Towards Enabling Next Generation Mobile Mashups

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

[1] IBM Research - India, New Delhi
{avikas,gsunil,sumittal,smukherj}@in.ibm.com
[2] IBM T.J. Watson Research Center, NY, USA
{jponzo,fenils}@us.ibm.com

Abstract. Evolution of Web browser functionality on mobile devices is the driving force for 'mobile mashups', where 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 the 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 various techniques exist for creating desktop-based 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 describe middleware support 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 and Telecom protocols. Based on our approach, we provide an implementation of mashup framework on three popular mobile platforms - iPhone, Android and Nokia S60, and discuss it's utility.

1 Introduction

'Mashups' are applications created by integrating offerings from multiple Web sources, and have become very popular in recent years. Further, adoption of *mobile mashups* is being 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 the mobile users. With rapid enhancements in processing 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.

⋆ This author is currently working at Google Inc., USA.

P. Sénac, M. Ott, and A. Seneviratne (Eds.): MobiQuitous 2010, LNICST 73, pp. 13–25, 2012.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2012

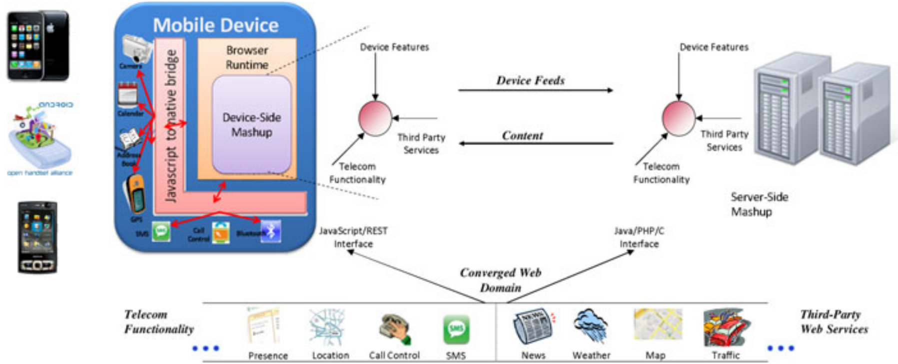


Fig. 1. Overview of a Mobile Mashup

To help developers create rich mobile applications, popular platform vendors like Nokia, Blackberry and Android expose interfaces for a variety of features available on the mobile handset, for both data (user's contacts, calendar, geographic location, etc.) and functionality (making calls, sending SMSes, using the camera, etc.) [2]. On similar lines, there is willingness by Telecom operators to move from a *walled-garden* model to an *open-garden* model [12], whereby they expose core functionalities of Location, Presence, Call Control, etc. as services to developers. From the perspective of mobile mashups, it is desirable that such Device and Telecom features are utilized to enhance the entire mashup experience of a mobile user. For example, location and presence information available on mobile phones/Telecom can be used to design interesting mashups related to directory services, workforce management, social networking, etc. Similarly, camera on the phone can help enrich a retail mashup with the facility to retrieve product information based on scanning of bar-codes.

Various tools and technologies exist [4,11,18] that help developers create mashup applications for the desktop. However, these fall short in the context of a three-dimensional setting present in the mobile domain - native Device features, Telecom services and Web-based offerings. In this paper, we start with our view of a next generation mobile mashup consisting of device side and server side mashup components, as depicted in Figure 1. The server side mashup 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 environment 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.

We argue that in order to enable the above mashup scenario, the following novel characteristics need to be incorporated -

(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).

(2) **Device Integration**. On most platforms, APIs for device features are available in native language (J2ME, C++, Objective C, etc.), while for mashup development the interfaces are required along 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 three distinct attributes -
(i) support for a *bidirectional communication* 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 (return values, exceptions) available as a result of native invocation can be marshaled back to the mashup through the browser.
(ii) provisioning a *channel for callbacks*, in case result of an API invocation is not immediately available or possible - for example, periodic proximity alerts. In essence, events generated in the native context on a mobile platform should be channeled through appropriate signaling and notification to the invoking browser context of a mashup.
(iii) handling immense *fragmentation* in syntax and semantics of various device features across different platforms - starting from diversity in the 'name' of the interface, to 'name', 'data type' and 'ordering' of attached parameters, to the differences in the set of 'exceptions' thrown by each platform.

(3) **Telecom Support**. A number of legacy protocols exist in the Telecom domain, while new standards continue to be drafted and absorbed by different operators. Therefore, to reduce burden on a mashup developer, we need an *abstraction* layer that lets various Telecom features be invoked without requiring the developer to know the underlying protocol specifics. Moreover, for a given feature, this layer should enable a *seamless switch* among different protocols; especially required in scenarios where the Telecom networks are gradually evolving to move from legacy interfaces to standards like Parlay-X [14] and SIP [8].

2 Mobile Mashup Framework

In this section, we describe a framework that enables realization of the mobile mashup view outlined above. Our framework consists of middleware components, both on the device side, as well as the server side. We describe these components next, while characterizing the role played by each component in the mobile mashup setting.

[1] in this case, the device

2.1 Mobile Device Middleware

As shown in Figure 2, the device side middleware runs on top of existing platform middleware, 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 to execute 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 discussed in the previous section, this bridge provides a bi-directional communication capability between the browser context and the native APIs (allowing the browser to invoke native APIs, pass inputs and receive outputs), and also supports passing of events from native APIs to the browser through JavaScript callbacks. The bridge can be realized in multiple ways, - 1) through *modification of the source code* of an existing browser to allow access to native capabilities via JavaScript, 2) through *creation of a plugin* for the browser that adds device features without modifying the browser source code, and 3) by *embedding a browser runtime* (where it is available as a 'class' in the native programming language) within a native mobile application, and enabling access to native features for mashups rendered through the embedded runtime. Depending on the specifics of a given platform, one or more of these techniques could be used.

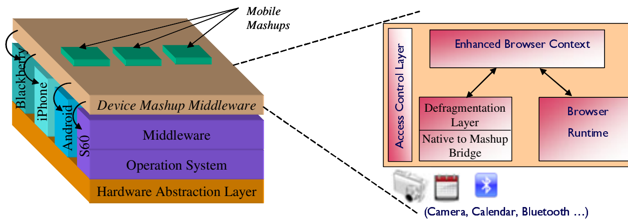


Fig. 2. Middleware Support on Device

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 interfaces across various platforms. Our generic model for common JavaScript interfaces is -

<interface name> (<arguments 1 ... n>, <callback>, <errorCallback>, <options>)

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)

Here, 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 [2], this parameter is optional and can be 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. Normally, `null` value should be used for this parameter, in which case default values for various attributes would be used on the corresponding platform.

Accessing a device or Telecom feature might have monetary cost associated to it, such as sending an SMS, making a Call, etc. or mashup 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 based on user defined policies. 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 is allowed. 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. A user should be able to configure various policies when the mashup is first accessed and refine the same over time.

2.2 Server Side Middleware

As shown in Figure 3, the server side middleware consists of two blocks - 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 is the **Protocol Binding** component that connects to various Telecom services using the underlying network protocols, such as SIP, Parlay-X and CORBA. Through these binding components, the framework removes the burden - from mashup developer - of knowing Telecom specifics, for instance 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

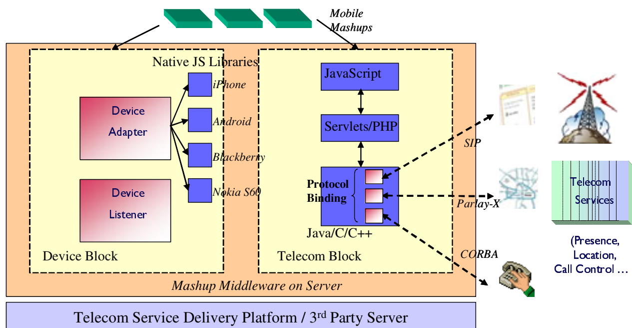


Fig. 3. Middleware Support on Server

languages - Java, C, C++, etc. While these interfaces can be directly used in a server side mashup, REST based interfaces are also exposed so that Telecom services can be invoked from client side mashups using JavaScript.

Binding stubs also enable seamless switching between different protocols. Consider a scenario where network location information can be fetched 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 returns the location information encapsulated again under SOAP. In our framework, these disparate steps are wrapped under generic interfaces and corresponding stubs for various Telecom protocols are provided. Properties and attributes required for each protocol, 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 data feeds 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 for server side content generation. 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 is 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, is done using this component.

As shown in Figure 3, the server side middleware can be hosted either on the service delivery platform (SDP) of a Telecom operator or on a third party server. In the first scenario, Telecom features only from a single operator are available. In the second setup, however, multiple operators can make their offerings available, which makes it imperative for the Telecom block to provide uniform APIs across different providers for enabling easy portability of mashups.

3 Implementation

In this section, we describe an implementation of our mobile mashup framework consisting of device-side and server-side middleware components.

3.1 Device Middleware

On all the three platforms we consider in this paper - Android, iPhone, and Nokia S60 - it is possible to embed a browser engine inside a native application that allows rendering of Web content developed using artifacts like HTML and JavaScript. We extend this embedded browser model to develop mashup bridges for native device capabilities on each platform.

The **Android** platform provides a `WebView` class in Java for creating a browser instance inside a native Android application. Further, this class provides a generic API 'addJavaScriptInterface()' that allows addition of Java objects within a `WebView` 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; we use this facility as the basis to expose JavaScript functions for various device features. An issue that required resolution pertained to callbacks, since a JavaScript 'connection' object in the browser context lacks the ability to provide a callback function for 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 invoked from a Java object. Also, since only string arguments can be passed to the called JavaScript function using this technique, all parameters are marshaled as string serialized JSON objects.

The **iPhone** and **Nokia S60** platforms provide mechanism for attaching listeners to the embedded browser for receiving events corresponding to changes in *URL*, *title* and *status text* of the browser. We use this mechanism for accessing native APIs through JavaScript on these platforms. In essence, we first define listeners that listen for changes in URL, title or status text of the embedded browser. Specifically, on iPhone we attached listener for *URL* change event, whereas on Nokia S60 we intercepted the *title* change event. Now, whenever we need to call a native API from JavaScript we change the URL/title to *<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 is trapped by above listeners, where the unique `mashupDomain` qualifier helps to deduce that a native service is being accessed. The listener extracts the values of the parameter and calls the specified native API. Moreover, these platforms provide a mechanism similar to that on Android for calling a JavaScript function from the native code - iPhone provides a method called *stringByEvaluatingJavaScriptFromString()* in the `UIWebView` class, whereas S60 provides a method called *setUrl()* in the `Browser` class. We use these methods to send the response back to JavaScript from the called native API.

Using the mashup bridges, we exposed various device features such as Location, Contacts, Camera, etc. through JavaScript interfaces within an embedded browser context on each platform. We then provided a de-fragmentation layer that absorbs differences in syntax and semantics of these interfaces across different platforms. Towards this, we build upon our previous work [2], and handle 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 [2] for more information.

Finally, we designed an access control layer [1] that allows a user to configure access policies for various device and Telecom features. Currently, the implementation is available for Android platform, and allows policies defined around three basic tenets - 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). In essence, whenever a feature is invoked in a mashup, depending upon the policies configured for that mashup, the invocation is either allowed or denied. We are working towards making this access control layer available on other platforms as well.

3.2 Server Middleware

For Telecom support, we first defined common APIs for various services in Java and then created protocol bindings for different Telecom protocols. In particular, we took two real-life Telecom products - IBM Telecom Web Services Server (TWSS)[2] and IBM WebSphere Presence Server (WPS)[3], and built stubs for calling SMS, Location, Presence and Third-Party Call Control (3PCC) services. TWSS allows access to network services through standards-based Parlay-X Web Services, whereas WPS provides real-time presence information via the SIP protocol. Apart from this, we also created stubs for various network services exposed in CORBA by a Telecom network simulator - Open API Solutions (OAS)[4] version 2. As discussed earlier, these Protocol Binding stubs allow easy switching of Telecom protocols without requiring any changes in a mashup application.

For mashable client side APIs, we created JavaScript interfaces for Parlay-X based functionality by defining XML fragments containing the desired SOAP headers - sending the XML as AJAX requests to the server, and parsing the returned XML fragments that contained 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 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.

For the device block implementation, we develop three JavaScript files containing code for invoking features of each device and three CSS files to tailor the mashup UI for the corresponding platform, one each for Android, iPhone and Nokia S60. The Device Adapter component is implemented as a servlet that detects the device platform using 'user-agent' field in the request header and adds appropriate JavaScript and CSS files. An option is also added to selectively disable certain device features that are not available - an instance in case is the SMS service which is not exposed in iPhone. The Device Listener component was also implemented as a servlet where information from the device can be submitted using the servlet *Post*

[2] `http://www-306.ibm.com/software/pervasive/serviceserver/`
[3] `www.ibm.com/software/pervasive/presenceserver/`
[4] `www.openapisolutions.com`

method. The device data is stored by the listener using a 3-tuple <*mobile phone#, device data, time-stamp*> and made available to various mashup applications.

3.3 Integration with Mashup Environments

We took two environments for creating mashup applications, Eclipse[5] - a popular open source meta application framework, and Lotus Mashups [11] - an integrated suite from IBM for mashing widgets and data feeds from multiple sources, and enhanced them to enable mashing of device and Telecom features in addition to web offerings. For Eclipse, these APIs were integrated via the 'Snippet Contributor Plugin' that provides drag-n-drop of APIs in the web editor. In Lotus Mashups, on the other hand, the APIs were made available in the required 'iWidget' format. We do not provide integration details here due to lack of space, and direct the interested readers to [3] for more information.

4 Discussion

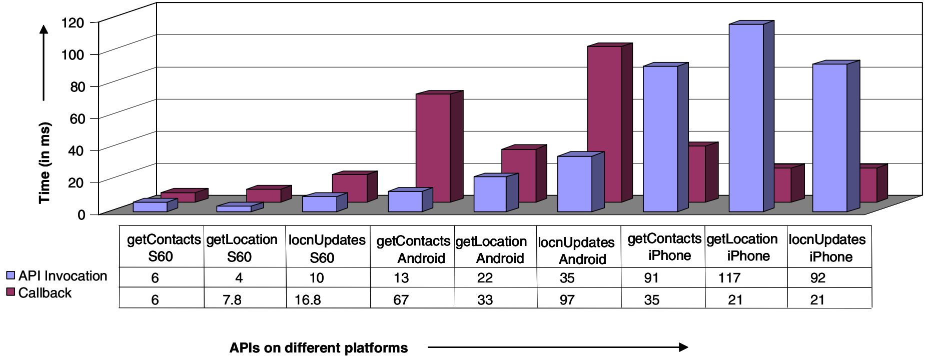
We have developed several mashups that use Device and Telecom features enabled by our framework; Figure 4 presents snapshots of one such social networking mashup. As shown, the mashup brings together various offerings from the device (*Camera* to take pictures, *Contact List* to obtain a user's friends list, *Location Updates* to get GPS location periodically), Telecom network (*Call* and *SMS* to communicate with friends, *Presence Services* to get their presence data), and the Web (*Twitter* to publish and fetch tweets of friends, *Facebook* for photo and profiles, *Google Maps* for visual display of friends on a map).

Note that a single application was developed, which looks and behaves exactly the same across Android, iPhone and Nokia S60 platforms. The common interfaces provided by the framework hide semantic and syntactic heterogeneities of device features, thereby allowing a single code base and ease of programming. Moreover, when a new version of a platform introduces changes in device APIs, it is absorbed in our framework, alleviating the need of application maintenance 'as platforms evolve'. For example, APIs for accessing contact information were changed in Android platform moving from release 1.5 to 2.1 - these changes were accounted for in the mashup bridge implementation of the corresponding Android releases. Similar arguments apply to usage of various Telecom features.

The core of our device middleware is the bi-directional communication between the mobile browser and the native platform. Figure 5 shows the overhead associated with this communication for each platform. Here, we took three device features and measured the time for - 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. There are variations,

[5] `www.eclipse.org`

Fig. 4. Social Networking Mashup on (a) Android (b) iPhone (c) Nokia S60



	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. 5. Performance of Device Middleware

however, across the platforms and across the APIs on a given platform, due to differences in JavaScript processing engines, event passing and handling mechanisms in the embedded browser, de-fragmentation logic, etc.

On the server side, we did a similar performance evaluation of various components - device adapter, device listener, and Telecom block. The overhead of these components too was found to be small - of the order of a few milliseconds. From a broad perspective, considering the time a typical application spends in UI interaction, business logic, etc., we conclude that the cost of using our middleware components is negligible as compared to the total mashup runtime.

5 Related Work

There are several professional tools in the industry that facilitate the creation of Web based mashups. Examples are Yahoo Pipes [18], and IBM Lotus Mashups [11]. Academic research on mashup tools has also been undertaken. For instance, [10] presents an environment for developing spreadsheet-based Web mashups. None of these works, however, establish components for integrating device and Telecom features in a mobile mashup. [6] 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 outline in this paper. Also, it currently runs on certain Nokia devices only.

Most smart-phone platforms today provide native APIs for several services, such as GPS location, address book, calendar and camera. We find that APIs for common services in even J2ME-based platforms have 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 [2], 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, [5] 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; but did not deal with 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 [7]. PeopleFinder [16] is a location sharing application that gives users flexibility to create rules with varying complexity for configuring privacy settings around 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 policies to a generic mobile mashup setting, and not to a specific application.

6 Conclusion and Future Work

Evolution of mobile browsers 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 described middleware components, both on the server side as well as the device side, to provide support for mashing device and Telecom features. Our framework allows portability across different device platforms and different Telecom protocols. 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 framework with several mashup and mobile development environments. Finally, we intend to conduct user studies that will help us gain valuable feedback from the developer community with respect to further refinements and extensions.

References

1. Adappa, S., Agarwal, V., Goyal, S., Kumaraguru, P., Mittal, S.: User Controllable Security and Privacy for Mobile Mashups. Technical Report RI10011, IBM Research (October 2010)
2. Agarwal, V., Goyal, S., Mittal, S., Mukherjea, S.: MobiVine: A Framework to Handle Fragmentation of Platform Interfaces for Mobile Applications. In: Proceedings of 10th International Middleware Conference, Illinois, USA (November 2009)
3. Agarwal, V., Goyal, S., Mittal, S., Mukherjea, S., Ponzo, J., Shah, F.: A Middleware Framework for Mashing Device and Telecom Features with the Web. Technical Report RI10009, IBM Research (July 2010)
4. BEA AquaLogic Family of Tools, `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/aqualogic/`
5. Bond, G., Cheung, E., Fikouras, I., Levenshteyn, R.: 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)
6. Brodt, A., Nicklas, D.: The TELAR Mobile Mashup Platform for Nokia Internet Tablets. In: Proceedings of 11th International Conference on Extending Database Technology (EDBT), Nantes, France (March 2008)
7. Hypponen, M.: Malware Goes Mobile. Scientific American (November 2006)
8. Rosenberg, J., Schulzrinne, H., et al.: SIP: Session Initiation Protocol (2002), `http://www.rfc-editor.org/rfc/rfc3261.txt`
9. JSR 289, `http://jcp.org/en/jsr/detail?id=289`
10. Kongdenfha, W., Benatallah, B., Vayssiere, J., Saint-Paul, R., Casati, F.: Rapid Development of Spreadsheet-based Web Mashups. In: Proceedings of 18th International World Wide Conference (WWW), Madrid, Spain (April 2009)
11. Mashups, L.: `http://www-01.ibm.com/software/lotus/products/mashups/`

12. Mittal, S., Chakraborty, D., Goyal, S., Mukherjea, S.: 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. Sadeh, N., Hong, J., Cranor, L., Fette, I., Kelley, P., Prabaker, M., Rao, J.: Understanding and Capturing People's 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. Yahoo Pipes, `http://pipes.yahoo.com/pipes/`