The Knowledge Engineering Review, Vol. 29:2, 134–153. © Cambridge University Press, 2014
doi:10.1017/S0269888914000022

Exploiting personal web servers for mobile context-aware applications

ANDRÉS FORTIER[1,2], CECILIA CHALLIOL[1,2], JUAN LAUTARO FERNÁNDEZ[1,3], SANTIAGO ROBLES[1,3], GUSTAVO ROSSI[1,2] and SILVIA GORDILLO[1,4]

[1]*LIFIA, Facultad de Informática, UNLP, 50 y 120, La Plata, Buenos Aires, Argentina;*
[2]*CONICET, Argentina;*
e-mail: andres@lifia.info.unlp.edu.ar, ceciliac@lifia.info.unlp.edu.ar, gustavo@lifia.info.unlp.edu.ar;
[3]*CICPBA, Argentina;*
e-mail: lfernandez@lifia.info.unlp.edu.ar, srobles@lifia.info.unlp.edu.ar;
[4]*CIC, Argentina;*
e-mail: gordillo@lifia.info.unlp.edu.ar

Abstract

There is an increasing trend in moving desktop applications to web browsers, even when the web server is running on the same desktop machine. In this paper, we go further in this direction and show how to combine a web server, a web application framework (enhanced to support desktop-like Model–View–Controller interaction) and a context-aware architecture to develop web-based mobile context-aware applications. By using this approach we take advantage of the well-established web paradigm to design the graphical user interfaces (GUIs) and the inherent ability of the web to mash up applications with external components (such as Google Maps). On top of that, since the web server runs on the device itself, the application can access local resources (such as disk space or sensing devices, which are indispensable for context-aware systems) avoiding the sandbox model of the web browsers. To illustrate our approach we show how a mobile hypermedia system has been built on top of our platform.

1 Introduction

Web applications are constantly gaining presence in everyday life. Today most languages have their frameworks and toolkits to build web applications and even Operating Systems are following this trend[1]. If we analyze this evolution we can see very good reasons that support it: web applications are inherently portable across operating systems and devices[2], they do not require installation, updates are managed in the server side (thus lowering the client burden) and the data back-up is a problem of the provider, not the client. On top of that, the web is rapidly becoming ubiquitous, part of our everyday life: we plan trips on-line, we buy electronics in virtual shops and meet people in social networks. We are also gearing towards the integration phase, where the so-called *mashups* combine different modules to provide better services.

On a different area, the ideas proposed by the Ubiquitous Computing (UbiComp) concept (and its related disciplines such as Pervasive Systems, Ambient Intelligence -AmI-, Context-Aware

[1] `http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html`

[2] Sadly things are not ideal, since there is no standard that works across all major browsers. However, there are libraries and frameworks that help to mitigate these problems such as Scriptaculous or ExtJs.

Applications, etc.) have been gaining importance. As time goes by, we are becoming more familiar with incorporating new technologies, appliances and applications in a way that was almost unthinkable 10 or 15 years ago.

In his seminal paper, Weiser (1995) envisioned a future where humans would be surrounded by a myriad of computing devices, aimed at improving the user's lifestyle. Nowadays there is a broad range of applications that are consider ubiquitous, like healthcare systems (Bricon-Souf & Newman, 2006), smart homes (Edwards & Grinter, 2001), automated tour guides (Cheverst *et al.*, 2002), Location-Based Services (Rao & Minakakis, 2003), memory aids (Lamming & Flynn, 1994) and augmented reality (Rekimoto *et al.*, 1998), just to name a few. In this scenario, it is reasonable to think that these two technologies (web applications and UbiComp) should meet to create the next generation of user applications: web applications that are cross-device, combining services that are aware of the user's situation and adapt to best suit his/her needs.

However, web applications have some limitations that are especially relevant to UbiComp. Due to security reasons, web applications are executed inside a sandbox, which restricts its access to local resources. This means that access to local files or a web cam in the client device is, in principle, forbidden. To overcome this problem we can embed special controls that break the sandbox model, such as Microsoft ActiveX controls. However, by doing this we may go against the main idea of web applications, since now we have to perform an extra installation step, the operating system stability may be compromised and the control has to be regularly updated.

Another major issue with web applications and UbiComp is the need for a network connection and the fact that the data itself resides outside the client's device. As a result the roaming user heavily depends on a set of external resources (access points, ISPs, hosting, etc.), which are a must for its application to run and that are not guaranteed to be accessible.

For these reasons, we consider that a different approach is required to successfully achieve the combination between web applications and UbiComp. In this article we propose our approach, which is based on a personal web server running in the user's device as a platform to provide UbiComp web applications. To build this platform we combine:

- A web server that supports streaming (Swazoo).
- A framework for building web applications (Seaside).
- A Comet-based approach to create real MVC web applications (Meteoroid).
- An architecture and its associated toolkit for developing context-aware applications.

By using these components we can create web applications that have access to local resources (since the web server runs in the device) with the natural facilities associated to web applications (e.g. using Google Maps). On top of this, the application framework has been extended with Meteoroid (Fernandez *et al.*, 2009), a Comet-based framework that provides server-client communication and desktop-like web widgets, allowing the developer to create web applications that have desktop-like interaction and responsiveness.

The rest of the paper is structured as follows: in Section 2 we review some of the most important works performed in different areas related to our research. In Section 3 we explain how we combined a personal web server with Comet to obtain web applications with full MVC support. In Section 4 we outline our context-aware architecture followed by a description of our mobile platform in Section 5. In Section 6 we show how a mobile hypermedia application can be built with our approach and we conclude the paper in Section 7 with the further work we are pursuing.

2 Related work

Weiser's (1995) pioneer work at Xerox Parc was followed by Schilit's (Schilit *et al.*, 2002) system architecture and the Stick-e Notes (Pascoe, 1997) framework. These first approaches to build context-aware systems helped to understand many of the challenges we are facing today and some interesting ways of tackling them. Maybe one of the most important lessons learned is Schilit's

decision to decouple the independently-changing blocks of the system to support scalability, which is a way of anticipating future changes. Also, Schilit emphasized that sensing devices should be abstracted, so that the application logic does not have to get involved with the burden of connecting to hardware devices and sensing information.

The next milestone in the history of context-aware architectures was Dey's Context Toolkit (Dey, 2000), which is a pioneer framework aimed at building distributed context-aware applications, based on a peer-to-peer architecture. The framework is built around the notion of context widgets, which are used to access context data (gathered by physical or logical sensors) using the widget metaphor. Context interpreters are used at a later stage to abstract the widget's information, which may also be composed with context aggregators. The toolkit is finally completed with services (which implement the required behavior) and discoverers, which know the state of the application in terms of available components (i.e. widgets, interpreters, aggregators and services).

In a recent work on mobile services by Pederson *et al.* (2008), we have found many interesting points that we share in our approach. One of their first statements is that different applications cater for different aspects of the user's context, which means that a unique context model for different applications is not a viable solution. They also stress the fact that the context may be on a server, on a client or distributed between both of them. We clearly share this view and this is one of the main reasons why our design splits context in a set of small grained features.

Moving to the web area, in Ceri *et al.* (2007) the authors show how to incorporate changes in the page generation logic that depend on context. The authors present a model-driven approach for context-aware web applications based on WebML (Ceri *et al.*, 2000), where context is treated as a first-class citizen, operating on the hypertext the users navigate. In Daniel and Matera (2008) the authors present a component-based approach to build adaptive web applications. In particular, the authors focus on adapting to context by using client-side context information (both from local and remote sensors) and context data found in the server. Each UI component is defined by an XML file stating the properties of the component and the events it can trigger. The authors propose an event-based communication mechanism between components, managing their connection through listeners, which are configured in another XML document (the XPIL file). Finally, Chang and Agha (2007) show an approach for building context-aware web applications by adapting to specific contexts through reconfigurable component distribution. The authors can add new context features by adding the corresponding context variables and monitors, along with the adaptation policies to these variables.

Since the Mobile Web era began, an important amount of research has been devoted to managing small screen resolution and low bandwidth, two common characteristics of mobile devices when compared with their desktop counterparts. Many researchers came to the conclusion that just shrinking a desktop-sized web graphical user interface (GUI) to fit a mobile display did not yield satisfactory results, and thus contents should be tailored to fit the mobile web characteristics (Billsus *et al.*, 2002). However, as noted in (Nichols *et al.*, 2008), as new devices offer better screen resolution and browsing capabilities (such as multi-touch screens and gestures) more support is given to the user to browse desktop-sized web sites.

In the human–computer interaction (HCI) area, the field work carried by Pascoe *et al.* (2000) concluded that mobile applications should avoid distracting the user from his current activity and that applications enhanced with context-aware behavior can improve the user experience. Finally, as mobile devices gained popularity, toolkits and frameworks started to appear to develop mobile web applications. While most of them are geared towards GUI management (e.g. iUI, jQtouch, iWebKit) domain-specific frameworks are also starting to appear (Simon & Fröhlich, 2007).

An interesting combination of web applications and UbiComp ideas is realized in the Physical Hypermedia (PH; Grønbæk *et al.*, 2003). In this approach the 'standard' hypermedia is extended to include physical artifacts and materials, augmenting real-world objects with digital information. In Grønbæk *et al.* (2003) the authors present a prototype based on a RFID tagging system, which associates digital information with real-world objects. When the user reads a RFID tag, he

receives digital information about it. In a similar area, Bouvin *et al.* (2003) present a framework for Context-Aware Mobile Hypermedia applications (HyCon), which supports four different context-aware techniques: browsing, searching, annotating and linking. Finally, Harper *et al.* (2004) extended the metaphor of links with the idea of 'walking' a link as they augment hypertext with physical relationships present in the real world. The authors use the concept of walking the link to represent the intentions and movement of the user to reach a real-world target.

In general terms, the PH approach is based on the well-known and intuitive navigation-by-links style of web software, extending it to the physical level, where the mobile user can access information items both physically and digitally. We have studied the field of PH and based our research in the mentioned previous works. We have also extended the concept of PH beyond information about real-world objects towards a generic mobile hypermedia model and have characterized the different browsing semantics (physical vs. digital). The interested reader can find more information in Gordillo *et al.* (2005) and Challiol *et al.* (2006, 2007b, 2008).

3 The Foundations: a Personal Mobile Web Server with real MVC

We created a lightweight platform for developing mobile applications which combines the benefits of web applications with the access to local resources. To achieve the latter, the web server is installed in the device itself, effectively becoming a personal web server. As a result, we display the application in a familiar environment (the web browser), we can use external services (by mashing up web components) and have access the local device out of the browser sandbox.

To build desktop-like interfaces that are easy to develop and maintain we created Meteoroid (Fernandez *et al.*, 2009), an MVC framework for developing web applications in the same way native GUI applications are written. Using this approach, the application model resides in the server and is completely decoupled from the view displayed in the web browser.

While we acknowledge that having a server running in a PDA requires the user to perform an extra installation step, we consider that the associated benefits of having a full fledged application environment outweighs this particular disadvantage. This is specially true in the case of our personal web server, since the only installation required is to copy a couple of files to the PDA. Also, as we will show later, this server can be used to host multiple applications.

3.1 Problem statement

HTTP[3] has long been the standard communication protocol for the web, fitting its purpose particularly well for static web sites. However, as web applications started to appear, more dynamic interactions were needed. As an example consider an on-line newspaper that keeps track of a tennis match or a web instant messenger. In those applications a model resides in the web server and the page acts as a view of it. When the model changes in the server, we would like to see it immediately reflected in the view. Unfortunately, the HTTP protocol is not suited for this kind of interaction since the request must be started by the browser and not the server. In other words, there is no way of notifying the client that there has been a change in the server.

Even though Netscape[4] started to research about this in 1995, not much attention has been paid to this problem. Up to date there are still basically two main approaches to solve this issue:

- Client Pull, where the web browser is constantly asking for new data in the server, essentially implementing a busy-waiting loop. The response may be the whole page or only a portion of it, obtained by issuing an Ajax[5] request and manipulating the DOM tree[6].

[3] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html`

[4] `http://256.com/gray/docs/netscape/pushpull.html`

[5] `http://www.adaptivepath.com/ideas/essays/archives/000385.php`

[6] `http://www.w3.org/DOM`

- Server Push, where an endless communication channel is established between the server and the client. By using this channel the server can communicate with the browser when changes arise, without a client request.

Comet (Crane & McCarthy, 2008), which is actually a set of possible techniques for server-browser communication, combines the Server Push approach with Ajax. By doing so, the server is able to send events and data to the client (through Server Push) while the later can perform asynchronous requests (through Ajax) to update only the required parts of the web page. Even though this approach does not support many clients concurrently (350 according to Bozdag *et al.*, 2007) it certainly allows to develop web applications that resemble desktop ones, since the only missing part up to date (sending events from the model to the view) is now solved. In our particular case, since the server is dedicated to a single client, the amount of concurrent connections is not an issue.

3.2 Security and web applications

Accessing to local resources such as GPS devices, the file system or even the browser history is a must for almost any UbiComp application. However, a standard web application is not allowed to do so due to security measures. These security restrictions appeared in the early web days as a way to protect the user when executing code obtained from an external server. This code can be either requested and executed by a plugin (e.g. Java Applets) or can be embedded in the web page and executed by the browser itself (e.g. Javascript). In any case, the problem we are facing is that the client has to execute code in his device that is coming from an unknown (potentially malicious) source. To overcome this problem, two main approaches appeared and are still in use today: *Same origin policy* and the *Sandbox model*. Since there is yet no accepted standard, each web browser (or browser plug-in) has its own implementation and policies. For example, Mozilla-based browsers define the same origin policy for Javascript as: 'The same origin policy prevents a document or script loaded from one origin from getting or setting properties of a document from another origin (…) Mozilla considers two pages to have the same origin if the protocol, port (if one is specified), and host are the same for both pages'[7].

In turn, the sandbox model approach creates a special environment where the mobile code can be safely executed. In this environment, the executed code does not have the rights to access almost any Operating System resources, thus making it theoretically impossible to damage the client device. As we previously stated, each browser implements it in its own way:

Javascript in Mozilla-based browsers. Besides the same origin policy, a script cannot access any browser or Operating System resource directly. To overcome this restriction the script must be signed[8].

Internet Explorer. Security in Explorer is based in the idea of Security Zones[9], which are actually five (Local Intranet, Trusted Sites, Restricted Sites, Internet and Local Machine). Each zone has different privileges assigned, which are inherited by the mobile code (Javascript, .Net, Java, etc.).

Chromium-based browsers. A Chromium browser is separated in two modules, the *browser process* and the *rendering engine* (Barth *et al.*, 2008). The former has the same privileges as the user running it and the latter is executed without any privileges in a sandbox model. The security in the browser is based on the fact that the rendering engine process is the one in charge of interpreting HTML and executing Javascript code.

[7] https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript

[8] http://www.mozilla.org/projects/security/components/signed-scripts.html

[9] http://www.microsoft.com/windows/ie/ie6/using/howto/security/settings.mspx

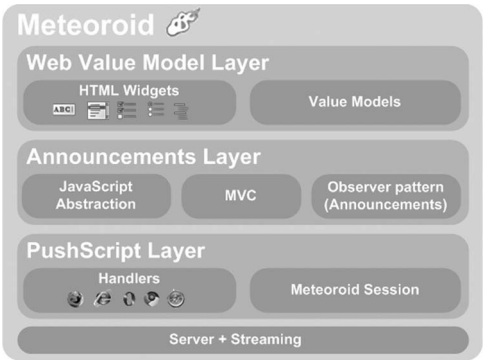

Figure 1 Meteoroid layered design

Java applets[10] also use a sandbox model to avoid evil code executing on the client device. However, signed applets can be executed outside the sandbox, accessing the host file system or hardware devices.

In our approach we use a personal web server in the device itself, where the application model resides. From the Operating System point of view the personal web server is just another application, which can freely access the device's resources (e.g. the GPS receiver).

3.3 Meteoroid in a nutshell

In our platform we heavily rely on Meteoroid (Fernandez *et al*., 2009), a Comet implementation built on top of the Seaside (Ducasse *et al*., 2007) web application framework. By using Meteoroid the server can send events with information to the client (i.e. the web page), which in turn can update the required parts. The framework is designed in three loosely coupled layers:

Push-Script layer. Handles the low-level Javascript functionality, including cross-browser issues and connection techniques. It provides the upper layer the functionality to create a Comet connection between the client and the server and a service to send arbitrary Javascript code to be executed on the client.

Announcements layer. Adds an observer-like mechanism to perform common update operations (e.g. replace a DOM object) based on events triggered by domain objects.

Web Value Model layer. Provides high-level components that act as standard desktop widgets. This allows the developer to bind a widget (e.g. a cell of a table) to a model (e.g. the value of a stock) so that when the model changes the widget is automatically updated. This layer realizes the full MVC pattern between the server and the client.

To complete the high-level description, we next show a schema of the layered approach used in Meteoroid (see Figure 1). It should also be noted that since our implementation is compatible with the native desktop GUI framework, the exact same application model can have both a native and web GUI without any conflicts.

3.4 Meteoroid close-up

The Push-Script layer provides the core, low-level behavior of Meteoroid. It has the responsibility of choosing the best technique to create a Comet-connection according to the client web browser

[10] `http://java.sun.com/developer/technicalArticles/Security/applets/`

and creating a session to keep the mentioned connection alive. This layer also provides a raw service to manipulate the DOM content of the client's browser through Javascript. Even though this layer can be used independently of the upper ones, the developer must handle all the page modifications by writing hard-wired Javascript functions. Since this is a repetitive, error prone task we added an event system in an upper layer.

The second layer provides an event system based on the Announcements framework (Cincom, 2008), which is an implementation of the Observer (Gamma *et al.*, 1995) pattern. This event system allows the developer to specify an action that can alter the page's source code (HTML) as a response of an event. As an example, consider a simple application that displays the position of the user, obtained by his GPS. The method for rendering such page in Seaside is:

```
GPSWebView>>renderContentOn: html
    html paragraph: [
                    html text: 'Position'.
                    html div id: 'position';
                             with: self gpsModel position printString.
                    ].
```

This code would create a static page, since it is not programmed to be update when the user's position changes. To achieve this behavior with Meteoroid, we must implement an initialization method that reacts to the model's change by updating the div's text. For this example, we will assume that the object that represents the user's GPS receiver (i.e. *gpsModel*) throws a PositionChanged event each time the user moves. Also, notice that the div's id is 'position', which is how we bind the update callback to the page component:

```
GPSWebView>>initialize
    self
        on: PositionChanged
        of: self gpsModel
        update: 'position'
        callback: [:html | html text: self gpsModel position printString].
```

Despite the fact that the event system helps to decouple the model from the view, we can still work at a higher abstraction level. For this purpose Meteoroid adds the Web Value Model layer, which implements web-widgets around the value model concept (Coplien & Schmidt, 1995). Web widgets are basically HTML form elements (such as inputs fields, text areas, selects, etc.) which are bound to a specific model. When the model changes an event is thrown and the form element is automatically updated.

To show how the widgets are used, we will rewrite the previous GPS example, this time using web widgets. In the *#renderContentOn:* message, we replace the div element with an 'updatable' div, which is bound to a model (a value holder):

```
GPSWebView>>renderContentOn: html
    html paragraph: [
                    html text: 'Position'.
                    html divUpdateableFor: self positionHolder.
                    ].
```

We now have to create the model in the initialization of the page, which must be configured to get the information from the GPS receiver.

```
GPSWebView>>initialize
    self positionHolder: (WebValueModel
                                with: self gpsModel
                                aspect: #position
                                announcement: PositionChanged).
```

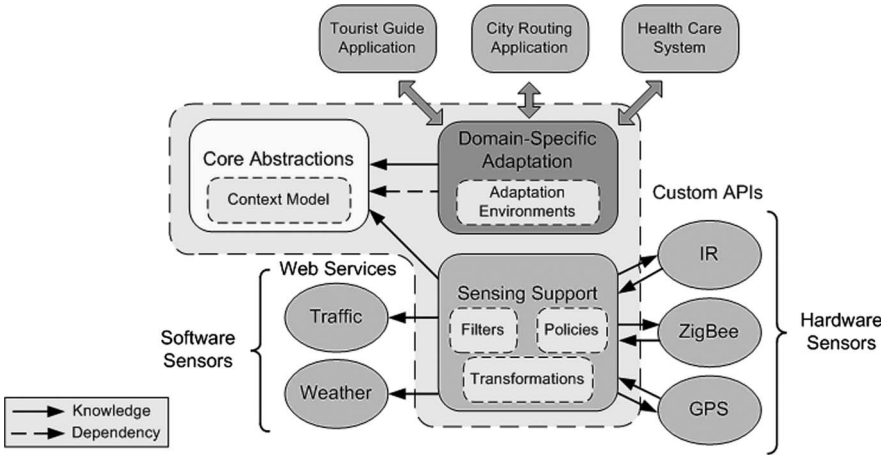

Figure 2 Overview of the Context-Aware architecture

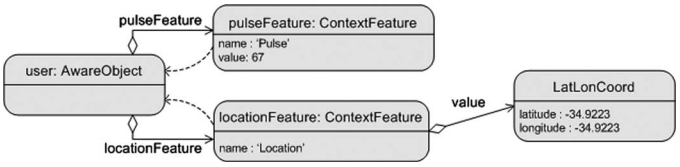

Figure 3 A simple context model

As it is shown, we do not need to explicitly state how the div contents will be rendered or updated, since the web value model knows how to do this. When the model (in this case, the object returned from the `#gpsModel` message) changes its state an event is triggered (`PositionChanged`). When the web value model captures this event, it obtains the new value (by sending the `#position` message) and pushes it into the web browser. Thus, each time the model changes, the div contents are automatically updated.

4 Context-aware architecture

In this section, we will briefly describe our architecture to develop context-aware applications. The architecture is based in a clear separation of those concerns that evolve in different directions and for different reasons. In particular we separate the context model, the sensing support and the domain-specific adaptation behavior. This design is briefly summarized in Figure 2.

4.1 Context model

Our architecture for context-aware applications is based on an object-oriented context model (Rossi *et al.*, 2005; Challiol *et al.*, 2007a; Fortier *et al.*, 2009; Perez *et al.*, 2009), where each entity whose context must be recorded is modeled as an aware-object (i.e. an object that is aware of its context). An aware object is in turn configured with a set of context features to define its context shape (note that context features can be added or removed in run-time). Thus, if we wanted to model a user and his location we would create an aware object (the user), which would have a context feature (his location). If we wanted to extend the user's context, for example, by adding his pulse for a jogging application, we would add another feature (see Figure 3).

As can be seen in Figure 3, there is a dependency relationship between an aware-object and its context features. By having this relationship, each time a context feature changes (e.g. because the

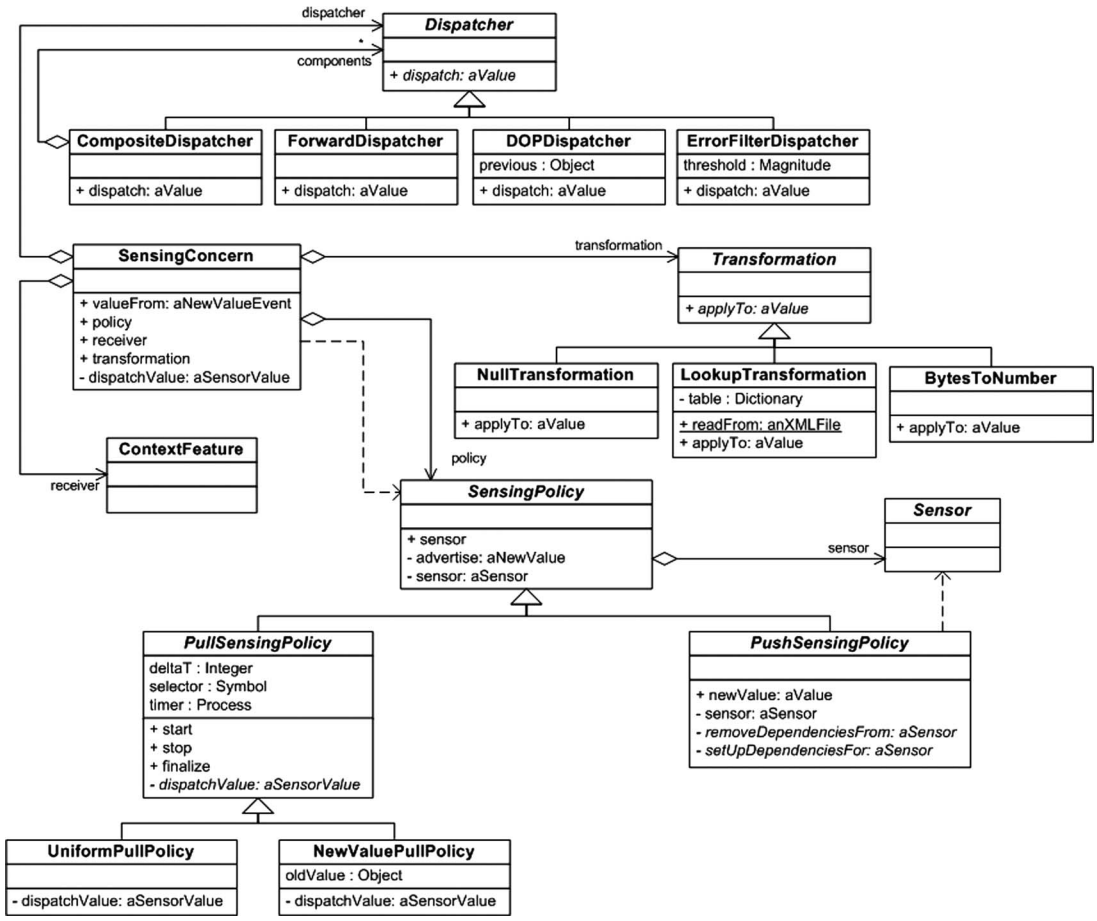

Figure 4 A class diagram of the sensing layer

user has changed its location) the aware object receives a notification. This notification is in turn converted into a context-change event, which is raised by the aware object.

To complete our model, we have created a library of context features where most common uses are already implemented and, if required, the programmer can add new ones. Examples of these pre-build features are tracking features (used to record context changes), derived features (a context feature resulting of applying a function to other features, such as inferring user activity by using his location, time of the day and schedule), model-based features (those features that rely on an existing model to make sense) or adapted features (information from an already existing model that must be treated as context).

4.2 Sensing support

Once the context model has been defined, the next step is to connect it to the required information sources. These sources can be hardware sensors (e.g. a GPS receiver) or software components (e.g. a web service). As a generalization, we call these sources sensors and we define a sensing layer that can be easily plugged into any context model. The main component of this layer is the sensing concern, which is in charge of taking input information from a sensor and updating a context feature accordingly. To do so a sensing concern is configured with three collaborators:

- A sensing policy, which can be push or pull according to the sensors characteristics.
- A dispatcher, that decides if the quality of a sensed value is suitable for the context feature (e.g. the GPS error must be less than 10 m).

- A transformation, which is used to map low-level data to semantically rich objects (e.g. applying reverse geo-coding to a GPS coordinate to obtain an address).

A sensing concern acts like a 'big brother', always watching the sensor and updating the context features as required. As we did with the context features, on top of the architecture we have built a set of commonly used dispatchers (e.g. to avoid GPS signals with high dilution of precision) and transformations (e.g. bytes to number, lookups, parser-based, etc.) that can be easily instantiated to develop new applications. In Figure 4, we show a simplified class diagram of the layer's main abstractions. The interested reader is referred to (Grigera *et al.*, 2007) for more information on the sensing layer.

To conclude this section, consider the simple context model presented previously based on a user, his location and his pulse. To effectively update the user's location as he moves we must create a sensing concern that connects the GPS receiver in the user's device to the location context feature. This is achieved by:

- Using a `NewValuePullPolicy`, since the GPS must be polled for data and we are only interested in updating the context model when the location has changed.
- Using a `DOPDispatcher`, which only allows a location to pass to the next stage if its DOP is less than a certain threshold.
- Creating a new `Transformation` subclass (`GPS2LatLon`), which removes the GPS metadata and returns a `LatLon` object. Notice that this transformation depends on the API provided by the GPS and/or underlying Operating System.

4.3 Domain-specific adaptation

Besides the GUI, the final step to build a context-aware application is to define the behavior of the application as a reaction to a context change. To do so we have isolated each application-dependent behavior into adaptation environments. An adaptation environment implements the context-aware behavior of an application, such as providing location-based services, a context-aware messenger or a mobile hypermedia system. To effectively manage context changes, each environment is configured with one or more handlers.

To illustrate our approach, consider a simple application that defines a set of information areas, so that when the user enters any of these areas a brief description appears in his screen. Assuming that the context model and sensing layer have been set up as described in the previous sections, we must:

- Create an `InformationAreasEnvironment`, which holds a collection that relates areas with their corresponding information.
- Create a handler so that when the user's location changes we check if he has entered an information area. If this is the case, an information window will appear.
- Register the user as part of the environment.

We next show the handler class that is in charge of reacting to a change of a user's location:

```
LocationChangeHandler>>handle: aContextEvent
                        from: aContextFeature
                        on: aUser
                        in: anEnvironment
(anEnvironment hasAreaFor: aContextEvent position)
    ifTrue: [anEnvironment popUpInformation: aContextEvent position]
```

We next show how to configure the information area environment by adding rectangles of (lat,long) coordinates and associating a simple text message. After the areas are loaded, the user is registered to the environment so that context changes are notified to it. Finally, when the

Figure 5 Mobile platform layout

handler is added to the environment, it is instructed to listen only for the changes in the *locationFeature* of the user.

```
| env handler |
env:= InformationAreasEnvironment new.
env
    addArea: ((−34.9219@−57.9561) corner: (−34.9236@−57.9562))
    information: 'This is the Cathedral of La Plata,...'
...
env register: user.
handler:=LocationChangeHandler new.
env addHandler: handler listenFor: user locationFeature.
```

5 The platform's basics

Our mobile platform relies on a web server running in the mobile device, which can in turn execute many applications. When the platform is launched an application manager is shown, displaying the list of available mobile applications to the user. The upper part of the screen is used as a notification area, which is a place for any application to post application-related information (e.g. the arrival of a new mail). In Figure 5(a), we show a screenshot of the application manager layout, while in Figures 5(b) an example of a Google Maps application is depicted.

When the user taps on an icon the associated application is launched, using the lower part of the display. When the user wants to quit the application he can tap in the **X** button. However, this does not mean that the application is effectively closed, since the GUI can be closed while the application is still running in background (e.g. polling a mail server for new messages). If necessary, a hidden application can post relevant information to the user in the notification area.

Since the framework is conceived to quickly implement mobile applications, creating a 'Hello World' application is almost straight forward:

- A subclass of `MobileApplication (HelloWorldApplication)` must be created.
- A title and icon must be provided for the application manager.
- A small icon must be provided for the notification area.

With these basic settings, the application will be shown in the manager and any event triggered by it will be recorded in the notification area with the application's icon. We now must program how the application will be displayed in the browser. To do so the *#renderContentOn:* message must be implemented:

```
HelloWorldApplication>>renderContentOn: html
    html paragraph: 'Hello World!'
```

A final aspect to note about the platform is that each application has its own history of visited pages and can redefine what *back* and *forward* means. In the `MobileApplication` class, the navigation strategy is decoupled from the application itself, delegating how the back and forward buttons are created. The default strategy is the `EmulatedBrowserStraetgy`, which is the standard stack-based navigation. However, we also implemented a custom strategy (`CustomBrowsingStrategy`) that is used to implement a different behavior.

At this stage, we can now put together the different pieces to build a simple application that shows the users location as he moves. To do so we combine

- Our context model.
- The Meteoroid capabilities to push information from the server to the web browser.
- The mobile platform.

In this example, we will have a user (which is an aware object) that holds his location information (a context feature). The location context feature will be bound to a sensing concern that takes the coordinates from the GPS receiver and updates the feature accordingly. A `MobileApplication` subclass (`SimpleUserLocation`) must then be implemented, defining the basic properties of a mobile application and creating an updatable div tag that is updated on a context change. The two main methods in this class are:

```
SimpleUserLocation>>initialize
    super initialize.
    self positionHolder: (WebValueModel
                                    with: self user
                                    aspect: #currentLocation
                                    announcement: ContextEvent)
```

which establishes the relationship between the aware object, his location and the event triggered on a context change. Finally, we must specify how the page is rendered:

```
SimpleUserLocation>>renderContentOn: html
    html paragraph:[
                    html text: 'Position'.
                    html divUpdateableFor: self positionHolder.
                    ].
```

The resulting application will produce the screenshot shown in Figure 6(a). However, the application can be enhanced by usign a Google Map component instead of a div tag, updating the marker's position as the user's location changes (see Figure 6(b)). To do so we rewrite the *#initialize* method (the Javascript used to move the marker is written in the *#updateGoogleMarker:announce:* method):

```
SimpleUserLocation>>initialize
   super initialize.
   self gpsMap: (GPSMobileMap fromUser: self user).
   self on: ContextEvent
        of: self user
        update: 'javascriptDivHolder'
        callback: [:html :ann | self updateGoogleMarker: html announce: ann].
```

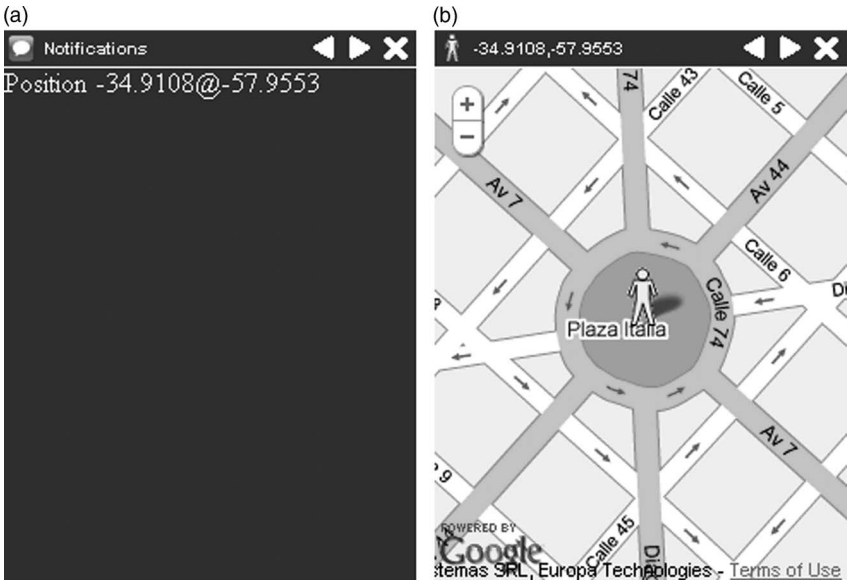

Figure 6 Using Google Maps to show the user's location

6 Mobile hypermedia

As we described in Section 2, a Mobile Hypermedia application combines digital and physical information into a single application, using the well-known hypermedia concepts (such as nodes and links). In this section, we analyze a typical mobile hypermedia application for tourism and show how it can be implemented with our platform. As a by-product we present a generic infrastructure for mobile hypermedia we built using the platform.

6.1 The basics

To represent the mobile hypermedia information in a conceptual model we use two different domains (the digital and physical one) and a type of node (the usual representation in most hypermedia applications) for each domain (thus we have digital and physical nodes).

A digital node represents the common hypermedia node, which contains descriptive information (i.e. text, images, etc.) and digital links (i.e. 'conventional' hypermedia links). On the other hand, a physical node represents a real-world object and is mostly relevant when the user is standing in front of this object (e.g. a detail in the walls of a cathedral) or when the object is involved in some activity that implies interaction in the real world (e.g. walking to that object or finding it). For this purpose, these nodes also have physical links, which express a physical relationship between this type of nodes. In case two nodes of different domains represent the same object, we can establish a mapping between them (see Figure 7).

A mobile hypermedia application can be browsed in different modes (physical and digital) and, thanks to the nodes mapping, the user can alternate through different 'views' of the same object. In the digital domain nodes are connected through digital links and the behavior is the usual one, where clicking on a link implies navigating to the target node. However, in the physical domain (where nodes are connected through physical links), clicking on a link has a different semantic, since it shows the intention of the user to *physically* navigate to the target object. This difference is quite important in terms of navigation, since it implies that the user has to move physically (e.g. walking to the target node) and that he may decide to stop doing so in the middle of his path. Thus, compared to the digital navigation, physical navigation is neither atomic nor immediate and there are actually many reasons for the user to cancel a physical navigation (getting lost, being tired, finding a more interesting place to visit, etc.). For this reason, when the user clicks on a

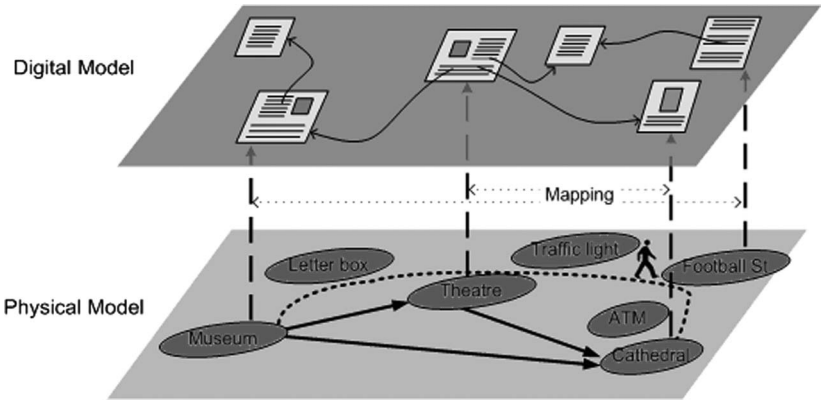

Figure 7 Nodes in a mobile hypermedia application

physical link the system must help him arriving to his intended destination (e.g. by showing a map with the path or recommending the available public transportation means).

The other main difference between the digital and physical domains is the intentionality of the navigation; in a mobile hypermedia application the physical objects are sensed automatically by the users device (e.g. by using GPS, the IR port, visual tags, etc.). Thus, if the user is walking from the Museum to the Cathedral and passes by the Theater, the physical object will be discovered. Depending on the user preferences and current activity (whether the user is physically navigating or just wandering around), the system may show information about the object and even offer the user an activity related to it (e.g. a free piano rehearsal that starts in 30 min).

At this point the navigation strategies play a central role, since the notion of 'back' and 'forward' are very different in each domain (see (Challiol *et al*., 2007b) for a detailed discussion in this subject). While the digital one uses the standard stack-based strategy, the strategy for the physical domain is not straightforward. As an example consider the following situations:

- Cities have lots of potential physical objects (government buildings, ATMs, attractions, etc.) and after a short walk, the user may have passed by many of them. To reach his starting point, the user should have to hit the back button dozens of times. Should the back function consider the physical objects that the user has accidentally passed by?
- Physical nodes can be tagged according to their characteristics (e.g. tourist points of interests, public places, etc.). Should we provide different tagged-based filters to navigate the user's physical history?
- Suppose that the user has physically navigated from the Museum to the Theater and then to the Cathedral, passing by the Football Stadium. Now the user hits back three times to go to the Museum again. Should we calculate a new path (maybe shorter and avoiding the Cathedral) or should we go through the Theater and the Stadium again?

What is interesting about these questions is that there is no single right answer, since it heavily depends on the application. Luckily, in our mobile platform the navigation is decoupled from the browser and handled by each application, which allows us not only to implement different navigation strategies, but even allow the user to select the one he finds more appropriate.

The navigation case presented before is just one example of the many issues that arise when building mobile hypermedia applications. This is due to the many concerns that have to be managed at the same time, each of them evolving for different causes.

An alternative to implementing these kind of applications from scratch is to work at a higher level of abstraction, using models than can be later derived into a concrete implementation. This approach is not only useful for the first design, but also for later maintenance. Due to the lack of space we will not explain how to the conceptual model of a mobile hypermedia applications is achieved; the interested reader can refer to Challiol *et al*. (in press) for details about this subject.

6.2 Implementing a mobile hypermedia application

To implement a mobile hypermedia application we must design a domain model that is able to handle the different navigation concerns. For this example, as we only focus on the digital and physical concerns, the discussion is simplified. The basic idea in our approach is to model each navigation concern with its specific nodes and navigation semantics. This is particularly easy in our platforms for two main reasons:

- The web application framework is component-based, which means that the html generation is encapsulated in objects. Also, links are associated with actions triggered in the server on components directly.
- The mobile platform delegates the navigation logic to each application, thus we can decide how to react to back and forward commands according to the current navigation domain.

The mobile hypermedia model takes a navigational model and maps its components to nodes. For each navigation domain a separate node graph is created. For those cases where a digital node has a physical counterpart a relationship is established, thus allowing the user to jump from one domain into another. The nodes that are related to application domain objects are then linked to them, so that domain-specific services can be called (e.g. in a tourist application the museum may offer the service of buying on-line tickets).

Having the mobile hypermedia application model in place, the next step is to create the views of each navigation domain. For the digital domain this is straightforward, since the node description is rendered as html and the associations to other objects as standard hypermedia links. For the physical domain a Google Map is shown as a general view. According to the map viewport, the nearby physical nodes are shown. If the user is physically near a physical object, the application assumes that the user is in front of that object[11]. In this case, the association between the node and other physical nodes are rendered as hyperlinks in the bottom of the map. However, clicking on a hyperlink does not take the user immediately to the node, but shows the path from the users location to that physical object. At this point the user has started a physical navigation, which is modeled as an object that decouples its current State (Gamma *et al.*, 1995) in a hierarchy of possible situations:

- InFrontOf. The user is standing in the buffer zone around a physical object.
- Started. The user has selected a target object and the path has been calculated.
- InCourse. After the path has been calculated the user is moving towards his target in the suggested path.
- OutOfCourse. The user has selected a target object and the path has been calculated, but he is moving outside the planned path.
- Canceled. The user has decided to cancel his navigation.
- Ended. The user has reached his target object.
- NotSpecified. The user is moving, but no target object has been selected to navigate to.

At any time the user can decide to switch between the physical and digital view, which is possible since the physical nodes are mapped to their digital counterparts. We next show a couple of screenshots of the mobile hypermedia basic application (Figure 8).

In Figure 9 we show a simplified class diagram of the generic model for a mobile application. Note that all classes in the diagram are completely independent of the concrete application we are building (e.g. tourist cities guide, augmented reality museum, etc). In this way, this software substrate becomes itself a framework for specific mobile hypermedia applications, which can be instantiated in different applications.

[11] To determine this we use a buffer zone around the user's location. If more than one physical object is inside this zone, the nearest to the user is considered the current physical node.

Figure 8 A basic mobile hypermedia application

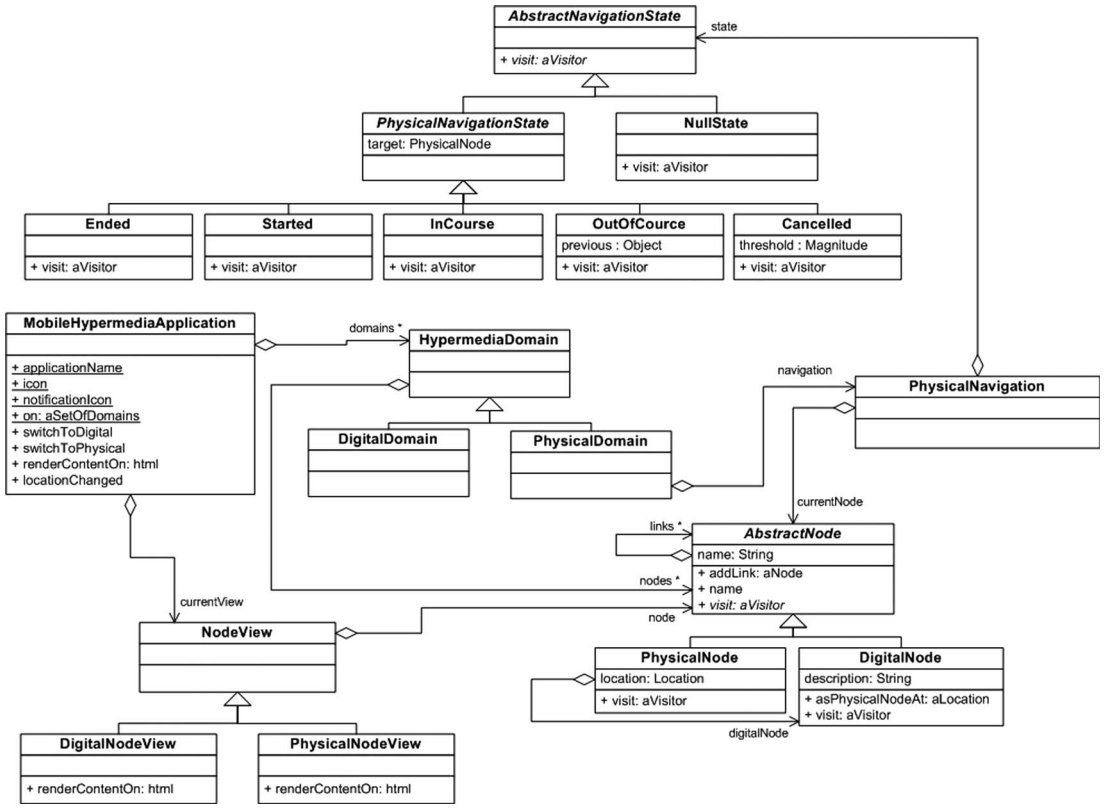

Figure 9 Class diagram of a mobile hypermedia application

Concrete application nodes (e.g. the cathedral, monuments, etc.) become instances of their corresponding classes in the application domain model. By means of the `DigitalNode` and `PhysicalNode` classes these instances are integrated into the mobile-hypermedia model.

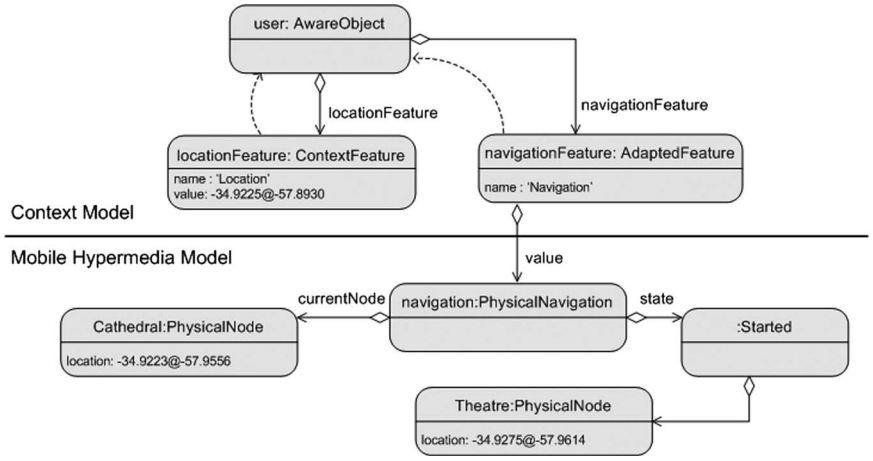

Figure 10 Context model for an extended mobile hypermedia application

6.3 Adding context-aware behavior

The first step to create context-dependent behavior in our approach is to define a context model. To do so we will use an aware object to represent the user and a location feature to model his location. However, we can also take advantage of the navigation information already present in the hypermedia application and use it as part of the context model. To do so a derived feature is added to the aware object, presenting application information as part of the context model. Figure 10 depicts this situation.

Having this basic information as context, we can start to add context-dependent behavior to the application. To combine the context model and the context-dependent behavior, a `MobileHypermediaEnvironment` must be created and the user has to be registered to it.

In this scenario, maybe one of the easiest behaviors to implement is to give a visual (or audible) cue to warn the user when he has left the proposed path while navigating from a physical object to another. To do so a new handler class is created (`OutOfPathWarning`) implementing the abstract message used to handle context changes:

```
OutOfPathWarning>>handle: anEvent from: aCF on: aUser in: anEnvironment

(aUser navigationFeature isNavigating
  and: [(aUser navigationFeature includesPoint: aUser location) not])
      ifTrue: [self warnUser].
```

Where the *#warnUser* message can send a notification to the general notification area or play a warning sound to alert the user in case he is not watching his device. To add this new behavior, we just add the handle to the environment and configure it to only listen for changes in the users location.

```
warningHandler:= OutOfPathWarning new.
environment addHandler: warningHandler listenFor: user locationFeature.
```

Another interesting behavior would be to listen for changes in the navigation state, so that when the user plans a new physical navigation the system can perform some sort of recommendation (e.g. if it is lunch time and the trip is expected to last more than an hour, offer the user to recalculate a new path that passes by a restaurant).

To complete this section we will now show how to integrate location-based services (LBS) to the mobile hypermedia application. As part of our context-aware prototypes, we developed a small LBS framework that defines the basic abstractions present in any LBS application (such as services and service consumers) and creates a new environment class (`ServicesEnvironment`), which coordinates these abstractions (adding and removing services, the activation and

deactivation of services, etc.). With these abstractions different applications can be easily created, like friend finders, location-based information systems, location-based messengers, etc. In our prototypes these applications have, in average, four classes and nine methods per class. As a concrete example consider a location-based messenger, which only delivers a message to another user if he is in a certain range. In case the recipient is outside the physical area the message is stored in a queue and delivered when the recipient is back in the range. Also the sender is informed of the relevant events, getting a notification when the message has been finally delivered.

An interesting thing about the LBS framework is that it poses a single restriction: to add an aware object (i.e. a service consumer) to the environment it must have a location feature. As expected, the user's location model must be compatible with the one used to specify the services description (in other words, if the location feature uses a (latitude, longitude) pair we can not express the services availability with a symbolic location model (Leonhardt, 1998)). Since the aware object in the mobile hypermedia application naturally supports this prerequisite, it can be directly added to any LBS application.

Even though the integration of new context-dependent behavior can be done without any rewriting, extending the GUI is not a simple task. At this time, the mobile hypermedia GUI would have to be modified and extended to support the new features. From an architectural point of view, this is still a weak point that must be addressed, since it means actually changing a part of an existing application.

7 Conclusions and further work

In the introduction of this paper, we highlighted the advantages of web applications and the new trends towards UbiComp applications and mobile software. We consider that in the years to come it is natural that these technologies converge to create the next generation of mobile applications. However, web applications have their own restrictions, due to the sandbox model and the dependency on a wireless connection, which is extremely important for UbiComp software. To solve these problems, we conceived a platform that takes the best of both worlds by developing web applications that have access to local resources and that can gracefully degrade to work without remote resources like any native mobile application.

In this paper, we have combined a personal web server with an application framework to create web applications that run locally (thus having the ability to access local resources) with the natural facilities to access external services (such as Google Maps). The application framework has been extended with Meteoroid, a Comet-based framework that provides server-client communication and desktop-like web widgets, which are automatically updated when their model changes. With this infrastructure, web applications can be developed using the same approach that is used to develop native mobile applications.

Using these tools as a basis we created a development platform by merging them with our context-aware architecture, which is based in loosely-coupled components and a rich context-model. As a case study, we have shown how a mobile hypermedia application can be developed from scratch and later on incrementally extended to support context-aware behavior.

We next summarize the main contributions of the paper:

- We explained how to create mobile web applications with a familiar look and feel, while keeping native access to local resources.
- We described a framework for easily creating and maintaining web applications that require server changes to be immediately propagated to the client.
- We presented a flexible architecture for developing context-aware applications.
- We showed how both designs can be combined for developing context-aware mobile applications.

Regarding our further work in the mobile platform area, there are many topics that still need to be addressed. In the first place, every application of our platform runs in the same operating

system process and is internally scheduled by using green threads. This exposes the entire platform to failure in case an application has a hard failure. A possible alternative that we will investigate is to launch each application in its own separate process; however, this would mean an important refactoring in our architecture. A second issue that must be addressed is to lower the platform's memory footprint. Our current implementation of the platform uses an average of 24 MB in RAM, which is about three times the memory used by the Opera mobile web browser and two times the memory used by Adobe's Flash plugin. While most modern PDAs support this load, we consider that better results can be achieved.

Focusing on the implemented toolkit, even though we have modeled most of the html form tags as web widgets, we must add more complex components to it, so that interesting applications can be quickly prototyped without explicitly handling events or writing custom Javascript code. Finally, as we stated in the previous section, we are researching in a mechanism that allows a GUI to be extended to support new context-aware behavior.

In the context-aware area, we are currently revisiting the context model implementation and distilling a set of conceptual guidelines for context modeling (a first approach can be found in (Perez *et al*., 2009)). Also, since the mobile web platform is a work in progress we still have to port our previous prototypes to the new platform.

References

Barth, A., Collin, J., Charles, R. & Team, G. C. 2008. *The security architecture of the chromium browser*. Technical Report.

Billsus, D., Brunk, C. A., Evans, C., Gladish, B. & Pazzani, M. 2002. Adaptive interfaces for ubiquitous web access. *Communications of the ACM* **45**(5), 34–38. ISSN 0001-0782.

Bouvin, N. O., Christensen, B. G., Grønbæk, K. & Hansen, F. A. 2003. Hycon: a framework for context-aware mobile hypermedia. *The New Review of Hypermedia and Multimedia* **9**(1), 59–88.

Bozdag, E., Mesbah, A. & van Deursen, A. 2007. A comparison of push and pull techniques for ajax. CoRR, abs/0706.3984. In *Proceedings of WISE*, 15 – 22.

Bricon-Souf, N. & Newman, C. 2006. Context awareness in health care: review. *International Journal of Medical Informatics* **76**(1), 2–12.

Ceri, S., Daniel, F., Matera, M. & Facca, F. M. 2007. Model-driven development of context-aware web applications. *ACM Transactions Internet Technology* **7**(1), 1–32.

Ceri, S., Fraternali, P. & Bongio, A. 2000. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks* **33**(1–6), 137–157.

Challiol, C., Fortier, A., Gordillo, S. E. & Rossi, G. 2007a. A flexible architecture for context-aware physical hypermedia. In *Proceedings of DEXA Workshops*, 590–594. Regensburg, Germany.

Challiol, C., Muñoz, A., Rossi, G., Gordillo, S. E., Fortier, A. & Laurini, R. 2007b. Browsing semantics in context-aware mobile hypermedia. In *Proceedings of OTM Workshops*, 211–221. Vilamoura, Portugal.

Challiol, C., Fortier, A., Gordillo, S. E. & Rossi, G. 2008. Model-based concerns mashups for mobile hypermedia. In *Proceedings of MoMM*, 170–177. Linz, Austria.

Challiol, C., Fortier, A., Gordillo, S. E. & Rossi, G. 2008. Model-based concerns mashups for mobile hypermedia. In *Proceedings of MoMM*, 170–177. Linz, Austria.

Challiol, C., Rossi, G., Gordillo, S. E. & Cristófolo, V. D. 2006. Systematic development of physical hypermedia applications. *IJWIS* **2**(3/4), 232–246.

Challiol, C., Rossi, G., Gordillo, S. E. & Fortier, A. Separation of concerns in mobile hypermedia: architectural and modeling issues. In *Handbook of Research on Mobile Software Engineering: Design, Implementation and Emergent Applications*, Alencar, P. & Cowan, D. (eds). IGI Global, in press.

Chang, P.-H. & Agha, G. 2007. Towards context-aware web applications. In *Proceedings of DAIS*, 239–252. Paphos, Cyprus.

Cheverst, K., Mitchell, K. & Davies, N. 2002. The role of adaptive hypermedia in a context-aware tourist guide. *Communications of the ACM* **45**(5), 47–51.

Cincom. 2008. Application developer's guide (chapter 11).

Coplien, J. O. & Schmidt, D. C. (eds) 1995. *Pattern Languages of Program Design*. Software Pattern Series. Addison-Wesley.

Crane, D. & McCarthy, P. (eds) 2008. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Software Pattern Series. APRESS.

Daniel, F. & Matera, M. 2008. Mashing up context-aware web applications: a component-based development approach. In *Proceedings of WISE*, 250–263. Auckland, New Zealand.

Dey, A. K. 2000. *Providing architectural support for building context-aware applications*. PhD thesis, Georgia Institute of Technology. Director-Gregory D. Abowd.

Ducasse, S., Lienhard, A. & Renggli, L. 2007. Seaside: a flexible environment for building dynamic web applications. *IEEE Software* **24**(5), 56–63. ISSN 0740-7459.

Edwards, W. K. & Grinter, R. E. 2001. At home with ubiquitous computing: seven challenges. In *Proceedings of Ubicomp*, 256–272. Atlanta Georgia, USA.

Fernandez, J. L., Robles, S., Fortier, A., Ducasse, S., Rossi, G. & Gordillo, S. 2009. Meteoroid: towards a real mvc for the web. IWST.

Fortier, A., Rossi, G., Gordillo, S. E. & Challiol, C. 2010. Dealing with variability in context-aware mobile software. *Journal of Systems and Software* **8**(6), 915–936. ISSN 0164-1212.

Gamma, E., Helm, R. & Johnson, R. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

Gordillo, S. E., Rossi, G. & Lyardet, F. 2005. Modeling physical hypermedia applications. In *SAINT Workshops*, 410–413.

Grigera, J., Fortier, A., Rossi, G. & Gordillo, S. E. 2007. A modular architecture for context sensing. In *Proceedings of AINA Workshops*, 147–152. Niagara Falls, Canada.

Grønbæk, K., Kristensen, J. F., Ørbæk, P. & Eriksen, M. A. 2003. ''Physical hypermedia'': organising collections of mixed physical and digital material. In *Hypertext*, 10–19. Nottingham, United Kingdom.

Harper, S., Goble, C.A. & Pettitt, S. 2004. Proximity: walking the link. *Journal of Digital Information* **5**(1), Article No. 236.

Lamming, M. & Flynn, M. 1994. Forget-me-not: intimate computing in support of human memory. In *Proceedings of FRIEND21 Symposium on Next Generation Human Interfaces*.

Leonhardt, U. 1998. *Supporting Location-Awareness in Open Distributed Systems*. PhD thesis, Department of Computing, Imperial College.

Nichols, J., Hua, Z. & Barton, J. 2008. Highlight: a system for creating and deploying mobile web applications. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, 249–258. New York, NY, USA. ACM.

Pascoe, J. 1997. The stick-e note architecture: extending the interface beyond the user. In *Proceedings of IUI*, 261–264.

Pascoe, J., Ryan, N. & Morse, D. 2000. Using while moving: HCI issues in fieldwork environments. *ACM Transactions on Computer–Human Interaction* **7**(3), 417–437. ISSN 1073-0516. New York, USA.

Pederson, T., Ardito, C., Bottoni, P. & Costabile, M. F. 2008. A general-purpose context modeling architecture for adaptive mobile services. In *Proceedings of ER Workshops*, 208–217. Barcelona, Spain.

Perez, E., Fortier, A., Rossi, G. & Gordillo, S. 2009. Rethinking context models. In *Proceedings of OTM Workshops*, 78–87. Vilamoura, Portugal.

Rao, B. & Minakakis, L. 2003. Evolution of mobile location-based services. *Commun. ACM* **46**(12), 61–65.

Rekimoto, J., Ayatsuka, Y. & Hayashi, K. 1998. Augment-able reality: situated communication through physical and digital spaces. In *Proceedings of ISWC*, 68–75. Pittsburgh, USA.

Rossi, G., Gordillo, S. E. & Fortier, A. 2005. Seamless engineering of location-aware services. In *Proceedings of OTM Workshops*, 176–185. Agia Napa, Cyprus.

Schilit, B. N., Hilbert, D. M. & Trevor, J. 2002. Context-aware communication. *Wireless Communications, IEEE* **9**(5), 46–54.

Simon, R. & Fröhlich, P. 2007. A mobile application framework for the geospatial web. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web*, 381–390. New York, NY, USA. ACM. ISBN 978-1-59593-654-7. In Williamson, C. L., Zurko, M. E., Patel-Schneider, P. F. & Shenoy, P. J. (eds).

Weiser, M. 1995. The computer for the 21st century. Human–computer interaction: toward the year 2000, 933–940. Baecker, R., Grudin, J., Buxton, W. & Greenberg, S. (eds). Morgan Kaufmann Publishers.