed ones is that the visual glitches are avoided.

Figure 4 Summary of connection techniques (see online version for colours)

	Forever IFrame	XML HttpRequest	Server-Sent Events	ActiveX + IFrame	WebSockets
	●	●	β		β
	●		●		●
	●			●	
(others)	●				

Figure 5 Class diagram of the handler hierarchy

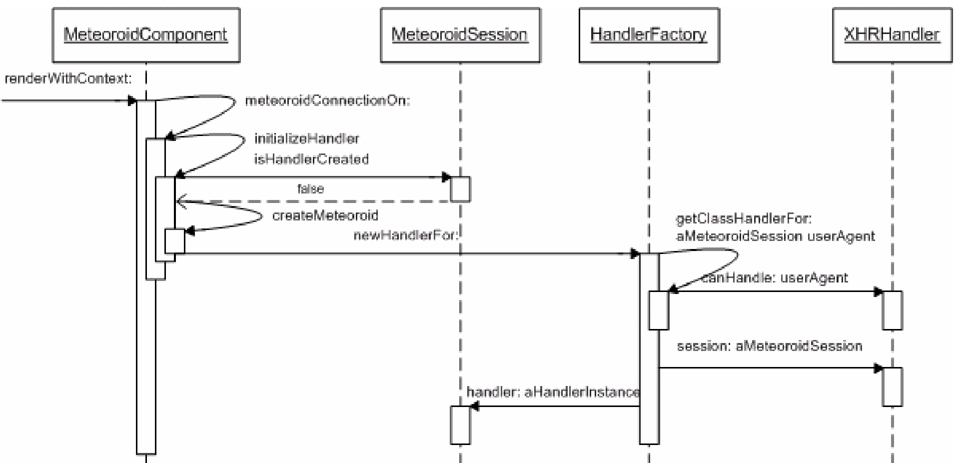


All the previous techniques were implemented in Meteoroid by means of handlers. Each handler encapsulates a specific connection technique, thus making it easy to accommodate new techniques or improving existing ones. A class diagram depicting the Handler hierarchy is shown in Figure 5.

When a browser requests a connection, the framework chooses the handler that best fits the client by using a HandlerFactory, which checks for the user agent. It is then the responsibility of the handler to implement the push functionality according to the

concrete connection. An interaction diagram depicting the creation of the appropriate handler and its binding to the session is shown in Figure 6.

Figure 6 Interaction diagram depicting the handler creation



Thanks to this design, we keep the connection mechanisms encapsulated in highly cohesive objects. If a new connection technique appears (or an old one changes), the handler will be modified to stay up-to-date without affecting the rest of the framework. As a matter of fact, while developing this layer the WebSocket protocol was in the process of being incorporated to Firefox and adding support for it was almost straightforward.

3.1.3 Integration with seaside

Meteoroid is not a stand-alone server, but an extension to the Seaside web framework (Ducasse et al., 2010). Seaside is a framework for developing web applications completely written in Smalltalk, which uses components as a way to structure web pages. Seaside components are basically web views used to display information or generate forms. If components are properly designed, they can be composed and further reused in different applications; for example a search component that shows a list and an input field to filter the list contents can be embedded as a child of other component and thus reused in many applications. As an example consider a simple web chat with three components:

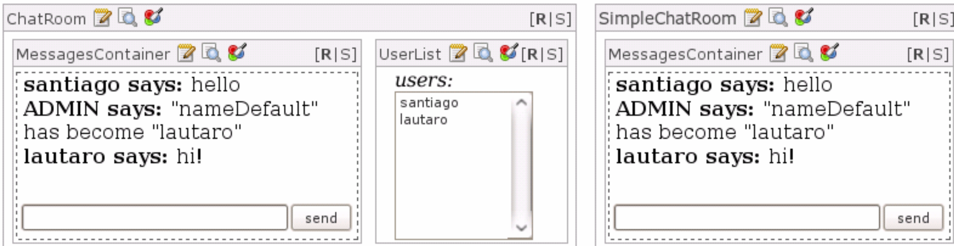
1 a component to render the users list, called UserList

2 a component with a text area where the comments are shown and an input to send text, called MessagesContainer

3 a root component used to integrate the previous two, called ChatRoom.

By using this design the chat components can be reused in different ways, for example in a simpler chat (called SimpleChatRoom), without the users list (see Figure 7).

Figure 7 On the left side a chat made of three components and on the right other made of two (see online version for colours)



In Seaside, the base philosophy is to develop web applications without writing HTML tags or directly coding JavaScript; instead Seaside provides an object-oriented framework that generates XHTML based on the implementation of the rendering method on each component. Meteoroid runs on top of Seaside, allowing components to send arbitrary JavaScript code from the server to the browser. From the developer's point of view, the only difference is that instead of subclassing from WAComponent (the base class for all Seaside components), he must do so from Meteoroid (which is, in turn, subclass of WAComponent). By doing that the developer has now access to the *#pushScript:* message, and can asynchronously send JavaScript code to be executed in the browser. As an example consider popping up an alert in the web browser as a result of an event in the server:

```
NewComponent>>sendAlert
    self pushScript: 'alert("Test");'
```

With the *#pushScript:* added functionality we can now develop a real-time online chat application by broadcasting to the clients each new message that gets posted. The update of the client web pages is achieved by manipulating the DOM tree through JavaScript (Snook et al., 2007) and adding the text broadcasted by the server.

Summing up, the PushScript layer provides a transparent way for server-client communication by means of a single message (*#pushScript:*). Thanks to the handler hierarchy, the developer doesn't have to worry about the low-level details of the communication channel.

3.2 Observer layer

Despite its benefits, the PushScript layer has two main drawbacks: it forces the programmer to write JavaScript code (instead of Smalltalk) and leaves the burden of synchronising the model with the views to the developer.

The first problem can be solved by using a port of the JavaScript library script.aculo.us (Crane et al., 2007) for Seaside, which allows the developer to code the entire web application in Smalltalk. As a result the framework generates the final JavaScript, helping with encoding issues.

The second problem is addressed in the Meteoroid Observer layer by setting up an observer mechanism. The Observer pattern is widely used in desktop applications because it allows the model to be decoupled from the view. To do so, a subject has a list of interested objects (called observers) that will be notified of any change of the subject's

state. This notification is generally passed in the form of an event with additional parameters if necessary. An example of such an implementation is VisualWorks' Announcements framework. In this framework the events are represented as objects, with its own state and behaviour. Representing the events as objects allows implementing applications easier than using simpler Observer implementations (for example, one that uses strings to represent the events) because the developer can delegate behaviour to the announcement itself.

By combining the JavaScript abstraction with the Announcements implementation, we created the Observer layer. The idea of this layer is to extend the basic API provided by the lower layer, allowing the developers to define updates based on events triggered in the server. The dependencies and the generation of JavaScript code is completely handled by the framework, which means that the developer only has to associate an event with the rendering code he wants to execute when that event happens. As a result we can express the program behaviour in a higher level, stating things like "when the event X is triggered in the model, update the Y component in the web page". To this end the Observer layer provides three different types of functionality, each one with a set of helper functions to configure how to react to an announcement:

- *Updates:* Through the update messages different DOM elements can be modified, replacing them with new ones that are created dynamically when an event occurs. This can be useful for showing status to the user, like updating his cart representation while he shops.
- *Insertions:* Given a DOM element, the insertion messages allow inserting new information in predefined positions (before, at top, at bottom or after the element) when an event is announced. A typical case requiring this functionality is the chat example, where each time a new message arrives a line is added at the bottom of the list.
- *Scripting:* The previous categories are very useful to generate new DOM elements, replacing or inserting them into old ones. However, sometimes web developers use different libraries to create widgets that are not provided by the standard HTML specification. In these cases, a way to execute new JavaScript code is needed, and for that reason a protocol to send JavaScript was implemented.

An important feature of this layer is that the developer only has to define the association between the events and the actions (by using the previously explained messages) and the framework will manage the dependencies needed to execute the defined action. These dependencies not only contemplate the trivial cases, like entering in a page or closing the window, but also the complex ones, like using back and forward buttons, multiple page refresh, closing tabs and reopening them, navigating to another page, etc.

To complete this section, we now show how the chat example can be improved by using the Observer layer. In this case the chat view is registered as an observer of the model with triggers an announcement each time a new message arrives. As a consequence the view generates an update script and pushes it to all the web clients (see Figure 8).

Figure 8 A simplified example of dispatching a new chat message (see online version for colours)

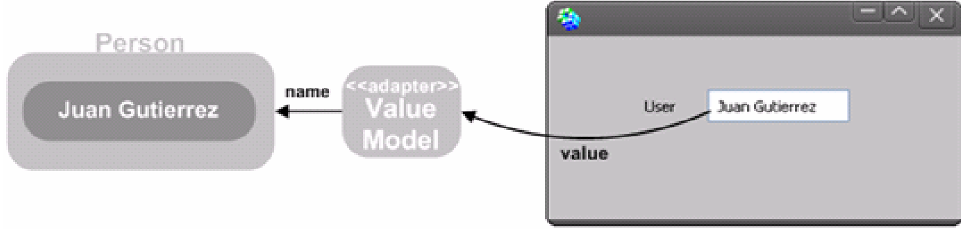
As a final remark we must note that the Observer layer is in charge of managing the dependencies between the model and the views, letting the developer focus on the domain application problems.

3.3 Web live widgets layer

While the Observer layer provides an improvement over the simple PushScript API, managing component updates is still cumbersome. Consider for example a view with three textboxes; using the Observer layer we should define three update actions that are exactly the same, except for the object which provides the text boxes information. To cope with this type of situations we developed a new layer which uses the concept of live widgets, which are automatically synchronised with the server.

In order to understand the live widgets implementation we must briefly review the desktop GUI widgets in VisualWorks Smalltalk. These widgets rely on an underlying framework called ValueModels (Coplien and Schmidt, 1995) which is basically a collection of adapters (Gamma et al., 1995) used to provide a generic way to access the values of the different widgets. A widget can easily retrieve or modify an aspect of an object without caring about the concrete getter or setter used. As an extra feature the value model sends a notification, by means of the Observer pattern, each time an aspect changes. Following this philosophy, our live widgets were implemented considering the access to the data, independently of the real underlying model (see Figure 9).

Figure 9 Adapter example (see online version for colours)



The standard Seaside tag hierarchy provides regular HTML tags, which means that once they are rendered it is not possible to update them unless an explicit request from the client is made. We implemented a new set of widgets (live tags) by combining the static tags with the ability to send updates form the server to the client. As a result, these tags are like the normal HTML widgets, but have an associated value model. Thus, when the widget receives a notification that its model has changed, a JavaScript code is generated by the framework and sent to the client to update the tag. In the current implementation, we have built the live counterparts of the div, span, text input, text area, select, ordered collection, unordered collection, radio button, check box, table and image tags and also added custom div lists and progress bar widgets, giving a very complete variety of live components.

Comparing this new layer with the previous one, now the dependencies declaration and the update actions are no longer developer's responsibilities but the framework's. To illustrate the benefits of the live widgets, consider using the Observer layer to update the contents of a list: in this situation at least three updates declarations are needed: one for adding items, another for removing items and a third to change the selection. By using a live list tag, all this functionality is encapsulated, simplifying its creation, usage and maintenance. This not only is a benefit in terms of less lines of code to achieve the same result, but a conceptual one: in the previous layer we had to program the framework to react in a given way to an event. We are now providing a higher level of abstraction, by just telling the framework to synchronise an aspect of the model with a component in a web page. This approach can be used to implement an improved chat example, where we just associate live tags with the desired aspect's of the model. Once this association is done the framework takes care of updating all the clients. In Figure 10, we depict this situation.

Figure 10 Using live web widgets to implement the chat example (see online version for colours)

3.4 Features review

In Section 2.7, we summarised in a table the different aspects that we considered relevant for a Comet-based framework. We now revise each feature and see how it applies to Meteoroid:

- *Open source:* Yes, our framework is dual licensed under GNU GPL and Creative Commons, Attribution Non-Commercial-Share Alike.
- *Requires external Web Server:* Yes, and also runs on top of an existing web framework. Meteoroid can run on top of any server that can run Seaside and that supports streaming.
- *JavaScript API:* Yes, a client JavaScript library is automatically injected into the generated web page.
- *Streaming connection:* Yes, provided by the web server.
- *Choose best Comet technique:* Yes, including the latest WebSocket implementation.
- *Updateable HTML tags:* Yes, a full set of widgets are provided.
- *Needs plugin:* No, the framework uses only standard JavaScript.

4 A running example

Live auctions are currently one of the paradigmatic cases of web applications that use Comet. These web auctions mimic real auctions in the sense that are open for a short time and that the bidding process takes place in real time. In this kind of online auctions the user initially buys 'virtual' bids which can be later used to make real bids on a concrete auction. When the auction starts, the closing time of the process is relatively small (e.g., a couple of hours) and users who enter the auction can use the pre-paid virtual bids to make a real bid for the item. If a new bid is placed when less than ten seconds are left to close the auction the closing time is automatically extended, letting other users perform a new bid. Sites like Swoopo (http://www.swoopo.com), StuffBuff (http://www.stuffbuff.com), DeACentavos (http://www.deacentavos.com), etc., implement this kind of auctions and have become a success in very short time.

Live auctions are an example of web applications that range from moments of high activity (especially in the last seconds before closing a bid) to moments of calm (where there are no new bids). In those moments of high activity the user will be receiving a big amount of events coming from the server, while in the calm times no notification will be sent. For this reason, it is almost impossible to implement such an application using a long-polling technique without flooding the network with useless requests.

To show the simplicity and flexibility of our framework an Auction House Web application was developed. In our implementation, we built an administrator role (that can create new users, new auctions and schedule them) and a standard user role (that can buy bids, join an auction and bid for a product). The different views developed in this application were built using live widgets, thus the programmer only had to worry about creating a domain model for the auction house and setting up the links between the model and the web widgets.

When the user enters the web application, he access to the Multiple Auctions Management component, which is in charge of showing all the auctions he has joined in. By clicking on the tabs at the top of the page the user can switch between the different auctions he has entered. As can be seeing on Figure 11, the bottom of the page displays a bid graph that shows the bid price evolution of each auction.

Figure 11 Live auctions application (see online version for colours)

The bid graph was implemented using the 'Annotated Time Line' chart provided by Google Charts. Since the Google chart was not part of our standard live widgets, the component was built using the Observer layer to insert the necessary JavaScript to keep the graph updated. We next show an excerpt of the code required to build the Auction House application:

```
MultipleAuctionsManagement>> initialise: aUser size: aPoint
        super initialise.
        …
        self auctions do: [:auction |
                (1) self on: BidAddedAnnouncement
                        of: auction
                        javascriptSending: #addNewBidFrom:.
                (2) self on: AuctionClosedAnnouncement
                        of: auction
                        javascriptCallback: [:announcement :announcer |
                                        self auctionClosedScript: announcer]
                        ]
```

In the initialisation method we perform two event declarations for every auction. The first one [marked with (1)] states that every time the auction notifies about a new bid, the JavaScript returned by the *#addNewBidForm:* message will be sent to the browser. The second declaration (2) assigns a block of code to be executed when the auction is closed.

The second part of the component shown in the example is the detailed auction view shown by the AuctionComponent. This component, unlike the previous one, was implemented using web live widgets, which means that the 'time left' label, the 'current price' label, the 'current winner' label and the 'bidders' list are created by binding a widget to a model. To do so, we create the value models in the component initialisation and later associate them to the widgets in the rendering phase. In the following code each numbered expression is in charge of creating a new value model.

```
AuctionComponent >> initialise
    super initialise.
    ...
    (1) self timeLimit: (MeteoroidNS.ValueModelWrapper
                            with: self auction
                            aspect: #timeLeft).
    (2) self price: (MeteoroidNS.ValueModelWrapper
                            with: self auction
                            aspect: #bestBidPrice).
    (3) self currentWinner: (MeteoroidNS.ValueModelWrapper
                            with: self auction
                            aspect: #bestBidder).
    (4) self bidders: (MeteoroidNS.SelectionInListWrapper with: self auction users).
```

After the value models have been created we can bind them to specific live tags in the rendering phase. Note the correspondence of each numbered sentence with the previous code.

```
AuctionComponent >> renderContentOn: aCanvas

    (1) (aCanvas liveDivFor: self timeLimit)
                labelBlock: [:aTimestamp | aTimestamp asTime print24].
    (2) (aCanvas liveSpanFor: self price)
                labelBlock: [:aPrice | '$' , aPrice displayString].
    (3) (aCanvas liveSpanFor: self currentWinner)
                labelBlock: [:anUser | '(' , anUser displayString , ')'].
    (4) aCanvas liveSelectFor: self bidders.
```

As a result, a complex soft real time web application can be implemented in a modular way, freeing the developer of handling low level connection details or browser dependencies.

5 Concluding remarks and future work

Internet has evolved to a point where web applications started to mimic desktop ones. These new types of applications make the actual protocols outdated, needing novel technologies and techniques. Among the new techniques that appeared in the last years, Comet gives web developers the ability to establish a constant communication channel between the server and the client. Through this channel the server can push events and data to the client, allowing the web page to be constantly updated.

Along this paper we presented Meteoroid, a Comet framework designed in a layered architecture. The first layer of the framework is in charge of providing a bidirectional communication channel between the server and the client, choosing the best connection technique based on the web browser that requested the page. The public API of this first layer is a single message that takes an arbitrary JavaScript code from the server, pushes it to the client and later executes it. To ease the design and maintainability of the framework we have encapsulated the different Comet techniques in handlers, allowing for a clean way of adding new techniques as they appear.

On top of this first layer an observer layer was mounted, allowing the server to send events to the client and react accordingly. Thanks to this layer the developer does not have to worry about how to relate the model changes to the web views; the only thing he has to do is establishing the appropriate action to perform when the model announces a change. To help in this task the layer also implements a set of predefined functions (like replacing the contents of a component, inserting a component after another, etc.).

The final layer, built on top of the previous two, completes the framework by adding web live widgets that are automatically synchronised with the application model in the server, effectively mounting a real MVC paradigm for the web. Thanks to this last layer the developer only has to bind a widget to an aspect of the application model by means of a value model. Once this is done the widget will be automatically updated according to the model's changes.

With the new standardised WebSockets specification, Comet-like techniques are getting attention and hopefully we will some day see solid a cross-browser protocol for server-client communication. While these kind of protocols would help in the development of low-level communication (which can be correlated with our PushScript layer), there is still a big gap to fill to reach a full MVC architecture for the web. The Observer and WebLiveWidgets layers of our framework take care of that functionality by applying well-known micro architectural patterns (such as value model, adaptor, observer, etc.) and building a toolkit that is ready to use to create live web applications.

Despite we consider Meteoroid to be a very complete framework, there are still open issues for research. Maybe the most important one is performance benchmarking, to test if the framework can be used in a production server with hundreds of clients working at the same time. The only test carried out so far was a chat application supporting 90 concurrent users and showing no problems and fast updates. Yet, many stress tests have to be carried out.

An extension that we are currently working on is the use of PrintConverter class (Woolf, 1996), which is a class that converts the view representation of a value to the real value in the model and vice versa.

Finally, since the WebGUI development using Meteoroid resembles the desktop GUI development, most of the visual tools used to develop desktop applications can be modified to automatically generate web views code instead of desktop ones.

References

ActiveX Object (2009) Available at http://www.msdn.microsoft.com/enus/library/7sw4ddf8(VS.85).aspx (accessed on 21 December 2010).

APE (2010) *APE Official Site*, available at http://www.ape-project.org (accessed on 21 December 2010).

Comet Maturity Guide (2009) Available at http://www.cometdaily.com/maturity.html (accessed on 16 May 2011).

Coplien, J.O. and Schmidt, D.C. (1995) *Pattern Languages of Program Design*, Addison-Wesley, One Jacob Way Reading, Massachusetts.

Crane, D. and McCarthy, P. (2008) *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*, Apress, Berkeley, CA, New York, NY, distributed to the book trade in the USA by Springer-Verlag.

Crane, D. et al. (2007) *Prototype and Scriptaculous in Action*, Manning Publications, Sound View Court 3B.

Ducasse, S. et al. (2010) *Dynamic Web Development with Seaside*, e-book.

Fernández, J.L. et al. (2009) 'Meteoroid: towards a real MVC for the web', Paper presented at the *International Workshop on Smalltalk Technologies*, Brest, France.

Gamma, E. et al. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, One Jacob Way Reading, Massachusetts.

Garrett (2005) 'Ajax: a new approach to web applications', available at http://www.adaptivepath.com/ideas/essays/archives/000385.php (accessed 21 December 2010).

HTTP (1999) *Hypertext Transfer Protocol (HTTP/1.1)*, available at http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1.4 (accessed 21 December 2010).

IceFaces (2010) *IceFaces Official Site*, available at http://www.icefaces.org (accessed 21 December 2010).

JavaServer Faces (2010) *JavaServer Faces Official Site*, available at http://www.java.sun.com/javaee/javaserverfaces (accessed on 21 December 2010).

Java Servlets (1999) *Java Servlet Specification, v2.2*, available at http://www.java.sun.com/products/servlet/2.2/ (accessed 21 December 2010).

Krasner, G.E. and Pope, S.T. (1988) 'A description of the model-view-controller user interface paradigm in the smalltalk-80 system', *Journal of Object Oriented Programming*, Vol. 1, No. 3, pp.26–49.

LightStreamer (2004) *LightStreamer Official Site*, available at http://www.lightstreamer.com (accessed 21 December 2010).

Mahemoff, M. (2006) *Ajax Design Patterns*, O'Reilly Media, Gravenstein Highway North, Sebastopol, CA.

Meteor (2010) *Meteor Official Site*, available at http://www.meteorserver.org (accessed on 21 December 2010).

Migratory Benchmarks (2010) Available at http://www.migratory.ro/data/MigratoryPushServerBenchmarks.pdf (accessed on 16 May 2011).

Orbited (2008) *Orbited Official Site*, available at http://www.orbited.org (accessed on 21 December 2010).

Pushlets (2007) *Pushlets Official Site*, available at http://www.pushlets.com (accessed on 21 December 2010).

Russell, A. (2006) 'Comet: low latency data for the browser', available at http://www.alex.dojotoolkit.org/wp-content/LowLatencyData.pdf (accessed 21 December 2010).

Server-Sent Events (2009) *Server-Sent Events*, available at http://www.w3.org/TR/eventsource (accessed on 21 December 2010).

Snook, J. et al. (2007) *Accelerated Dom Scripting with Ajax, Apis and Libraries*, Apress, Springer-Verlag New York, Inc., New York, NY.

WebSockets (2009) *The WebSocket API*, available at http://www.w3.org/TR/websockets/ (accessed on 21 December 2010).

WebSockets Disabled (2010) Available at http://www.hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/ (accessed on 21 December 2010).

Woolf, B. (1996) 'How to display an object as a string: TypeConverter and PrintConverter', *The Smalltalk Report*, July–August, Vol. 5, No. 9, pp.4–7.

XMLHttpRequest (2010) *XHR – XMLHTTPRequest Object*, available at http://www.w3.org/TR/XMLHttpRequest (accessed on 21 December 2010).

Notes

1 Benchmarking a Comet server involves not only the amount of concurrent clients but also the update ratio and the messages size.

2 http://www.lightstreamer.com/demo/WebTelemetryDemo/

3 http://www.voltaaomundoem26dias.net/

4 http://www.lightstreamer.com/pieceofnews_0033.htm