Meteoroid
Towards a real MVC for the Web*

Lautaro Fernández
LIFIA. Facultad de Informática. Universidad Nacional de La Plata, Argentina
lfernandez@lifia.info.unlp.edu.ar

Santiago Robles
LIFIA. Facultad de Informática. Universidad Nacional de La Plata, Argentina
srobles@lifia.info.unlp.edu.ar

Andrés Fortier
LIFIA. Facultad de Informática. Universidad Nacional de La Plata, Argentina
DSIC. Universidad Politćnica de Valencia, Valencia, España
Also at CONICET
andres@lifia.info.unlp.edu.ar

Stéphane Ducasse
RMod, INRIA Lille Nord Europe - LIFL - CNRS UMR 8022.
stephane.ducasse@inria.fr

Gustavo Rossi
LIFIA. Facultad de Informática. Universidad Nacional de La Plata, Argentina
Also at CONICET
gustavo@lifia.info.unlp.edu.ar

Silvia Gordillo
LIFIA. Facultad de Informática. Universidad Nacional de La Plata, Argentina
Also at CICPBA
gordillo@lifia.info.unlp.edu.ar

ABSTRACT

Web development has moved from simple static pages to complex web applications, some of them resembling desktop ones. In most of these applications the web browser acts as thin-client (or a view) of the model that sits on the server. Despite the technological evolution of the web, there is still no standard mechanism to send data or events from the server to the client without an explicit request from the later, thus forcing the web browser to constantly poll the server for updates. To solve this problem a set of techniques under the name of Comet were proposed, allowing to send information from the server to the web browser without an explicit client request. In this paper we introduce Meteoroid, a Comet approach to make "live" Seaside applications. Our framework exploits the Model-View-Controller (MVC) paradigm for building simple yet scalable web applications, requiring very little programming effort.

Categories and Subject Descriptors

D.1.5 [**Object-oriented Programming**]: Language; D.3.2 [**Programming Languages**]: Smalltalk; D.3.3 [**Language Constructs and Features**]: Frameworks

General Terms

Design, Languages

Keywords

Comet, Seaside, Web, MVC, Push vs. Pull

*This paper has been partially supported by the Argentine Ministry of Science and Technology (SeCyT) under the project PICT 11-32536. *Separación de Concerns de diseño en aplicaciones Web, Colaborativas y Móviles*

IWST'09 August 31, 2009, Brest, France.

1. INTRODUCTION

In its "standard" conception, the HTTP protocol uses a request/response pattern to achieve stateless, client-server communications. Each time the client (i.e. the Web browser) needs to receive (or send) new information from (to) the server, a request is created and sent to the server. As a result of that request a response is returned by the server, ending the transaction.

This kind of communication has proven to be effective for many years and for millions of sites, especially where changes on the server are not usual or do not happen at all (e.g. sites displaying static html). This kind of sites are usually referred as static websites [23]. However, as websites evolved new requirements appeared, like the need for an underlying domain model, data persistence and better interaction with the client. This evolution continued until Web applications appeared (i.e. full-fledged applications that are accessed by means of a Web browser [24]) taking all the complexities of software development to the World Wide Web. Compared to the "old" static Web pages, in these applications the interaction with the client and the content updates (which tend to be quicker than before) play a central role. This last statement is especially important for those applications where changes in the model (i.e. in the server) are useless if not seen on time.

As an example consider an on-line newspaper that keeps track of a tennis match or a Web messenger that can be used to chat and share files. If the information is not delivered on time (i.e. correctly updated in the client) the whole

point of the application is lost. In the Web2.0 [16] this issue takes more relevance, since the users are the content creators and the Web becomes the supporting platform for their activities. Web2.0 users need to share information and collaborate in real time. In this scenario the information displayed in a Web page must not only be updated according to the action taken by the client (or by an editor), but according to the changes made by many on-line users. In this kind of websites the old request/response communication becomes insufficient, since the Web browser does not know when the model has changed. As a result, the view in the browser may show an old version of the model.
To cope with this problem a simple (and rather primitive) solution is widely used, based on the update of the Web page (or parts of it) based on timeouts, which will be further explained in Section 2). The basic idea is that the web browser has a threaded script which constantly asks the server for updates. To illustrate this approach (and the problems related to it) consider a chat application with a simple domain model, comprising a chat room and a collection of registered users. To have a coherent conversation not only the order of the messages arrival must be reflected correctly in all the clients, but also the updates must be done as soon as the model changes. If we fail to provide these basic properties, the application may become sluggish, causing a negative effect on the final user. In a pull-like approach we must define a timeout to refresh the client content, which means anticipating the application's change rate. This is not a simple task, since in a chat application the change rate in the middle of a conversation may be in order of a second, whereas on idle times the rate may slow down to minutes or even hours. The bottom line here is that in certain application domains it is not possible to establish a definitive change rate and a compromise must be achieved between final user experience and resource consumption (like bandwidth and server processing).
The problems presented earlier are not new and there have been many attempts to use web-based technology to update time-critical information. There are two main streams to solve the above, a Client Pull [33] or a Server Push [33] technology, which will be further explained in Section 2.
In this paper we will present a framework for sending server events and data to the client as an extension to Seaside. In particular we will show:

- A Server Push implementation on top of Seaside [31] that runs efficiently on most Web browsers.
- How this implementation allows a developer to implement client (view) updates without any Server push specific knowledge.
- How to create widgets that are automatically updated whenever the model changes. With these widgets the developer can build desktop-like applications in an easy and scalable way.

2. PULL VS. PUSH: HISTORY OF COMET

In the HTTP protocol, the communication between the server and the browser was conceived (and it still is thought in that way accordingly to RFC 2616 [4]) to deliver a response if and only if the client has previously made a request. This means that the server does not have the ability to send new data to the client if it was not explicitly instructed to do so.
In 1995 Netscape acknowledge this as a drawback and presented "The Great Idea" [33] to solve the pushing disability from HTTP by proposing the Client Pull and Server Push approaches. The Client Pull mechanism is based on requests made by the browser in order to obtain novel data from the server. Each time the Web browser wants to refresh information within the page, it must do a request to obtain the desired resource from the server and update its contents accordingly. On the other hand the Server Push technique leaves an open connection between both sides, enabling the server to send new data when required. In the next sections we will review both approaches, describing some techniques to achieve them.

2.1 Client Pull

In this section we will present three different techniques to update data in the Web browsers using Client Pull. The first is based on refreshing the whole page while the second and third are based on updating only portions of it. For an overview of how the Client Pull technique works see Figure 1.

Meta Refresh Tag. This technique is achieved by inserting a meta-tag in the head element of the HTML in order to force a full refresh of the page. Along with the meta-tag a timeout is defined, which means that the whole page will be refreshed whenever the time expires. If the timeout is set accordingly to the changes in the model, the page will be effectively reflecting the server contents.

Ajax and Javascript's Timers. Another approach to do Client Pull is by combining Ajax [22] and JavaScript's timers [21]. Ajax allows the Web browser to perform "silent" calls to the server, meaning that the browser can send an asynchronous request to the server to query for changes. With the help of Javascripts' timeout functions (setTimeout or setInterval) an infinite loop can be written to periodically query the server for new data, analyze it and decide if any part of the Web page should be changed (e.g. manipulating the DOM [26][3] tree). The request to the server is performed by using an XMLHttpRequest [15], which basically retrieves the data from the server to the calling script, which can later process the data.

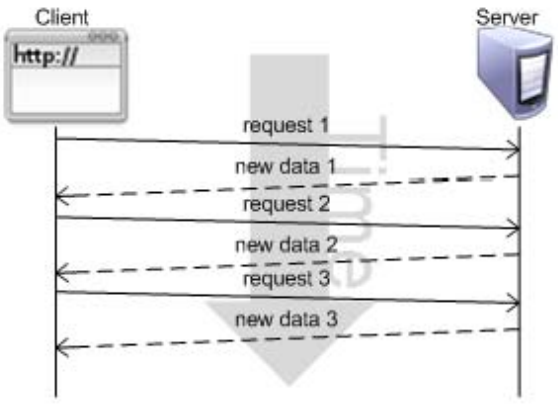

Figure 1: Client Pull diagram

The second approach has some advantages over the previous method, since an XMLHttpRequest is done asynchronously and "behind the scenes" by the browser. This means that the page is always responsive to the user (this would not happen if it was a synchronous call, because the browser's Javascript engine will be blocked) and that there is no visible change to the user (of course, unless the script that receives the response decides to alter the page). Besides being transparent, an Ajax call allows the Web browser to only ask for the desired data instead of requesting the full page (which is the case of doing a full refresh). Once the response has been sent by the server, the browser can work with the new data (e.g. to update the page). This leads to a better user experience, avoiding flickering, reducing network traffic (for a case study see [19]) and processing load both at the client and the server.

Long Polling. This proposal is a technique [17, page 41] which consists on making a request to the server, but instead of returning a complete response, the server leaves the connection open. Then, when the server needs to send data to the browser the response is sent and the connection is closed. After that, the client makes another request to enable receiving data from the server asynchronically. By constantly opening a new connection after the previous has been closed a persistent channel between the server and the client is emulated. This technique is similar to Ajax and Javascript's timers, but works different at the request level: Long Polling will do a request once, and until the answer is not delivered by the server it will not make another request. On the other hand Ajax and Javascript's timers are constantly sending new requests to query the server for changes.

2.2 Server Push

A completely different approach for server updates is to implement a Sever Push mechanism, where the server sends new information (events or data) to the client. This approach has the advantage of sending information only when needed, avoiding late data arrival and unnecessary pushes. However, this means that the server must have a constant connection (generally an open socket) per client, which (depending on the server capabilities) can lead to a server crash. To avoid this problem, different approaches can be taken, such as using grids of computers or load balancing between servers.

This difference between Client Pull and Server Push becomes more important when developing Web applications whose change rates can not be anticipated, or in those cases where bursts of changes are followed by poor activity. In those cases using a Client Pull technique with a very short timeout will give a responsive application but will cause unnecessary network traffic and CPU processing. On the other hand, setting a long timeout will make the user miss changes, especially when they arrive in bursts. A clear example of the later are Web based chats or tweets, where many short lines of text can be exchanged in seconds, while later on minutes or even hours can pass until a new change is posted in the server. In this situation using a Server Push technique would deliver the messages in time, without unnecessary processing or network traffic (see Figure 2 to clarify how Server Push works).

By combining the Server Push approach (i.e. sending data from the server to the browser) with Ajax (i.e. creating asynchronous requests from the browser to the server) the concept of Comet [17, page 7] was born. Comet is basically a way to exchange information between server and browser, specially focused on changes triggered by the server. Unfortunately, Comet (which is actually an umbrella for a set of possible techniques for server-browser communication) is not a standard one yet, and the developer has to deal with intricate browser-specific techniques to achieve it. In the current state of the art there are two main trends to achieve server push: using a plug-in-based implementation or combining HTML and Javascript.

2.2.1 *Plug-in-Based Implementations*

Plug-in-based implementations were the first ones to achieve this kind of connection. Maybe the best known in this area are the Java Applets, which use a persistent TCP connection between the server and the client. Other examples of plug-in-based approaches are Flash [7], Silverlight [8] and OpenLaszlo [9]. The main problem with all of these approaches is that they are not part of a standard browser product, requiring a specific installation in the client, restricting to vendors requirements (e.g. Silverlight only works for Windows OS) and, in some cases, the use of proprietary software. On the other hand if a Comet implementation only uses standard components (e.g. Javascript), the application would work out-of-the-box in all browsers that are W3C-compliant [10].

2.2.2 *Comet with HTML and Javascript*

An alternative approach is to combine HTML and Javascript, thus avoiding any special requirements on the client. In fact, by using the standard protocols and specifications the Comet implementation works also in smaller Web browsers (e.g. those which run on mobile devices). In this area we can mention four main techniques:

Streaming Servers. A possible approach for Comet implementations based on HTML and Javascript is using streaming [29]. In the old days, when a server received a request a response was created, converted to text and sent back to the client. As web servers evolved they started to support streaming, which means that they can send the response to the client in chunks as the different parts of the page are created. Streaming greatly improves the user experience, since the information in the page can appear faster, thus giving the idea that the page itself is loaded faster. The

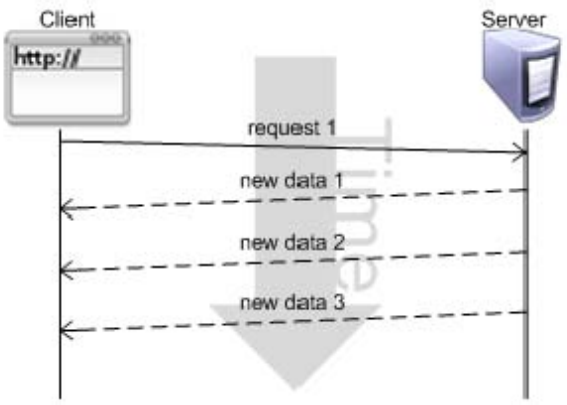

Figure 2: Server Push diagram

streaming facility is also useful to retrieve long responses, such as multimedia resources (big images, videos, audio, etc.) which would otherwise take a lot of time to download. In this scenario the streaming capabilities can be exploited to implement server-client communication. The trick is done by forcing the server to leave the response always open (i.e. never closing the response) and use that response as a communication channel. When new data must be passed to the client it is encoded and appended to the open response. The client in turn can incrementally process the response with Javascript, updating its contents as needed.

Gifs: a Limited Workaround. In 1999 a different approach was used to implement streaming, based on the approach used by the Web browsers to load gif images. Ka-Ping Yee created at that year a chat application using a gif image [34]. The basic idea is to deliver an endless gif where the messages posted by the users are rendered. Each time a chat user sends a new message the server completes a new image row with the message and delivers it to all the browsers. Even though it is quite limited (since there is no DOM manipulation through Javascript and the changes are only reflected inside the gif) it is also a way of doing streaming.

Generic Solution: Forever IFrames. To accomplish the DOM manipulation via streaming an approach called forever IFrames was developed. This Comet technique uses an IFrame element in the page, which will load a "special" URL where the streaming response will be delivered. This approach is quite interesting since it works in most browsers because the IFrame is a standard tag. However this approach has many usability problems when it comes to user interfaces, since all Web browsers have different ways of showing that a page is still being loaded. In some browsers while a page is being processed the mouse pointer is rendered with hourglass icon, throbbers are shown [28] (small icon which loops if the page is not fully downloaded) or the status bar displays a text cue indicating that the page is incomplete (like "loading..."). Given that this approach is based on a page that is never fully loaded those notifications will be constantly displayed (and depending on the browser, all of them may appear at the same time). Even though this has no effect in the website logic, it does give the user semantically wrong cues about the page, since the contents have already been loaded and are the events from the server what the browser is waiting for.

Browser-Specific Solutions. Since Comet is not a standard yet (at least not in the W3C sense) different Web browsers require different techniques. In particular we have found the following approaches to work well in each browser:

- Opera. Uses Server-Sent events [13], which allows sending events from the server to the client. These events are handled in the client with Javascript.
- Internet Explorer. Uses a combination of ActiveX and IFrame [1].The ActiveX object creates an in-memory page, which has an IFrame that is dynamically loading the page. With this technique the loading bar and the throbber are not seen.
- Mozilla-based Browsers. The client uses the XMLHttpRequest [15] and uses its interactive state to parse the upcoming data which came from the server in the browser

In case a browser-agnostic approach is required (e.g. to handle old browsers) the forever IFrame technique must be used, even though the disadvantages discussed earlier will be shown.

3. COMET THE SMALLTALK WAY

Seaside is a Web application framework written in Smalltalk, which has lately gained a lot of attention. The framework combines an object-oriented approach with continuations [32], allowing multiple control flows on a single page, one for each component. Seaside is also a component-oriented framework, which means that modular components can be developed and later reused in many different applications. A great advantage of the framework is that the applications are written in Smalltalk itself, avoiding most of the html and Javascript coding and enabling the developer to use the same environment for web development that was used to build the domain model. As a simple example of the above, the following code:

```
renderContentOn: html
  html heading level: 1;
    with: 'Header'.
  html div id: 'divID';
    with: [html strong: 'bold'].
```

will render

```
<html>
  <body>
    <h1>Header</h1>
    <div id='divID'>
      <strong>bold</strong>
    </div>
  </body>
</html>
```

In the rest of the paper we will describe Meteoroid [5], a Comet implementation built on top of Seaside. The aim of Meteoroid is to allow Seaside developers to build Web applications that exploit the Comet idea of sending events from the server to the client. Special care has been put in the implementation of Meteoroid to fit the Seaside style of coding, integrating Comet in a natural way to the Seaside developer.

3.1 Basic Usage

In our approach we decided to create the Comet communication by means of a small Javascript script. This script is inserted as part of the processing phase of a Meteoroid page in the server side and is executed when the page is loaded in the web browser. When the script is executed a new channel between the client and the server is opened. In this channel the server will send the new information to the browser when required.
To add Comet behavior to a Seaside application only two tasks must be performed:

- Use the `Meteoroid` abstract class as an application superclass instead of the standard `WAComponent`.

- Change the session class to `MeteoroidSession` instead of `WASession`.

These two requirements will be further explained in Section 4.
The core behavior added by the `Meteoroid` class is the `#pushScript:` message, which takes as a parameter a javascript function encoded in a string. This message allows the server to send the script to the browser, which will then be executed. As an example of its usage consider a Meteoroid page (`MeteoroidExampleAlert` class) that has a `#showDialog` message used to trigger an alert dialog within the Web browser. This can be achieved by using the following code:

```
MeteoroidExampleAlert>>showDialog
  self pushScript:'alert("Hello Meteoroid");'
```

It is important to note that the `#showDialog` message does not have to be initiated by a browser request, but by any object in the server image. To illustrate Meteoroid's real aims consider now the Seaside counter example [11], which is a simple page that can increase or decrease the value of a counter. In this example each counter page holds its own counter model and thus there is nothing shared between different instances of the same Web application. A slight variation of that example that shows Meteoroid's power would be to have a shared counter model between all the pages. In this case all the pages will have as domain model a `SharedCounterModel`, which is a singleton [20, Singleton pattern] of a counter model. Each time a web browser increases or decreases the counter's value, it will modify the unique shared counter and that new value will be displayed on all Web browsers. To make it possible, we use the Observer pattern [20, Observer pattern] in which the subject is the `SharedCounterModel` and the views (observers) are the `MetCounter` pages (see Figure 3). For each new change in the value of `SharedCounterModel`, a `#change` is sent to update all the views which depend on the collaborative model.

```
MeteoroidExample>> updateCounter:  aValue
  self pushScript:  '
    document.getElementById("counter").innerHTML
    = ' , aValue printString , ';'
```

Where the DOM element under the "counter" ID, is a div element that holds the value of the `SharedCounterModel`.
A slightly more elaborated application would be to implement a chat room using Meteoroid. To do so we assume that

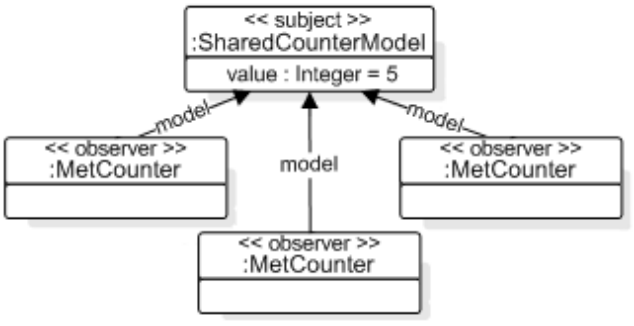

Figure 3: Chat Instance diagram

the application model is composed by the classes:

- `Message.` It holds a text and the user who wrote it.
- `User.` An object which can post new messages into the `Room`.
- `Room.` It has a collection of users and messages. Each time a user makes a post, the `Room` is the responsible of create a `Message` from that post and replicate it into all the users.

Besides the model, a Seaside component named `MetChat` was created to be the view of the chat room, which will show all the chat changes. The Observer pattern is used again to show the chat changes, where each User is the *subject* of the `MetChat` view. Each time a user writes a new message and sends it (e.g. through the `MetChat` view), the `Room` model will receive it and it will replicate that message to the rest of the `User` instances. The `#message:from:` message is in charge of doing so by creating an instance of the `Message` class. When the new `Message` arrives to each concrete `User` instance, it will trigger a `#change:` that is going to be listened by his `MetChat` instance, which will later trigger the `#update:` message. That instance will push the new message through the Comet connection in order to show it, by using the already mentioned `#pushScript:` message:

```
MetChat>>update:  aMessage
  self pushScript:  ( 'insertMessage(" '
    aMessage user, ' ", " '
    aMessage body,
    , ' ");' )
```

Where the Javascript function `insertMessage(user, message)` will print the new message within the browser using DOM techniques (i.e. mostly using Javascript libraries like Prototype [18, page 7] or similar). In order to clarify the example we next show a small sequence diagram (see Figure 4) where two users are logged (User1 and User2). In the diagram User1 has just posted a new message.

3.2 Meteoroid at a Higher Level

In Seaside, the rendering process is described by using Smalltalk objects and avoiding, as much as possible, the direct coding of html and Javascript. Since our aim is to fit Meteoroid in Seaside in a seamless way we created a protocol that works in a more abstract way than the `#pushScript:` one.
The first step to improve our implementation was to create helper methods to automate repetitive tasks that should be otherwise hardcoded in Javascript. The second improvement was achieved by adding an Observer-like protocol, so that the Web browser can be considered as a view of the model that resides in a Smalltalk image. Historically the MVC [27] architecture has been used to decouple the underlying model from the GUIs build on top of it. This architecture heavily relies on the Observer pattern, which has many different implementations in the different Smalltalk dialects (note that even the same Smalltalk flavor can have many observer implementations). One of the most widespread implementation of the observer pattern was the one based on the `#change:` family of messages, where a symbol was passed as a parameter indicating the aspect of the object that had changed. Even though this implementation has proven to be

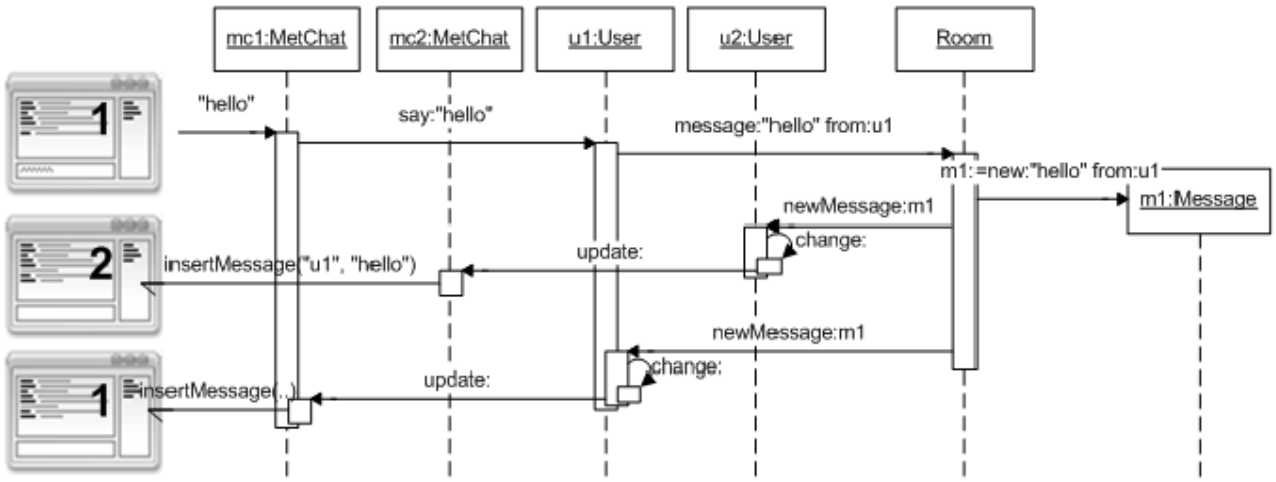

Figure 4: Sending a new message

useful, it has its drawbacks and different alternatives to the `#change:` mechanism were later introduced (such as triggers or announcements). In this area, Announcements [30] offer a very interesting approach, since they allow the developer to express events as objects and to handle them using an exception-like mechanism.
In our framework the benefits obtained by the use of an observer mechanism are combined with a set of common actions performed in Web applications exposed by script.aculo.us [18, page 11]. As a result, a general Meteoroid protocol is provided, allowing either to update or insert new information when the model changes. To establish the dependency between the Meteoroid components and the model the Announcements framework is used.
As an example, consider again the `Counter` example explained in Section 3.1. By combining announcements with high-level messages to handle script.aculo.us functions we can express the dependency and update code in a very simple way. For this example we assume that the shared counter triggers a `ValueChangedAnnouncement` each time its value is increased or decreased. Thus, in the `Meteoroid` component initialization we should write:

```
self on:  ValueChangedAnnouncement
  of:  self counterModel
  update:  'value'
  callback:  [:html |
    html text:  self counterModel value
    ]
```

Notice that in the example the update code is performed by means of a block that gets evaluated each time the counter changes. The block can receive up to two more arguments, which are the announcement received and the announcer that triggered the event. The previous example can also be coded by using the announcer parameter, since it is the counter itself the one who triggers the change event.

```
self on:  ValueChangedAnnouncement
  of:  self counterModel
  update:  'value'
  callback:  [:html :announcement :announcer|
    html text:  announcer value
    ]
```

However, blocks are not the only way to implement an update. As an alternative, a selector can be used instead of the rendering block, which can also receive the same parameters that the block receives. To see this in action, consider the chat example described in the Section 3.1, but now implemented with announcements. In this example when a new message arrives, an announcement is triggered by the chat room. With the protocol provided by the `Meteoroid` class, the developer only needs to specify the announcement and the selector required to update the Web page.

```
self on:  NewMessageAnnouncement
  of:  self chatRoom
  insertIn:  'messages'
  at:  #bottom
  sending:  #renderNewMessageOn:
```

As a result each time a new message arrives to the chat room it will be inserted at the bottom of the 'messages' div. The rendering of the message is performed by the `#renderNewMessageOn:` selector. Another important thing that is provided by the `Meteoroid` class is the automatic management of the dependencies. Every time a dependency is requested using `Meteoroid` protocol, the information is kept in a collection owned by the `Meteoroid` component. With this information the component creates the dependency after the rendering process is finished and breaks it automatically when the page is closed. Therefore, the developer can focus on the functionality of the application and forget about implementation issues related to the `Announcements` framework.

3.3 WebValueModels (WVM)

The desktop GUI widgets in VisualWorks [2] rely on an underlying framework called ValueModels [25]. The idea behind a value model is that it represents a single value for a widget that can be accessed by means of the `#value` /

`#value:` messages. Also, when its value is changed, the `ValueModel` triggers an event so that any interested party can receive a notification. By using this approach widgets can be programmed assuming that their target will respond to the `#value` family of messages, independently of the real underlying model (e.g. a concrete subclass of `ValueModel` abstract class is an adaptor that converts the `#value` / `#value:` messages into domain-specific messages).
We found the value model approach to be well suited for developing GUIs and decided to take it to web development in our framework, in what we called `WebValueModels`. Even though the underlying idea remains the same as in the standard value models, some technical issues are different. In particular the announcements framework is used instead of the old dependents protocol, thus allowing a straightforward integration with the `Meteoroid` protocol presented earlier. Also, to be backward-compatible, the web value models are able to wrap almost all the existing value models and thus use them in a `Meteoroid` application.
Having the `WebValueModels` hierarchy in place, we implemented a set of Web GUI widgets that work in the same way that their desktop counterparts. These widgets are a normal HTML controls (i.e. input, text areas, lists, etc.) with the difference that they have an associated source of information that is a `WebValueModel`. The connection between the widget and the `WebValueModel` lets the widget access its model in a simple way and to get a change notification when the model has changed its value, so that it can be automatically updated. In our current implementation we support divs, inputs, text areas, selects, radio buttons, check boxes, ordered lists and unordered lists as Web widgets. A schema of the layers and the updating process is shown in Figure 5.

In order to show how the value models work we will next present two examples. The first one shows the value of a shared counter in a div tag. The model used for the counter is the same that was presented in the previous section, the change in this case is how the view is implemented. Recall that when using the `#on:of:update:callback:` approach, the Meteoroid component had to explicitly synchronize the model and the view. This behavior now has been moved to the web value model, thus we only need to create the value model:

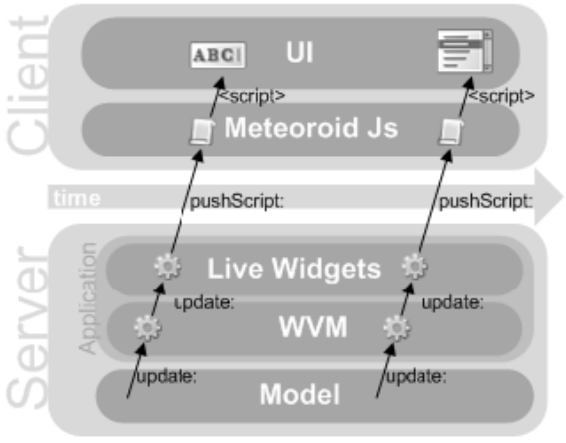

Figure 5: Meteoroid WebValueModel layers

```
MetCounter>>initialize
  super initialize.
  self count:  (WebValueModel
    with:  self counter
    aspect:  #count)
```

and connect the div tag with it:

```
MetCounter>>renderContentOn:  html
  ...
  html divUpdateableFor:  self count
```

Notice that in the example the web value model just acts as an adapter [20, Adapter pattern] between the counter interface and the protocol expected by the web widget (in this case the div tag).
As a second example we will return to the chat example (see Section 3.1) and show how to keep the list of logged users of a chat room. To do so, two simple things have to be done: bind the model to a component and display that component in a browser.
The binding between the model and the component is generally done in the `#initialize` method of the parent component (i.e. the component that renders the list). Since we are working with a dynamic list, a `SelectionInList` [25] (a special type of value model) is created:

```
Room>>initialize
  super initialize.
  self usersList:
    (WebSelectionInList
    model:  self chatRoom
    listAspect:  #users
    selectionIndexsApect:  #selectionIndex)
```

In this code a `WebSelectionInList` is created. Every time a user logs in, the room triggers a change event, which is converted by the `WebSelectionInList` into an announcement (`ListChangedAnnouncement`). Once the value model is created, we only need to connect it to the widget. Therefore the code of the `#renderContenOn:` message should look like:

```
ChatRoom>>renderContentOn:  html
  | select |
  ...
  select := html
    selectUpdateableFor:  self usersList.
  select
    labels:  [:user | 'User:  '  , user name];
    size:  5
```

These are the only two things required to show the user list, which will be automatically updated every time a new user arrives to or leaves the chat room. Thus, creating Web pages that are automatically updated is pretty straightforward.

4. METEOROID UNDER THE HOOD

In Section 3 we had shown how to use the Meteoroid API at different levels, starting with the basic `#pushScript:` message. In this section we will focus in the technical details of the work done to achieve the functionality mentioned in

Section 3.

Inheriting from Meteoroid. One of the requirements presented in the Section 3.1 was the fact that each component must inherit from the `Meteoroid` class. Besides adding a handler (at the end of this section we will explain it a bit more) and a couple of helper methods, the core behavior added by the `Meteoroid` class is the `#pushScript:` message, which receives a Javascript script as a string object. When sending this message the script passed as a parameter is sent from the server to the client and executed by the browser afterwards.
Even though this basic behavior requires low level programming (since we have to code Javascript by hand) is very powerful, because we can send any script from the server to the browser.

Choosing the Best Communication Technique for Each Browser. The Internet community is characterized for using different flavors of browsers. Each one has its own way to communicate with the server, which is a major issue when it comes to Comet. In order to handle each browser in a specific way, a new kind of session is required. For this reason the second requirement (as specified in the Section 3.1) is changing `WASession` for `MeteoroidSession`. When the client asks for a new Comet connection, the session is the one in charge of creating it and finding an appropriate technique (which is basically a handler) for the specific browser. For example, if the browser requesting the Comet connection is Opera the server will use Server-Sent events, whereas if the browser is a Mozilla-based one the XMLHttpRequest technique will be used. Thus, our session knows which is the "best" Comet technique based on the user agent [14] reported by the client. To achieve this we have modeled a hierarchy of classes to encapsulate each specific technique (for Mozilla-based XMLHttpRequest, for Internet Explorer based ActiveX and forever IFrames, etc.), allowing us to handle the streamed response in a better and cleaner way.

Handling the Browser. At the internal level, the handler represents the channel between the server and the client. Therefore, each time the page sends Javascript to the client using `#pushScript:` it is actually the handler which sends it. The handler also has a state to identify the status of the connection with the browser. The different states of a Meteoroid connection are modeled as a state machine, using the State pattern [20, State pattern]. A Meteoroid handler may be in three different states:

Beginning. This state is set when the handler is not ready for pushing scripts from the server (e.g. while the page is being loaded). This state is mainly used to avoid sending data to an early connection, which would otherwise cause a disconnection.
Running. Set when the handler is able to push data to the browser. If the data can not be sent (e.g. because the browser has been closed), the handler changes to the Waiting state.
Waiting. Used when the server is not able to send data to the browser. It models the fact that the connection is unable to send data, but that can be later reestablished. It is triggered in two different situations: when the web browser has been closed or when the client has left the Meteoroid page by navigating to another one.

The context provided by those states give Meteoroid developers the possibility to have more control over the flow of the browser, because the handler can be accessed by a `Meteoroid` component. Thus, custom actions can be taken upon state changes. For instance, if the handler is set to `Running`, the page is capable of sending new events to the browser, but if at some moment the connection breaks, the state becomes `Waiting`. Then the page can advice that, by checking the state and using it in its domain model (for instance, to log how people uses the page).

Meteoroid Connection Everywhere. We mentioned previously that the browser is responsible for making the Comet connection by means of a small Javascript. This approach was chosen due to a problem regarding the back and forward buttons present in any web browser. The idea of the back/forward button is to allow the user to navigate through an already rendered page. When navigating by means of the back and forward buttons the pages already rendered do not generate new requests to the server, since they are cached in the client. This generates a big problem in some Comet implementations, since when the user moves between pages the persistent connection is lost and cannot be re-established later. By performing the Comet connection in a Javascript script, when the page is loaded the script is executed [10], even if the page is cached in the client browser.

Nesting Meteoroid Components. Subclassing from `Meteoroid` and using the Meteoroid session has a major importance in the flow of how Seaside renders pages. Seaside is a web framework based on components, where developers can add, compose and tweak them as they want. In our approach developers can work in Seaside using the "normal way", and then just add Meteoroid. The major improvement of Meteoroid is in the way the component's rendering is done, managing the handler creation and the details required to even work with children components.
The handler, which is present in each Meteoroid page, is an object bound to the session which is later (inside the rendering phase) delivered into the components that need it. Notice that Meteoroid will only create one handler per session, and therefore the same handler will be used across the entire page, meaning that each component will use the same handler. This is quite important, because having one handler across the application will greatly reduce resource consumption.
We can now depict how a standard chat could be shown using Seaside components: a first component (`MessagesContainer`) will be on charge of rendering the text (chat room) for new messages. Another component (`UserList`) will be in charge of rendering the list of users, whereas an input field (`UserInput`) is used to send messages from each user to the server. Finally all those components are grouped by a root (`ChatRoom`), being `MessagesContainer`, `UserList` and `UserInput` their children (see Figure 6).
To reproduce the already mentioned requirements (inherit and changing the default session), `MessagesContainer`, `UserList` and `ChatRoom` must inherit from the `Meteoroid` class, and change the `WASession` session to `MeteoroidSession`. Notice that it is not necessary to inherit the input component `UserInput` from Meteoroid, because this component does not need to be "updated" from the server.

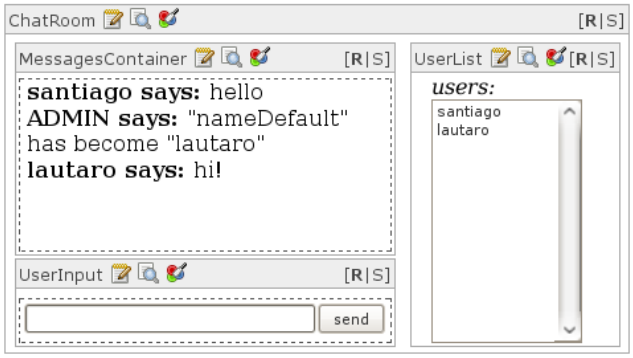

Figure 6: Chat components diagram

5. CONCLUSIONS AND FURTHER WORK

In this paper we presented Meteoroid, a Comet implementation in Smalltalk that works on top of Seaside. Our implementation of Comet is based on the early work of Lukas Renggli [12], a core developer of Seaside. This initial work only covered a way to send data from the server to each client. In our approach not only we modularized the implementation, but enhanced it to effectively support different techniques according to the target browser. On top of that layer we created the required abstractions to easily connect a domain model to a live web interface, effectively maintaining the information on the browser always updated.
Meteoroid can be used in three different levels of abstraction:

- At a very low level, by means of the `#pushScript:` message.
- In a medium level, by combining Ajax, Announcements and script.aculo.us.
- By using Web widgets associated to Web value models.

The above three layers allows the developer to effectively create web applications using the web browser as a renderer and web pages as views, inside the MVC paradigm.
As a result of our design an interesting feature emerged: since Meteoroid is compatible with the value model approach for MVC creating a Web application for a model which was used in desktop applications is almost straightforward, since only the visual part must be written again.
The next planned steps of the project are:

- Create a complete set of widgets, managing all their associated events.
- Define a tool to pack Seaside with Mozilla Prism [6], so that a Smalltalk desktop application can be delivered as a single package containing a web server and a client.
- Take the Meteoroid approach to mobile clients.

6. REFERENCES

[1] Activexobject object. http://msdn.microsoft.com/enus/library /7sw4ddf8(VS.85).aspx.
[2] Cincom visualworks official site. http://www.cincomsmalltalk.com.
[3] Document object model (dom). http://www.w3.org/DOM.
[4] Hypertext transfer protocol. HTTP/1.1 http://www.w3.org/Protocols /rfc2616/rfc2616-sec1.html#sec1.4.
[5] Meteoroid official site. http://cag.lifia.info.unlp.edu.ar/cag/Wiki /01.+Projects/Meteoroid.
[6] Mozilla prism official site. http://labs.mozilla.com/2007/10/prism.
[7] Official site. http://www.adobe.com/es/products/flashplayer.
[8] Official site. http://silverlight.net.
[9] Official site. http://www.openlaszlo.org.
[10] Scripts. http://www.w3.org/TR/REChtml40/interact /scripts.html.
[11] Seaside counter example. http://www.seaside.st/about/examples/counter.
[12] Seaside's core developer. http://www.seaside.st/community/contributors.
[13] Server-sent events. http://labs.opera.com/news/2006/09/01 http://www.w3.org/TR/html5/comms.html.
[14] Useragent specification. http://www.w3.org/TR/WAIUSERAGENT.
[15] The xmlhttprequest object http://www.w3.org/tr/xmlhttprequest. http://www.w3.org/TR/XMLHttpRequest.
[16] P. Anderson. What is web2.0? ideas, technologies and implications for education. 2007.
[17] D. Crane and P. McCarthy. Comet and reverse ajax. 2008.
[18] B. B. T. L. Dave Crane. Prototype and scriptaculous in action.
[19] A. M. Engin Bozdag and A. van Deursen. A comparison of push and pull techniques for ajax. 2007.
[20] R. H. R. J. J. M. V. Erich Gamma. Design patterns: Elements of reusable object-oriented software.
[21] D. Flanagan. Javascript:the definitive guide. 2006.
[22] J. J. Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/essays /archives/000385.php.
[23] D. F. C. H. C. J. G. F. Hale French. Internet based learning.
[24] D. C. James Duncan Davidson. Java servlet specification (specification) version: 2.2 final release. sun microsystems. pages 43–46, 1999.
[25] D. C. S. James O. Coplien. Pattern languages of program design. pages 467–494.
[26] A. G. S. L. D. W. Jonathan Snook. Accelerated dom scripting with ajax, apis, and libraries. 2007.
[27] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system.
[28] S. M. Lauriat. Advanced ajax architecture and best practices. page 17.
[29] M. Mahemoff. Ajax design pattern. page 113, 2006.
[30] C. Smalltalk. Application developer's guide.
[31] A. L. Stephane Ducasse and L. Renggli. Seaside - a

multiple control flow web application framework.
[32] A. L. L. R. Stephane Ducasse. Seaside - a multiple control flow web application framework.
[33] N. C. Technical report. An exploration of dynamic documents. 1995. Dead link: http://www.netscape.com/assist/net_sites/pushpull.html Chached reference: http://web.archive.org/web/*/www.netscape.com/assist /net_sites/pushpull.html.
[34] K.-P. Yee. Chat using dynamic animated images. http://zesty.ca/chat.