RI 10009 July 28, 2010 Computer Science

IBM Research Report

A Middleware Framework for Mashing Device and Telecom Features with the Web

Vikas Agarwal, Sunil Goyal, Sumit Mittal, Sougata Mukherjea
IBM Research Division
IBM Research - India
4, Block - C, Institutional Area, Vasant Kunj
New Delhi - 110070. India.

John Ponzo, Fenil Shah
IBM Research Division
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne
NY, USA - 10532.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

A Middleware Framework for Mashing Device and Telecom Features with the Web

Vikas Agarwal[1], Sunil Goyal[1], Sumit Mittal[1], Sougata Mukherjea[1], John Ponzo[2], Fenil Shah[2]

[1]*IBM Research - India, 4, Block - C, Institutional Area, Vasant Kunj, New Delhi, India - 110 070*

{avikas, gsunil, sumittal, smukherj}@in.ibm.com

[2]*IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY, USA - 10532*

{jponzo, fenils}@us.ibm.com

Abstract. Evolution of Web browser functionality on mobile devices is the driving force for 'mobile mashups', whereby content rendered on a device is amalgamated from multiple Web sources. From richness perspective, such mashups can be enhanced to incorporate features that are unique to a mobile setting - (1) native device features, such as location and calendar information, camera, Bluetooth, etc. available on a smart mobile platform, and (2) core Telecom network functionality, such as SMS and Third Party Call Control, exposed as services in a converged IP/Web network setup. Although numerous techniques exist in the Web domain for creating mashups, these are insufficient to utilize a three-dimensional setting present in the mobile domain - comprising of the Web, native device features and Telecom services. In this paper, we first establish that middleware support is required for this purpose, both on the server side dealing with processing and integration of content, as well as on the device side dealing with rendering, device integration, Web service invocation, and execution. Moreover, we characterize how various components in this middleware ensure portability and adaptation of mashups across different devices. Based on our approach, we implement a mobile mashup framework for three popular platforms - iPhone, Android and Nokia S60, and evaluate it against several software engineering principles and performance metrics.

Keywords - Mashups, Mobile Web, Telecom

1 Introduction

A large number of Web accessible services publish their data and offerings in the form of open interfaces for the developer community. 'Mashups' are applications that create new innovative services by integrating such interfaces from multiple Web sources, and have become very popular in recent years. Although in the beginning, mashups were primarily created for desktop-based browsers, they are quickly gaining traction in the mobile domain as well. Technology-wise, adoption of such *mobile mashups* is driven by the evolution of Web browsers on the mobile device. Most modern smart phones today have browsers that are

HTML and JavaScript standards compliant, and provide a rich, powerful Web browsing experience to mobile users. With rapid enhancements in computing power, memory, display and other features of mobile phones, and with continuous improvement in mobile network bandwidth, mobile mashups bear the potential of being as successful as the desktop ones.

To help developers rapidly create rich mobile applications, popular platform vendors like Nokia, Blackberry and Android offer extensive middleware support, such as access to the underlying operating system, programmable constructs, useful libraries and tools, etc. In addition, most platforms also expose interfaces that provide access to, from within the applications, a variety of features available on the mobile handset - information (user's contacts, calendar, geographic location, etc.) as well as functionality (making calls, sending SMSes, using the camera, etc.) [1]. Richer applications for a mobile platform can be composed by combining application logic with these features. For example, using the location information available on the mobile phone, one can design a number of interesting location-based applications - directory services, workforce management solutions, etc. Similarly, camera on the phone can be used in various mobile applications related to bar-code scanning and decoding. From the perspective of mobile mashups, it is desirable that such device features be integrated with other Web offerings to enhance the entire mashup experience of a mobile user.

Another trend relevant for the mashup setting is the willingness of Telecom operators to move from a *walled-garden* model to an *open-garden* model [12]. In essence, with Telecom markets reaching saturation, revenues from voice calls is decreasing. Moreover, alternate services from an operator such as games, news, ring tones, etc. are also facing strong competition from similar offerings provided by Internet Content providers. Telecom operators, however, are still unmatched in terms of their core functionalities of Location, Presence, Call Control, etc., characterized further by carrier-grade Quality-of-Service (QoS) and high availability. Therefore, a potential revenue channel for the operators is to 'open' these functionalities as services to developers for creating new innovative applications. For example, Location and Presence information from Telecom can be clubbed with offerings such as Facebook and MySpace to provide an enriched social networking scenario. When applied to a mobile mashup setting, the opening up of Telecom networks provides avenues to developers for mashing these features with other services on the mobile Web.

Numerous tools and technologies exist [2, 11, 19] that help developers create mashup applications for the desktop world. In this paper, we argue that these efforts are not sufficient in the context of a three-dimensional setting present in the mobile domain - native device features, Telecom services and Web-based offerings. We outline a systematic middleware approach to mash these dimensions on the mobile browser, taking into account different aspects of a mashup - processing and integration of content, along with rendering, device integration, Web service invocation, and execution. In particular, our contributions can be summarized as:

- We establish that middleware support is required, both on the server side as well as the device side, to enable incorporation of device and Telecom features

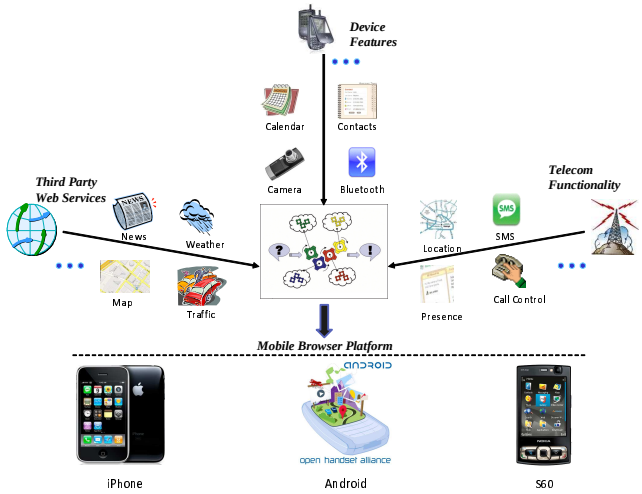


Fig. 1. Setting for Mobile Mashups

in a mashup. Moreover, unlike existing mashup models in which the client is responsible only for rendering the content it receives from the server, our approach allows the device to participate as an active component.

– We characterize how various components in our middleware ensure portability and adaptation of mashups across different devices, while hiding the heterogeneity of mobile platforms and complexity of Telecom network protocols.
– We implement a mobile mashup framework for three modern platforms - iPhone, Android, and Nokia S60, and evaluate it against several software engineering as well as performance metrics.

The rest of this paper is organized as follows. In Section 2, we motivate the reader towards challenges in developing rich mobile mashups. We then present our middleware framework for creating mobile mashups in Section 3, followed by its implementation for three mobile platforms in Section 4. In Section 5, we evaluate our framework, while in Section 6, we discuss some related work. Finally, Section 7 draws our conclusions and provides pointers for future work.

2 Motivation

Figure 1 shows the unique convergence of Telecom services, device features, and Web offerings on the mobile browser of different platforms such as Android, iPhone and Nokia S60. To appreciate the power of this setting, consider the richness it can impart to a social networking mashup on the mobile device. To start with, current GPS-based 'location' can be obtained from the device and combined with contacts on the phone 'address book' to project friends on a Web-based map service such as Google Maps. Similarly, 'camera' functionality can be embedded within the mashup to let users click and share pictures with others in real-time. With respect

to Telecom services, a Presence Server hosted by a Telecom provider can offer current 'Presence' (availability, off-hook, on-hook status, etc.) of friends that can be used effectively during collaboration. Moreover, communication features from Telecom, like the 'Third Party Call Control' service, messaging service, etc. can also be incorporated. It is important to realize that the advantage of a converged setting holds not only for new mashups being created, but also for existing Web applications. For example, a Web chat client, such as Google Talk, can be enriched by using device features like GPS, Camera, etc. as well as Telecom offerings such as Call, SMS/MMS, etc.

Although a number of mashup technologies exist in the Web domain, they currently lack capability to utilize the three dimensional convergence present in the mobile domain. We argue that the following issues need to be addressed:

1) **Two-way Mashup Model**. In current mashups, the client is responsible only for rendering the content it receives from the server. However, for mobile mashups, the device can act as an *active component*, augmenting a mashup with its own information and features. In essence, a client[1]-server architecture for mashups requires middleware enhancements to provision (a) a *two-way* flow of information - from device to server, apart from server to device, and (b) a *two-way* mashup support where the device and Telecom features can be used both on the server (during processing and content integration) and the client (during mashup rendering and execution). For the first enhancement, one needs to allow *feeds* of device information to the server, while provisioning the server to receive those feeds. For the second enhancement, one requires different set of mashable interfaces for each feature from the Telecom and device domains. As far as Telecom services are concerned, this means exposing interfaces in various programming languages - Java, PHP, C, etc. for inclusion in the server-side mashup, while for the device-side mashup, REST-based interfaces invocable through JavaScript should be provided. For device features, on the other hand, interfaces are required in JavaScript format to allow inclusion during mashup execution on the device.

2) **Device Integration**. To highlight this issue, let us examine the interfaces for adding location proximity alerts on Android and S60 platforms:

On Android, the exposed interface is

- *LocationManager.addProximityAlert (double latitude, double longitude, float radius, long expiration, Intent intent) throws SecurityException*

On S60, on the other hand, the corresponding interface is

- *LocationProvider.addProximityListener (ProximityListener listener, Coordinates coordinates, float proximityRadius) throws SecurityException, LocationException, IllegalArgumentException, NullPointerException*

As evident, the APIs on S60 and Android are available in native Java language, while for mashups we require interfaces that can be embedded in the Web programming model, i.e. JavaScript. Therefore, on a given mobile platform, we first need a *bridge* that takes the native APIs and exposes them in a Web mashable form. We emphasize that such a bridge needs to have two distinct attributes. The primary attribute corresponds to support for a *bidirectional communication*

[1] in this case, the device

between the mobile browser and the native APIs, so that a) input objects required for invoking a native feature can be passed from within the mashup, through the browser, to the underlying platform, and b) output objects available as a result of native invocation can be marshalled back to the mashup through the browser. The second attribute of this bridge pertains to provision of proper *callback handling* - callbacks generated in the native context, for example, those related to proximity alerts on a mobile platform, should lead to appropriate signaling and notification in the invoking browser context of a mashup.

Another problem that needs to be addressed by this bridge is the immense *fragmentation* in syntax and semantics of interfaces across various platforms. For example, on Android, registration of a location alert leads to two sets of events - one for the device entering a proximity region, and the other associated with exiting. Further, multiple such events are generated, governed by an expiration period. On S60, on the other hand, only one event is generated - when the device enters a proximity for the first time. On the syntactic front, Android uses the *Intent* and *IntentReceiver* objects to realize a callback function for the alert mechanism, whereas on S60, one needs to define an implementation for the abstract *ProximityListener* class. Similarly, there is diversity in terms of the name of the interface, as well as in the 'name', 'data type' and 'ordering' of attached parameters. Finally, the interfaces also differ in the set of exceptions thrown by the underlying platforms. Broadly speaking, the requirement for de-fragmentation arises because there is an inherent desire by platform vendors to differentiate their offerings from others, even though standardization efforts such as J2ME exist in the mobile domain.

3) **Network Protocols**. Our third issue pertains to handling of complex protocols associated with features of the Telecom domain. A number of legacy protocols exist in this domain, while new standards continue to be drafted and absorbed by different practitioners. For example, to help developers access Telecom services without knowing the underlying protocol details, a standard called Parlay-X [14] has been developed that exposes Web service interface for core Telecom functionalities. On similar lines, IP Multimedia Subsystem (IMS) [7] provides a reference framework to expose these functionalities as services to Web-engineered systems using the SIP [8] standard. From a mashup creation perspective, this means that a developer has to deal with various Telecom specific protocols arising out of not only legacy implementations, but also various upcoming standards. Therefore, to reduce burden on the developer, we need an *encapsulation* model at the middleware level that hides protocol specifics from the developer. Moreover, for a given functionality, this model should enable a seamless switch among different protocols - this is especially required in scenarios where the Telecom networks are gradually evolving to move from legacy interfaces to standards like Parlay-X/SIP.

3 Mobile Mashup Framework

Figure 2 presents an overview of next generation mobile mashup framework, whereby a mashup comprises of two portions running in conjunction - one on

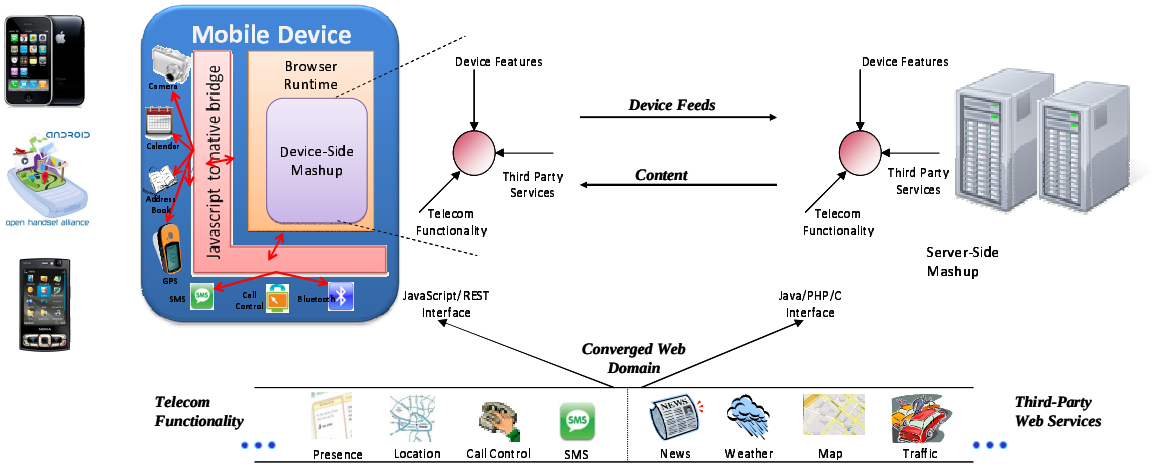


Fig. 2. Mobile Mashup Framework

the server side and the other on device. The server side mashup, as shown in the figure, executes on a Web server and deals with processing and integration of content received from each dimension. The device side portion, on the other hand, resides on the browser context [2] of different platforms, such as iPhone, Android or Nokia S60, and renders the content received from server. Through client-side scripts (JavaScript and AJAX), this portion executes Telecom functionality - SMS, Third Party Call Control, etc., Web offerings - Map information, News, etc., as well as various features of the device, such as Camera and Bluetooth. Moreover, this portion also participates actively in a mashup by feeding device information, such as Location and Calendar, to the server side component for inclusion in the processing logic.

Enabling the above framework requires middleware components, both on the device and the server. We next describe these components in more detail.

3.1 Mobile Device Middleware

As shown in Figure 3, the device mashup middleware runs on top of existing platform support, and provides an *Enhanced Browser Context* in which the client side mashups execute. In essence, the enhanced browser context uses the existing *Browser Runtime* available on a platform to render the HTML pages and also to execute the associated JavaScript code. It further uses a *Native to Mashup Bridge* to allow access to device features (such as Camera, Calendar, Contacts, etc.) using JavaScript interfaces from within a mashup. As argued in the previous section, this bridge provides a bi-directional communication capability between

[2] for this paper, browser context/runtime means either the browser itself or any component thereof that can be embedded within a mobile application to render Web content.

the browser context and the native APIs - allowing the browser context to invoke native APIs, pass inputs and receive outputs, and also the native APIs to invoke JavaScript functions in the browser context for callback support. The bridge itself can be implemented in multiple ways, - 1) by modifying the source code of an existing browser to allow access to native capabilities through JavaScript, 2) by creating a plugin for the browser that adds device features without modifying the browser source code, and 3) by embedding a browser runtime (where the browser is available as a 'class') within a native mobile application, and extending this application to enable access points for native features within mashups rendered through the runtime. Depending on the capabilities and support provided on the device platform, one or more of these techniques could be used.

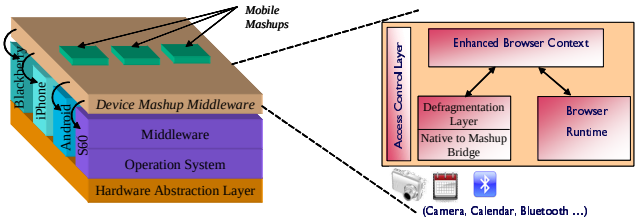


Fig. 3. Middleware Support on Device

Another component of device mashup middleware is the *De-fragmentation Layer*. This layer builds on the JavaScript interfaces exposed by Native-to-mashup bridge and exposes a consistent set of mashable interfaces across various platforms. For instance, consider the following listing that defines a uniform API across Android, iPhone, and Nokia S60 for invoking periodic location updates -

startUpdatesOnTimeChange (timeFilter, callback, errorCallback, options)

In contrast to the original APIs on these platforms, semantics of the above API as well as data structures of the parameters involved, such as `timeFilter`, are uniform across the platforms. Similarly, updates for location information, encapsulated in a `Location` object are provided through a commonly defined `callback` method, while errors are propagated with the help of a common `errorCallback` method. Note that a generic `options` parameter is also provided with the interface. Similar to what we argue in one of our earlier works [1], this parameter is optional and used to configure platform-specific attributes, such as *Criteria* in the case of S60, *Location Provider* in the case of Android, and *accuracy* for iPhone. The structure and values of this parameter are platform dependent, and therefore should be strictly used only when a developer wishes to fine-tune mashups on a particular platform. For general usage, `null` should be passed for this parameter, in which case default values for various attributes would be picked up on the corresponding platform.

Accessing a device or Telecom feature might have cost associated to it, such as sending an SMS, making a Call, etc. or might deal with sensitive personal information stored on the device, for instance Location and Calendar entries. Therefore, a component that becomes intrinsic to the entire set-up is the *Access*

Control Layer that performs the task of regulating access to these features. More specifically, this component intercepts each request from within a mashup application for invoking a feature, and performs appropriate checks to determine whether the desired access should be allowed or not. Policies for various checks are to be configured by the user when a mashup application is first accessed, and refined/altered over a period of time. These policies take into consideration different factors such as frequency of access, time of day, user's current location, etc. to take automatic decisions or prompt the user for explicit approval.

3.2 Server Side Middleware

Figure 4 shows the server side middleware required to enable our mashup framework. As depicted, this middleware consists of two parts - 1) a *Telecom block* to enable access to Telecom network functionalities, and 2) a *Device block* to receive information feeds from mobile devices and to perform device specific adaptation of mashups.

At the heart of Telecom block, we have a *Protocol Binding* component that connects to various Telecom services using the underlying network protocols. For example, in Figure 4, Location information is obtained using Parlay-X, Presence information is fetched using SIP, and the Call Control functionality is invoked using CORBA. Through these binding components, the framework removes the burden of knowing Telecom protocols on part of a mashup developer, such as session management for SIP, broker object for CORBA and SOAP headers for Parlay-X. Note that once the bindings are in place, the Telecom block provides mashable interfaces for different services in various programming languages - Java, C, C++, etc. While these interfaces can be directly used in a server side mashup, REST based interfaces are also exposed using servlet/PHP so that Telecom services can be invoked from client side mashups using JavaScript.

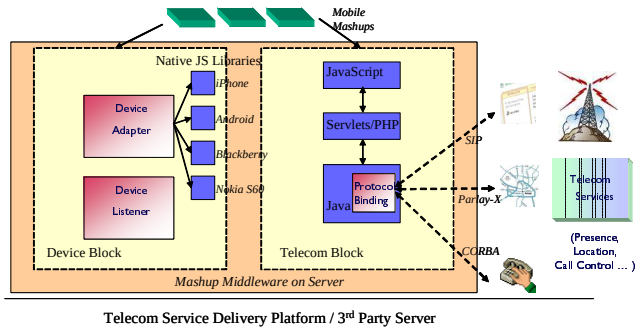


Fig. 4. Middleware Support on Server

Binding stubs also enable seamless switching between different protocols. Consider a scenario where API for fetching network location information is exposed using both SIP and Parlay-X. In SIP, this information is obtained by subscribing to a Presence Server for the presence information, and parsing the returned

document. Parlay-X, on the other hand, requires the request to be made using a SOAP envelope, and like-wise returns the location information encapsulated under SOAP. We can wrap these disparate steps under a generic *getLocation()* interface, and provision the same using corresponding stubs for Parlay-X and SIP. Properties and attributes required for Parlay-X (and similarly for SIP), such as service port information, tolerable delay, accuracy, etc. are configured using an *options* parameter similar to the device middleware model.

The device block consists of a *device listener* component that receives feeds (such as location, contact list) from a device side mashup and provides this information to the server side mashup for processing and integration. As mentioned earlier, these feeds enable the mobile device to participate actively in a mashup by helping determine what content is generated at the server side. Another component in this block is the *device adapter* that performs device specific adaptation of mashups. For example, appropriate JavaScript libraries (containing code to access device features) need to be included in a mashup page depending on the device where the mashup is being rendered. Also, the look and feel of a mashup can be adjusted based on the device properties, such as screen size, resolution, etc. by including appropriate CSS files. Further, any other device specific adaptation, for example, altering the layout of a mashup to resemble native look-and-feel, would be done using this component.

As shown in Figure 4, the server side middleware can be hosted either on the service delivery platform (SDP) of a Telecom provider or on a third party server. In the first scenario, Telecom features only from a single provider are available. In the second setup, however, multiple operators can make their offerings available. In this case, it is imperative for the Telecom block to provide uniform APIs across different providers for enabling easy portability of mashups.

4 Implementation

In this section, we take three mobile platforms - iPhone, Android, and Nokia S60 - and implement various device and server-side middleware components for each platform. We also describe how these components are utilized to mobile enable two Web-based mashup environments, Eclipse and Lotus Mashups.

4.1 Device Middleware

On all the three platforms we consider in this paper, it is possible to embed a browser engine inside a native application that allows rendering of Web content developed using artifacts like HTML, JavaScript, and CSS. Moreover, this feature enables native access to various browser DOM elements, for example the current Url, title and cookies. We extend this embedded browser model to provision mashup bridges for native device capabilities on each platform. A high-level summary of this effort is given below:

Android: SDK release 2.1 for Android provides a `WebView` class in Java for creating a browser instance inside a native Android application. Further, the

platform offers a generic API 'addJavaScriptInterface()' in the `WebView` class that allows addition of Java objects within an embedded browser instance, and lets them be treated as pure JavaScript entities. In essence, any such object now becomes a 'connection' between the browser context and the native Android platform. In our setting, we use this facility as the basis to expose JavaScript functions for various device features. For example, for Location information, we create a Java object *locObj* containing an instance *lm* of the `LocationManager` class, and add *locObj* in the browser context through 'addJavaScriptInterface()'. Now, any JavaScript function within this browser can use *locObj* as a channel to access *lm* and invoke the underlying *getLocation()* functionality. In case an exception is encountered during invocation on the Android platform, it is propagated via *locObj* to the corresponding JavaScript interface by passing error codes in the return object.

An issue that required resolution pertained to callbacks, since a JavaScript 'connection' object in the browser context lacks the ability to define a callback function from the underlying Java object. To overcome this, we utilize the 'loadUrl()' method in the `WebView` class using which a JavaScript function, qualified as a `Url`, can be called from a Java object. For example, consider the JavaScript interface for Location information where a callback function *locObtained()* is exposed to a developer. Now, to associate *locObtained()* with availability of location information inside the connection object *locObj* (once underlying *lm.getLocation()* returns), we invoke 'loadUrl()' in the WebView instance and pass "javascript:locObtained(<Location>)" as the argument. Here <Location> needs to be marshalled as a string serialized JSON object, since only String objects are allowed to be exchanged through the 'loadUrl()' mechanism.

iPhone: A browser instance can be embedded in an iPhone application (developed using the Objective-C language) via the `UIWebView` class of the `Cocoa Touch API` in the OS version 3.1.2. To capture certain key browser events, a 'delegate' object can be attached to `UIWebView` through the `UIWebViewDelegate` protocol. In particular, one of the methods defined by this protocol - 'shouldStartLoadWithRequest:navigationType:' - gets called every time the browser navigates to a new URL. This method also gets invoked when JavaScript in the page tries to change the page URL by setting the 'location' property of the window object. From the perspective of a mashup bridge, we use this feature every time a browser needs to send request for accessing a native API, by executing the following JavaScript code template: *window.location =*
http://<mashupDomain>?Id=<id>&Name=<deviceFeature>&Values=<values>
Here, `Id` is the request identifier, `Name` is the native feature being called and `Values` contains the parameters needed to invoke the feature. Execution of this code gets first trapped in the above mentioned method of the delegate protocol, where the unique `mashupDomain` qualifier helps to deduce that a native service is being accessed. This method also extracts values of the parameters from the URL, following which the native service is invoked.

Once the native service has been executed, it may need to send a return value back to the browser. The `UIWebView` class has a method called 'stringByEvaluatingJavaScriptFromString()' which takes a piece of JavaScript code, wrapped in a

String, as the input parameter, and executes it in the embedded browser context. Using this method, we send results from a native service execution back to the browser through the code template
stringByEvaluatingJavaScriptFromString:@"callBack ({Id=<id>, Response=<response>})"
Note that this invokes the desired `callBack` function in the mashup. Moreover, the JavaScript object passed as a parameter to this function contains an `Id` property that allows us to identify the request message that originally invoked the native feature, and a `Response` property encapsulating the results of underlying execution.

Nokia S60: Symbian Series 60 5th Edition of Nokia contains a `Browser` class in the `eSWT` package for embedding a browser instance within a J2ME Midlet application. An instance of this class not only provides means to visualize and navigate through HTML documents, but also supports attachment of listeners that receive events corresponding to changes in the Url, title and status text of the rendered content. For accessing native APIs through a mashup, we define a `TitleListener` for the embedded browser that listens for title changes, and make use of the following JavaScript code template:
currentTitle=window.title;
window.title= <mashupDomain>:Name=<deviceFeature>&Values=<values>;
window.title=currentTitle;

Similar to the iPhone model, execution of this code first gets trapped inside the provided `TitleListener`, where it is interpreted as a request for executing a device feature with the passed values of associated parameters. For sending results from a native feature to the invoking browser instance, we utilize the *setUrl()* routine of the `Browser` class. For example, to invoke the callback function *locObtained()* attached to the *getLocation()* feature, we execute the following code in J2ME:
browser.setUrl("javascript:locObtained(<Location>)"). Again, <Location> here corresponds to a String serialized JSON object of the location information.

Following this realization of mashup bridges, we took three device features - Location, Contacts and Camera, and exposed them using JavaScript interfaces within an embedded browser context on each platform. We next proceeded to provide another important component of our device middleware - a defragmentation layer that absorbs differences in syntax and semantics of these JavaScript interfaces across different platforms. Towards this, we build upon our previous work [1], that handles heterogeneity of mobile features using a three-phased process - 1) semantic phase, where we fix the structure of the interface, in terms of the method name, associated parameters (including their name, ordering and dimensions), as well as the return value, 2) syntactic phase, in which we remove differences in data structures of various objects, and 3) binding phase, which contains implementation of the common interface on top of the original platform offering, and also provides mechanisms to fine-tune an interface using platform specific attributes and properties. Due to lack of space, we omit further details here, and direct the interested reader to [1] for more information.

Finally, we designed an access control layer that helps a user configure policies pertaining to access of various device and Telecom features. In essence, these

policies are defined around three basic tenets or criteria - domain (determined by Url) of the mashup in consideration, context of the user (determined through a combination of user's current location and the current time), and frequency of access (for example, how often can a feature be accessed). Whenever a feature is invoked, depending upon the configuration of policies in this layer, one of the following four options is exercised - (i) allowing a feature to be invoked (ii) denying the invocation, (iii) partial invocation, for instance, providing abstracted location information instead of available precise latitude and longitude coordinates, (iv) prompting the user for further clarification, before deciding upon the next action - allowing, denying or partial invocation. Note that the access control layer applies to both data features, such as Location and Calendar information, and functionality, such as SMS and Call Control.

4.2 Server Middleware

For Telecom support, we take two real-life Telecom products - IBM Telecom Web Services Server[3] (TWSS) and IBM WebSphere Presence Server (WPS)[4], and build mashable interfaces in both Java and JavaScript format for SMS, Location, Presence and Third-party Call Control (3PCC). TWSS is an offering from IBM that enables Telecom operators to provide developers with controlled, reliable access to network capabilities such as Location, SMS and Call Control through standards-based Parlay-X Web Services. On the other hand, WPS is a substrate that collects, manages, and distributes real-time presence information to applications and users via the SIP protocol. While SMS, Location and 3PCC are designed on top of TWSS, Presence functionality is developed on SIP interfaces exposed by WPS. In addition to TWSS and WPS, we take a Telecom network simulator - OAS[5](Open API Solutions) version 2 - that exposes various simulated network services using the CORBA protocol, and enable mashable interfaces for the same.

Java interfaces for Parlay-X based services were implemented by first generating Java clients from the given WSDL descriptions, and wrapping those inside our interfaces. For Presence service based on SIP, we used the JAIN[6] SIP standard and rendered the interfaces in Java for publish, subscribe, etc. Services in OAS simulator were offered by creating Java stubs that talk to the simulator through the CORBA protocol. As far as JavaScript interfaces are concerned, those for Parlay-X based functionality were created by defining XML fragments containing the desired SOAP headers, sending these as Ajax requests to the server from within JavaScript, and parsing the returned XML fragments containing SOAP responses. However, for SIP based Presence service, this procedure does not work since SIP messages are exchanged over TCP/UDP. A JavaScript interface in this case was created by first implementing a servlet that talks to the Presence Server using SIP

[3] http://www-306.ibm.com/software/pervasive/serviceserver/
[4] www.ibm.com/software/pervasive/presenceserver/
[5] www.openapisolutions.com
[6] https://jain-sip.dev.java.net/

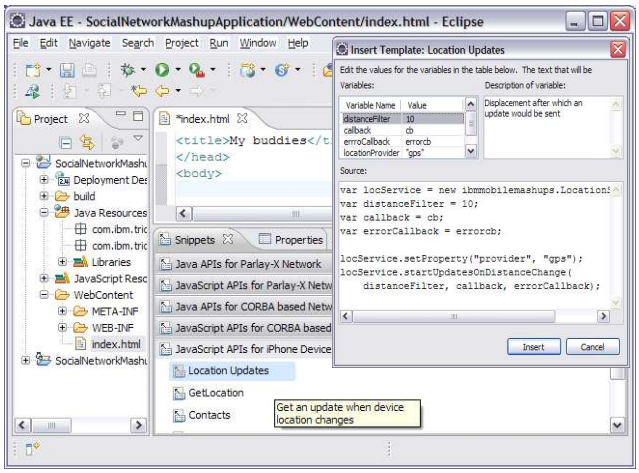


Fig. 5. Eclipse Plug-in to Integrate Device and Telecom Features

over UDP messages. The Presence JavaScript interface interacts with this servlet to fetch presence-related information. Interfaces for CORBA based OAS services were similarly developed using the servlet model.

As far as device block is concerned, our device adapter component is implemented as a servlet that first detects which platform the mashup is running on, with the help of 'user-agent' field of the invoking browser. Based on this knowledge, the appropriate *'.js'* files containing code for invoking device features are included in the mashup. We also developed three set of *'.css'* files, one each for Android, iPhone and Nokia S60, that tailor the mashup for the corresponding platform with respect to font style, font size and other UI attributes. Moreover, facility was also added to selectively disable device features in a mashup where they are not available - an instance in case is the SMS service, wherein the iPhone provides no way to programmatically send an SMS message but both the Android and the S60 platforms do so. Finally, we used dojo framework[7] to create several rich cross-platform UI widgets that can be used in multiple mashups. The other component of this block, i.e. device listener is also implemented as a servlet where information from the device can be submitted using the servlet *Post* method. Each device data is currently stored by the listener using a 3-tuple *<mobile phone#, device data, time-stamp>*. For this data, the server side middleware can also configure access policies related to its usage in a mashup.

4.3 Integration with Mashup Environments

In this subsection, we take two environments for creating mashups, and give an overview of how our middleware components were integrated with these environments to make them suitable for a three-dimensional mobile setting.

[7] www.dojotoolkit.org

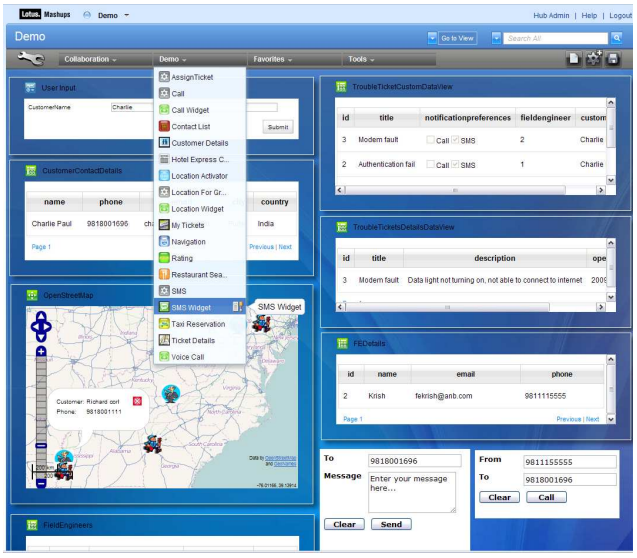


Fig. 6. Integration with IBM Lotus Mashups

Eclipse Platform: Eclipse[8] is a popular open source meta application framework for building, deploying and managing software across the life-cycle. On eclipse, we extend the notion of a *Dynamic Web Project* to define a "Mobile Mashup Project" that pre-packages various servlets and javascript libraries for accessing Telecom and device features. For building a mashup, a developer first creates an instance of the Mobile Mashup Project, and extends it by adding her own servlets, JSPs, HTML files, PHP scripts, JavaScript libraries, etc. This project is finally packaged as a *'war'* file, hosted on a Web server (attached to Eclipse or stand-alone) and accessed from our enhanced browser on the mobile platforms.

To ease a developer's task further, we designed an Eclipse 'plug-in' to make device and network interfaces available in various editors. The plug-in, as shown in Figure 5, is mainly composed of a Function Drawer and a Configuration View, and is realized by extending the *Snippet Contributor* plug-in of Eclipse. As seen in the figure, each network and device feature inside the Function Drawer is first organized as an 'item'. An item can be added to the mashup code by dragging it from the drawer and dropping on the development editor. During this process, the developer is provided with a configuration dialog box that lets her fill values for associated parameters and properties. Subsequently, an appropriate code fragment is generated for the corresponding item and inserted in the editor.

IBM Lotus Mashups: Lotus Mashups [11] is an integrated suite from IBM for mashing widgets and data feeds from multiple sources in a single graphical layout. At the core of this suite is the concept of 'iWidgets', that delineates a standard way to wrap Web content and allow that content to participate

[8] www.eclipse.org

in a mashup by providing the necessary user interface and supporting wiring capabilities for interaction with other widgets. We use the iWidget notion to integrate our middleware components within the mashup environment. In essence, we package our Javascript files, jars, servlets, etc. corresponding to a device or Telecom feature as an iWidget and add it to the Mashup Catalog. Once widgets for various features were added to the catalog, the tool provided automatic support for storing, sharing, discovering, and reusing these widgets.

New mashup applications can be created by dragging our widgets from the catalog to a mashup page and wiring them with other widgets to enable proper communication and coordination. For example, Figure 6 shows an SMS widget that displays a UI for entering phone number and text message as inputs. On the device front, *Mashup Enabler*, a component of Lotus Mashups, is loaded with the mashup application and provides the runtime environment for a mashup to execute within our embedded browser context. Going forward, we plan to bundle this enabler as a part of the device middleware itself - this would enhance performance during mashup loading and execution.

5 Evaluation

In this section, we evaluate our mobile mashup framework along two different perspectives. At first, we discuss the ease of mashup development with respect to various software engineering principles. Thereafter, we present performance analysis of our framework.

5.1 Mashup Development

Figure 7 presents the snapshot of a social networking mashup on Android, iPhone and Nokia S60 platforms, created using our mobile mashup framework. As shown, the mashup brings together various offerings from the device, Telecom network and the Web. Summarized below are features used from each dimension -

Device - Camera to take pictures, Contacts to obtain a user's friends list, Location Updates to get GPS location periodically.

Telecom - Call and SMS to communicate with friends, Presence Services to know Telecom presence.

Web offerings - Twitter to tweet as well as to fetch latest tweets of friends, Facebook for photo and profiles, Google Maps to render nearby friends on a map.

Note that our objective is not to emphasize a particular mashup, rather to highlight the ease with which such applications can be developed and executed on different devices. Let us look at the code fragment that enables picking of location information from the device, and sending it to the mashup server:

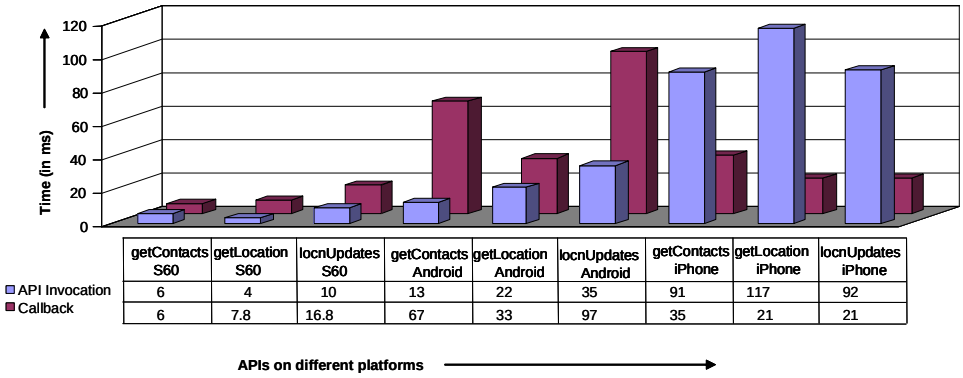
```
function  getLocationUpdates(){
      var locService = new ibmmobilemashups.LocationService();
      locService.startUpdatesOnTimeChange(timeFilter, callback, errorCallback,
                                    options);
}
```

(a) (b) (c)

Fig. 7. Social Networking Mashup on (a) Android (b) iPhone (c) Nokia S60

```
function locationUpdated (location){
      var latitude = location.latitude;
      var longitude = location.longitude;
      ...
      //Invoke servlet to update location on server
      ...
}
```

The first function initializes the service for accessing location information, and then calls the `startUpdatesOnTimeChange()` function to get periodic location updates. Every time an update is obtained, a common callback method `locationUpdated()` is invoked that parses the `location` object and sends updated information to the server through servlet calls. Using this code as an example, we argue the following benefits of our framework from a software engineering aspect.
- Portability: Once a common interface has been realized that hides semantic and syntactic heterogeneities of device features, the task of a developer in terms of porting a mashup across the platforms is significantly reduced. For instance, for location updates, the developer does not need to worry about "provider" (GPS or network) in the case of Android, "timeout" and "maxage" parameters on Nokia S60, etc. Moreover, any platform specific attributes can be provided in a consistent manner using the `options` parameter, as shown in the code above.
- Coding Complexity: Our approach hides the complexity of using platform specific provisions from the developers. For example, using our mashup bridge, a developer is not required to deal with the `Intent` and `IntentReceiver` classes



	getContacts S60	getLocation S60	locnUpdates S60	getContacts Android	getLocation Android	locnUpdates Android	getContacts iPhone	getLocation iPhone	locnUpdates iPhone
API Invocation	6	4	10	13	22	35	91	117	92
Callback	6	7.8	16.8	67	33	97	35	21	21

Fig. 8. Performance of Device Middleware

for handling callbacks on Android. Similarly, details of Telecom protocols across different standards, such as Parlay-X and CORBA, are shielded from the developer. As a result, the code becomes easier to develop and debug.

- Maintenance: As mobile platforms evolve, new version of a platform may have different APIs, as compared to the previous versions. For example, moving from release 1.5 to 2.1, Android platform changed the APIs for accessing contact information. In such scenarios, the differences can be absorbed inside the mashup framework for newer version of the platform, thereby requiring no changes in the mashup application. This makes a mashup easier to maintain 'as platforms evolve'.

5.2 Performance

The core of our device middleware is the bi-directional communication between the mobile browser and the native platform. Figure 8 shows the performance of this communication, wherein for each platform, we took three device features and measured the time with respect to - 1) invoking a native device feature from JavaScript code, and 2) callback from native code to JavaScript. Each number reported is an average of ten execution traces. As the figure shows, the overhead of device middleware is very small, and indicates a fast transfer between the embedded browser context and the corresponding native feature. Across the platforms as well as across the APIs on a particular platform, there are variations due to differences in JavaScript processing engines, event passing and handling mechanisms in the embedded browser, de-fragmentation logic, etc. From a broad perspective, considering the time a typical application spends in UI interaction, business logic, etc., we conclude that the cost of using the device middleware is negligible as compared to the total mashup runtime.

On the server side, we did a preliminary performance evaluation of various components - device adapter, device listener, as well as Telecom block, with a few sample mashups that we created using our framework. While portability across devices was easy to achieve, the overhead of these components too was found to

be very small - of the order of a few milliseconds. As the next step, we plan to extend the server side analysis by evaluating against a large number of mashups.

6 Discussion and Related Work

There are several professional tools in the industry that facilitate the creation of Web based mashups. Examples are Yahoo Pipes [19], AquaLogic [2], and IBM Lotus Mashups [11]. Most of these tools provide a browser-based assembly portal where developers can choose services made available by different providers, and integrate them in a mashup - all driven by a simple visual user interface. Academic research on mashup tools has also been undertaken. For instance, [10] presents an environment for developing spreadsheet-based Web mashups. As far as our work is concerned, our middleware components for device and Telecom features can be easily integrated with existing mashup technologies to enable support for mobile mashups, as we demonstrated earlier in Section 4.3. [4] presents a mobile mashup platform that integrates data from Web-based services on the server side and utilizes users' context information from sensors to adapt the mashup at the client side. However, this framework is far from a comprehensive approach required for incorporating the three-dimensional mobile setting that we outlined in this paper. Also, it currently runs on certain Nokia devices only.

Device browser fragmentation is a major challenge for mobile mashups. Although our target platforms - iPhone, Android, and S60 - all make use of the WebKit browser engine [18], each platform has adopted a different WebKit version, and has replaced or extended various browser subsystems. There are situations where certain user interface elements in the browser do not provide the same user experience on all platforms. For instance, consider the scrolling features of the browser. A web page might have more than one element in the current view that requires scrolling, such as an embedded 'iframe'. The touch screen interface on the Nokia S60 devices allows the user to focus on individual elements on the page via the stylus and scroll inside those elements. The iPhone, on the other hand, does not provide the ability to focus on individual elements and can scroll only the outer most browser frame via the flick gesture. In such cases, the developer either needs to use different UI elements on different platforms, or build a set that works well on all platforms. Going forward, our device adapter component can be enhanced to help a developer tackle this problem more efficiently.

Most smart-phone platforms today provide native APIs for several services, such as GPS location, address book and contacts, calendar, accelerometer, camera, and video. Despite the general need for accessing similar services during cross-device application development, there is a considerable level of API fragmentation. In fact, we find that APIs for common services in J2ME-based platforms have also become fragmented on platforms that support new hybrid Java runtimes, such as Android. Fragmentation is further exacerbated on non-Java based platforms like iPhone, which uses the Objective-C language and the Cocoa user interface library. Various standardization efforts, such as OMTP Bondi [13], attempt to overcome this fragmentation. In [1], we presented a three-tiered model for absorbing

heterogeneity in syntax, semantics and implementation of interfaces corresponding to device features across multiple platforms. Similarly, PhoneGap [15] is an effort towards enabling uniform JavaScript access to native APIs. From the perspective of our framework, we could use either of these approaches as building blocks in our device middleware stack.

Session Initiation Protocol (SIP) is a standard being widely adopted by Telecom operators to expose their core functionalities - voice service, SMS service, Call Control, etc. - using SIP. JSR-289 [9] has been proposed by Sun and Ericsson to enhance existing SIPServlet specification and support development of composed applications involving both HTTP and SIP servlets. Web21C [17] from British Telecom is a Web 2.0 based service aggregation environment that allows developers to integrate core Telecom functionality with other Web services into a single application. On the other hand, [3] gives a broad overview of existing approaches for enabling a unified Telecom and Web services composition. However, both fall short in describing a generic model for supporting mashable Telecom interfaces. In [12], we introduced SewNet which provides an abstraction model for encapsulating invocation, coordination and enrichment of Telecom functionalities. In this paper, we have extended the earlier work and presented middleware components for enabling cross-operator Telecom network features that can be utilized during creation of mobile mashups.

One of the major challenges in the area of mobile applications is the huge privacy and security implication around sensitive user information like location, contacts and calendar entries [6]. PeopleFinder [16] is location sharing application that gives users flexibility to create rules with varying complexity for configuring privacy settings for sharing their location. In this paper, we have created a similar policy framework, but differ on two counts. Firstly, we move beyond location and cover other sensitive information as well. Secondly, we apply the policy framework to a generic mobile mashup setting, and not to a specific application. The World Wide Web Consortium's Platform for Privacy Preferences (P3P) specification [5] provides a standard way for Web sites to convey their privacy policies in a computer-readable format. Extending the security block in our framework to support such policy languages will be an important step going forward.

7 Conclusion and Future Work

Most modern smart phones today have browsers that are HTML and JavaScript standards compliant, and provide a rich, powerful Web browsing experience to mobile users. Evolution of mobile browsers, in turn, is driving the adoption of mashups that are accessed through the mobile device. In this paper, we proposed a framework for creating next generation mobile mashups that amalgamate data and offerings from three dimensions: device features, Telecom network, and Web accessible services. Towards this, we established middleware components, both on the server side as well as the device side, to provide support for mashing device and Telecom interfaces. Our framework allows portability across different device

platforms and different Telecom operator networks. We demonstrated the utility of our framework using three popular platforms - Nokia S60, Android and iPhone.

In the future, we would like to extend our framework to cover more platforms, device features and Telecom services, as well as enhance the existing security and privacy considerations. Moreover, we wish to integrate our middleware framework with several mashup and mobile development environments. This would help us gain valuable feedback with respect to further refinements and extensions from the developer community involved in creation of innovative mobile applications.

References

1. V. Agarwal, S. Goyal, S. Mittal, and S. Mukherjea. MobiVine: A Framework to Handle Fragmentation of Platform Interfaces for Mobile Applications. In *Proceedings of 10th International Middleware Conference*, Illinois, USA, November 2009.
2. BEA AquaLogic Family of Tools. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/aqualogic/.
3. G. Bond, E. Cheung, I. Fikouras, and R. Levenshteyn. Unified Telecom and Web Services Composition: Problem Definition and Future Directions. In *Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*, Georgia, 2009.
4. A. Brodt and D. Nicklas. The TELAR Mobile Mashup Platform for Nokia Internet Tablets. In *Proceedings of 11th International Conference on Extending Database Technology (EDBT)*, Nantes, France, March 2008.
5. L. F. Cranor. P3P: Making Privacy Policies More Useful. *IEEE Security and Privacy*, 1:50–55, 2003.
6. M. Hypponen. Malware Goes Mobile. *Scientific American*, November 2006.
7. IP Multimedia Subsystem (IMS) Architecture. http://www.dataconnection.com/sbc/imsarch.htm.
8. J. Rosenberg, H. Schulzrinne et al. SIP: Session Initiation Protocol. http://www.rfc-editor.org/rfc/rfc3261.txt, 2002.
9. JSR 289. http://jcp.org/en/jsr/detail?id=289.
10. W. Kongdenfha, B. Benatallah, J. Vayssiere, R. Saint-Paul, and F. Casati. Rapid Development of Spreadsheet-based Web Mashups. In *Proceedings of 18th International World Wide Conference (WWW)*, Madrid, Spain, April 2009.
11. Lotus Mashups. http://www-01.ibm.com/software/lotus/products/mashups/.
12. S. Mittal, D. Chakraborty, S. Goyal, and S. Mukherjea. SewNet - A Framework for Creating Services Utilizing Telecom Functionality. In *Proceedings of 17th International World Wide Conference*, Beijing, China, April 2008.
13. OMTP Bondi. http://bondi.omtp.org/.
14. Open Service Access (OSA); Parlay-X Web Services; Part 1: Common. 3GPP TS 29.199-01.
15. PhoneGap. http://phonegap.com/.
16. N. Sadeh, J. Hong, L. Cranor, I. Fette, P. Kelley, M. Prabaker, and J. Rao. Understanding and Capturing Peoples Privacy Policies in a Mobile Social Networking Application. *Journal of Personal and Ubiquitous Computing*, 13(6), August 2009.
17. Web 21C SDK. http://web21c.bt.com/.
18. The WebKit Open Source Project. http://webkit.org/.
19. Yahoo Pipes. http://pipes.yahoo.com/pipes/.