EURASIP Journal on
Wireless Communications and Networking
a SpringerOpen Journal

RESEARCH Open Access

Mashing up the Internet of Things: a framework for smart environments

Edgardo Avilés-López* and J Antonio García-Macías

Abstract

Along with the advent of the Web 2.0 came a rich ecosystem of application services allowing developers to use the functionality provided by Web applications into their own customized solutions. This, together with the current developments on the Internet of Things are laying the foundations of new IP-based smart environments in which applications and services are combined to support users in ways not possible before. Recently, most of the research has focused on improving the networking capabilities of the Internet of Things infrastructure and in enabling the access to the following generation of services. However, there are two more issues that need to be attended. First, how data and functionality provided by services on these smart environments would be modeled in order to facilitate abstraction and composition, and second, how users are intended to interact with the environments in order to make applications support their particular needs. In this article, we present a framework and an user-interaction model for Internet of Things applications based on the technologies of the modern Web as a solution proposal for both issues. We start by describing the elements of the framework, and then discuss the user-interaction model by using a case-of-study scenario illustrating the capabilities of our contributions.

1 Introduction

In recent years, the Web has quickly evolved from a read-only source of information into a very dynamic applications platform. In the modern Web, applications no longer provide data and functionality only to human users but also to other systems and applications. Because of this situation, any user with the proper knowledge has now the possibility to build upon existing Web services and data feeds to put together, piece by piece, the needed support for customized application scenarios. This possibility led to the introduction of mashups [1,2], a concept originated in music that in computer science refers to applications created by mixing and rearranging different sources and service APIs. One of the key technologies involved in mashup systems and into the design of the last generation of Web services are the resource oriented architectures, or more commonly, RESTful Web services [3,4]. Web services following the architectural principles of REST use HTTP as application protocol, providing a very basic and simple communication platform for applications. A very common example of mashup is a Web application that uses Google Maps to show the location of the most recent pictures or events related to a specific geographic region. The Google Maps API [5] is used to draw a map and to mark positions on it, other services are queried to retrieve the list of items to show. Mashup systems can be used to combine user-interfaces, data, and functionality.

Another, more important, development is the "Internet of Things" vision [6-8] which seeks to interconnect the physical objects in everyday life through the Internet in effective, practical, and inexpensive ways. The connectivity of the integrated environment of devices of the Internet of Things together with REST-based systems will form the foundation of smart environments in which sensing and computing could be mashed-up in ways not possible before.

Several examples exist today that demonstrate how useful are REST-based platforms for different application domains. The smart grid scenario is one of the most illustrative examples as benefits are clearly identified. A good reference of a smart grid is the Smart Energy 2.0 specification of the ZigBee alliance. Smart Energy 2.0 [9] is a REST-based standard used by devices from different vendors to seamlessly interoperate with the objective to monitor, control, inform, and automate the delivery of

* Correspondence: avilesl@cicese.mx
Computer Science Department, CICESE Research Center, Carretera Ensenada-Tijuana 3918, Ensenada, Mexico

energy and water. The IP-based communication of this standard allows devices to work over Wi-Fi, HomePlug, Ethernet, and other communication channels, and, as resources are modeled after a RESTful interface, the only thing that must be standardized are how URIs and data is modeled. Another example of a successful REST-based platform is Pachube [10]. This platform provides a centralized repository of sensorial data for any user interested on sharing readings. Resources on the platform, consisting on the readings, are consumed through a REST interface and data is modeled after an standard to describe sensorial information. Pachube is also an instance of something that will be very common on the Internet of Things. Data will progressively cease to come from direct intervention of the user, through keyboards, touch-enabled surfaces, or other input devices, and will start to come from sensors and context induced by the natural interaction of the user with environments.

There is a large number of issues that must be resolved in the current Internet before it can be used as a proper infrastructure for the next generation of application services. This seems to be one of the reasons why most of the research on the Internet of Things has been focused into improving the current networking infrastructure and into providing the access for physical devices through the Internet. There are two important issues that haven't yet been fully addressed. They are: how the services and data available in the Internet of Things will be modeled to facilitate its composition and abstraction, and, how will users interact with these new smart environment applications composed by pieces of independent functionality.

In the following Section, we discuss some of the issues on the aggregation and composition of REST services and propose an architecture that can be used as a point of reference for platforms that presents the basic mechanisms needed to support mashups for the Internet of Things. In Section 3 we discuss the interaction of users with smart environments and present our interaction-model based on an mashup editor from which the environment functionality is changed or improved. In Section 4, we present a case of study scenario where the architectural elements and the user-interaction is illustrated. Section 5 includes a discussion of the related study and how our contributions differ from other proposals. Finally, in Section 6, we conclude with some final remarks and outline the future study.

2 A framework for mashups

The Internet of Things can greatly benefit from mashup technology. The common functionality for a number of scenarios in a particular domain can be identified and isolated into single, language agnostic, and independent Web services that can be later combined into new applications. Mashups not only offer a solution for composition, they can be also used as a mean to delegate processing for data-mining or aggregation, and generate new resources that are more manageable and scalable. There are some studies that are starting to bring services in representation of physical entities into mashups. The terms ubiquitous computing mashups [1] and physical mashups [11] are being used to refer to those efforts. In a similar way, there are other works trying to bring the notion of time, place, and other contextual information into traditional Web applications (for instance, the W3C's UWA Group [12]). Our architecture offers a comprehensive solution for mashups in the Internet of Things as it not only features the access to physical entities, but also provides the means for service discovery, event notification, aggregation of resources, deployment of applications, and importantly it is designed around the an user-interaction model based on ambient-intelligence (AmI) applications, as described in Section 3.

As with any infrastructure that allows the creation of mashup applications, our architecture features the following set of mechanisms:

- A way to describe each service in terms of their functionality.
- A way to discover the services that are available in the environment the user is in.
- A way for the services to communicate with interested peers when an event occurs.
- An easy way to combine data and functionality from services.

Our framework consists of four main elements: services in representation of data, logic, and physical entities, a hub service used for notifications, a service-discovery mechanism, and execution engines (see Figure 1). The rest of this section describes each element in detail.

2.1 Services description

Each element in our architecture consists of a RESTful Web service. One of the reasons why this kind of service

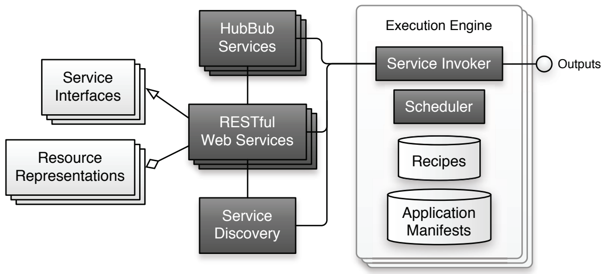

Figure 1 An overview of the framework showing all its elements.

has been used in recent services-based platforms is that they are more suitable for *ad hoc* integration [13]. REST services do not need special proxy objects to handle service calls as they are accessed through simple HTTP requests; a feature available in most modern programming-languages. Interacting with a RESTful Web service is no different than opening a webpage, or sending a form. Services provide resources (e.g., webpage, data file, image, etc.) when receiving HTTP GET requests, and handle changes on the resource by receiving POST, PUT, or DELETE requests.

One of the main issues with RESTful interfaces is that they lack a well-defined standard description language because of their own architectural restrictions. A REST-based API should not define fixed resources names, so the complete interface of a service could not be acknowledged a priori. This architectural constraint states that "hypermedia should be the engine of the application state" [3]. This is, after an initial request, the content of the response must include a relationship of further URIs that can be used to continue requesting resources according to the application flow. For example, if a user is searched on a bank's database, the resource containing the user's information should contain URIs for each of his bank accounts. However, we do need to define what starting URIs the clients could use. In the bank example, we would need to know where to send the request for the user search.

In our framework, a service is described in terms of its functionality by using a WADL document [14] containing pointers to just the starting URIs for interaction. The same document describes how data will be shared, this is, how it should be parsed and consumed. A full service is not completely described by a single WADL document. Instead, a service can implement multiple interface specifications, each stored on individual documents (see Figure 2). The intention is to simplify the specification of service compositions and to bring flexibility by allowing a service to perform richer, composed tasks. The service also needs to be registered on the service discovery mechanism, and, in order to multiply the opportunities of interoperability; all the resources provided by services in the platform must be represented in most of the following formats:

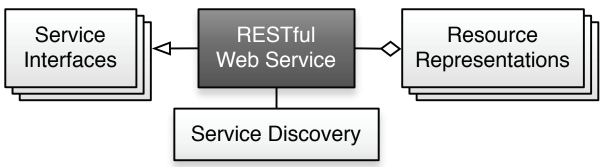

Figure 2 The structure of a service in our framework.

- *HTML*. A Web page designed to be used by a human user. This representation must provide access to all the functionality offered by the service.
- *XML*. A valid representation in an XML schema related to the functionality provided by the service.
- *JSON*. A JSON file containing the same data as the XML representation. Services featuring this data format must also provide a JSONP alternative.
- *ATOM*. An Atom feed whose items are augmented with XML tags of the related XML schema.

2.2 Service discovery

The service discovery mechanism of our base platform should be based on any protocol that can identify networking services on a local network. During prototyping we used Zero configuration networking (Zeroconf) [15], a protocol originated on a IETF working group. We use the capabilities of Zeroconf to advertise services. To do so, once a service is ready to receive requests it must be registered in the environment by broadcasting its description through the Zeroconf implementation. The description of a service consists of the following attributes: the service name, a human-readable description, the host name and port number of the network resource, a base URI in which the REST endpoints are contained, the URL of the hub service that is used to broadcast event notifications (explained in Section 2.4.4), and the interfaces the service implements (URLs to WADL specifications). The Zeroconf protocol is currently used by many printer vendors, Wi-Fi cameras, and others. Users of the protocol are expected to find a device as soon as it joins the network, much like an USB device connecting to a computer.

2.3 Event notification

While not really part of RESTful architectures, our architecture defines as events the notifications about the update of a resource sent to interested entities. The event notification mechanism allows the interchange of context information and control messages by alerting services running in the platform when a resource containing such data is updated.

The design of the mechanism is based on the PubSubHubbub protocol [16], or Hubbub for short, which introduces a special entity called *hub* into the traditional publish/subscribe communication model. In Hubbub, once that a consumer makes its first request for a provider's resource, the response representation contains a link pointing to the hub service used by the provider to advertise resource updates; consumers interested in receiving notifications for that particular resource must send a subscription request to the linked hub. The hub stores a local copy of the representation that will be updated after it receives a ping request from the provider notifying him that new data is available. If this does not happen within a certain time window, the hub requests the resource by itself and updates its local copy; whenever the resource is updated, the hub notifies each of the subscribers which can then request the hub's updated representation.

At the moment, Hubbub only supports resource representations in the Atom and RSS feed formats as they allow the documents to contain links to related resources, in this case, to hub services. However, we extend these capabilities by adding support to other possible representations. References to hub services are expected as links within documents or as an attribute in service's descriptions. In this way, services that cannot include links in the representations because of limitations in data formats, can point to a hub in the service's description. Additionally, if a service declares to implement the Hubbub protocol but does not point to a hub in any ways, an execution engine will search for an available hub service in the environment to use it for event notifications during the application flow.

2.4 Execution language and engine

In mashup systems, an application is actually a composition of services put together to provide the logic needed for an application. In order to build such applications, we need first to describe the service composition using whatever language is available in the platform that will run the mashups. As stated before, the definition of a formal description for REST services has yet to be standardized; general composition languages such as WS-BPEL [17,18] or EMML [19] which are highly dependent on the full service descriptions are having difficulties to offer an easy to use solution for unexperienced developers. Being general-purpose composition languages, they are very fine-grained systems so developers must be aware of how requests, representations, and the individual logic will be mixed up together. To confront all these issues, we propose a simple composition language in which the application logic is expressed by describing the dependencies between coarse-grained component services. Each composition step is performed accordingly to the interfaces that the involved services implement, so the full application logic can be expressed as a simple and straightforward pipelining logic graph. Applications in our framework are described by documents called *application manifests*, which define:

- The services involved in the composition (which can be specific or generic) and their description.
- The flow between the services.
- The mashup outputs where the final data or functionality will be served.
- Application control data such as execution schedule, authentication credentials, and other metadata.

The scenario of use for traditional mashup systems is rather simple: once a composition is defined, it is sent for execution to a server; the server runs the composition and exposes any resources generated throughout the application flow. Systems providing ubiquitous or physical mashups, in the other hand, should integrate additional special characteristics. In our platform:

- An application manifest is not only run on publicly available servers, they can run on local closed environments, personal mobile devices, and so on.
- The user can be a part of the execution flow (e.g., to select between alternatives).
- Composition could include previously unknown services that implement certain generic interfaces.
- Notification events from physical devices or context providers are involved in the application flow.
- The execution of applications can be scheduled.
- Users can share and collaboratively design application manifests using the different available execution engines (with the appropriate permissions).
- Services representing objects in the real world are managed through permissions and access control.

Application manifests are run by one of the special entities in the platform: the execution engine services. An execution engine is in charge of managing the described application flow by identifying providers, requesting resources, receiving related event notifications, and providing output resources to finally deliver the target functionality. Its architecture is composed of five main components (see Figure 3) which are in fact implementations of one or more of the core services interfaces. To further discuss the functionality of each component and the overall engine's scenario of use, lets consider as example an application that generates a map widget with the last sensor readings from two wireless sensor networks. Figure 4 shows the graphical representation of the services involved and the application flow that will end

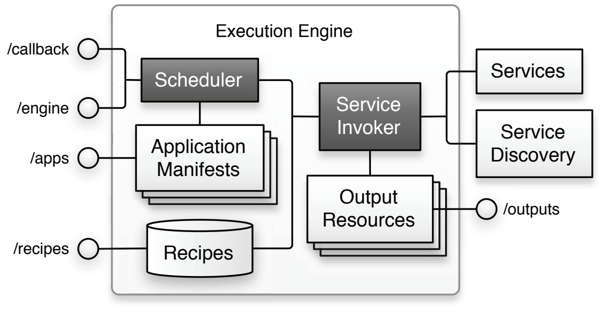

Figure 3 Architecture of the execution engine, one of the framework elements.

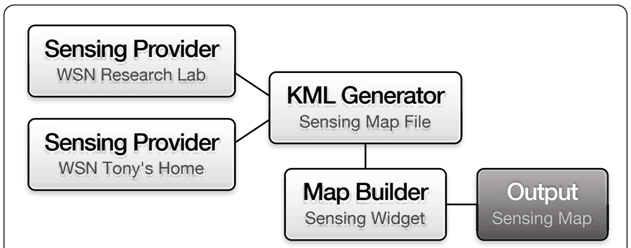

Figure 4 The graphical representation of an application manifest.

up in the generation of an output resource containing the representation of a events-enabled widget that can be embedded in Web pages.

The following sections detail how mashup applications are executed.

2.4.1 Deploying manifests

The scenario starts with the submission of the example application manifest (see Figure 5) to an execution engine. The manifest files deployed on an engine are managed through the/`apps` endpoint.

2.4.2 Running applications

Once an application has been deployed, the Scheduler component will start its execution at the time indicated in the manifest. Applications can be started immediately, at a certain time, or just put on hold. Also, for simple application flows, where real time notifications are not involved, the manifest can declare additional starting times or a repetition scheme. Whether the applications are running or not, they can be controlled by the direct interaction with the Scheduler component using the/`engine` endpoint.

```
{
   "interfaces": [
      { "id": "i1", "url": "http://…/push-publisher.wadl" },
      { "id": "i2", "url": "http://…/push-subscriber.wadl" },
      { "id": "i3", "url": "http://…/sensing-provider.wadl" },
      { "id": "i4", "url": "http://…/kml-generator.wadl" },
      { "id": "i5", "url": "http://…/widget-generator.wadl" },
      { "id": "i6", "url": "http://…/map-builder.wadl" } ],
   "services": [
      { "id": "s1", "name": "WSN Research Lab",
         "implements": [ "i3", "i1" ], … },
      { "id": "s2", "name": "WSN Tony's Home",
         "implements": [ "i3", "i1" ], … },
      { "id": "s3", "name": "Sensing Map File",
         "implements": [ "i4", "i2", "i1" ], … },
      { "id": "s4", "name": "Sensing Widget",
         "implements": [ "i6", "i5", "i2", "i1" ], … } ],
   "outputs": [
      { "id": "o1", "name": "Sensing Map", … } ],
   "flow": [
      { "from": [ "s1", "s2" ], "to": [ "s3" ], … },
      { "from": [ "s3" ], "to": [ "s4" ], … },
      { "from": [ "s5" ], "to": [ "o1" ], … } ],
   …
}
```

Figure 5 An application manifest in JSON format.

2.4.3 Logical recipes

To actually "run" mashups, the application flow as described in the manifest is first transformed into a logical dependencies tree. Then, the Service Invoker component examines each node and tries to match its current conditions against the requirements of a *logical recipe* which is used to handle the actual processing required for that step in the mashing process. The conditions of a step refer to: the description of the service represented by the current node, the presence or availability of the same service, and the services used by this node. The requirements of a recipe declare: the interfaces that the target service must implement, the interfaces implemented by the services involved, and how recent a resource from such services must be in order to be used for processing the logic provided by the recipe. We based the concept of recipes on the fact that many of the composition tasks present very similar situations. For instance, in our example application, we could expect that the functionality expressed by the connection of every instance of the KML Generator interface to a Map Builder interface is to generate a map widget highlighting hotspots from the file. If that is not the case, we could easily add more recipes to an execution engine by posting a new recipe resource (which includes requirements and the script to handle the composition step) into the/`recipes` endpoint.

2.4.4 Handling requests and events

To fulfill the composition step accordingly to the recipe, the service invoker casts all the involved requests for resources. As responses arrive, they are mixed (by the recipe's scripting) into a new resource that will be stored in the current node of the full logical tree. To handle events, the Service Invoker subscribes itself to the resources for which it requires notifications by using an intermediary hub service. Here is where service discovery is used to find a hub in case one of the services declares to implement the push-subscriber interface but does not include a link to one. After subscription, event notifications arrive to the/`callback` endpoint of the execution engine, where the scheduler identifies the target application manifest and activates the execution of the node where the subscription was made. Whenever a step of the application flow is completed, service invocation is activated on every child node to possibly update any derived resources.

2.4.5 User inputs

It is handled by services whose resources represent user interactions. As other publisher services, they implement the push-publisher interface so they can halt the execution of an application manifest until a response from a user is received.

2.4.6 Output endpoints

The resources generated on each composition step are stored on the services dependency tree. Sometimes, the application's complete functionality is performed by the accumulation of each individual service request, other times, the final objective of a mashup is to provide new resources. To support this, the application manifest includes outputs declarations into the application flow. In the example application (see Figure 5), all connected services end up in an output which generates source code that users can copy and paste into Web pages. To interact with outputs, each execution engine exposes the outputs of the applications in the/`outputs` endpoint. Output resources are not just plain, simple documents. An application could be kept running and updating those resources. In the example, sensing readings will be constantly notified. Widgets on the platform are supposed to receive PubSubHubbub updates on the client's side, so outputs are always updated, living documents. Additionally, an output resource can be declared to implement interfaces, as a normal service, so the discovery mechanism can identify and include them into compositions.

2.4.7 Sharing applications

Multiple users can simultaneously use execution engines. To collaboratively design applications, they could update the same manifests by interacting with the/`apps` resources. We have included a special sharing alternative: the ability to export application manifests. Exported manifest contain along with the original document, the definition of all the recipes found suitable by the execution engine at hand. So, if a developer wants to take an application into another execution engine it will be executed on the same way (provided that both engines are implemented with the same scripting technology).

In the following section, we present our user-interaction model based on the framework described above. On section 4, we discuss a sample scenario with the settings of the architectural elements and the interaction following the complete model.

3 The user interaction with smart environments

Besides providing a good infrastructure foundation for smart environments, we must address the question of how users are expected to interact with such environments. We think that, in order to illustrate this interactions in smart Web-based environments on the Internet of Things, we can use scenarios depicted on the AmI [20] and ambient-assisted living (AAL) visions.

The term AmI is used to refer to smart environments that are sensitive and responsive to the presence of users. Hardware devices and software services in smart environments are expected to work together to support users in their everyday activities in easy, natural ways hiding at the same time the network and other low-level details that support the environment. The AmI paradigm is characterized by systems and technologies that are [21,22]:

- Embedded. Many small-networked devices are seamlessly integrated into the environments.
- Context-aware. That can recognize users and their situational context.
- Personalized. That can be tailored towards the user's needs.
- Adaptive. That can change in response to user actions or other events in the environment.
- Anticipatory. That can anticipate users' desires without conscious mediation.

The AAL term refers to the application of ambient technology to assist individuals in performing everyday activities. This term is very similar to AmI, but the AAL systems present the following additional characteristics:

- Invisibility. They are embedded in clothes, watches, eyeglasses, and other day-to-day items.
- Mobile. They can be transported by the user and have the ability to act in different environments.
- Spontaneous. They can communicate dynamically with many previously unknown devices.
- Heterogeneous. They integrate several different types of hardware and software entities.
- Proactive. They can act in behalf of the user after the user activities inferred from his behavior.
- Natural. They communicate the user in natural ways such as voice and light.

To further illustrate how the user interacts with smart environments according to the concepts of the AmI and AAL visions consider the following scenario:

Mr. Smith is a senior of advanced age who, due to the physical inabilities that come with age, needs a wheelchair to move around his house. Whenever Mr. Smith is close to a door, he presses a button on his watch and the door opens, the same happens when he is close to a window. When Mr. Smith needs the assistance of another person, he presses the button again, and an email and IM alert is sent to the closest person around the area where he is providing him with Mr. Smith's location and identification details.

In the above scenario, the smart environment is supporting the user but the user by himself does not actively alters the functionality the environment is ready to provide. This scenario requires someone, a "power user", to set up how services in representation of doors and windows should act when the button on Mr. Smith's watch is pressed. In the following section, we present our infrastructural model in which we detail how the interaction of users would be in both cases: the passive, in which users take advantage of services and applications available in environments, and the active, in which power users arrange the services available in an environment to define new applications.

Following the concepts and characteristics introduced above, we can identify two scenarios of interaction between the user and the smart environment. One, in which the environment implicitly supports the users according to the functionality of existing applications, and other, in which the user explicitly creates or modifies an application on the environment. For the first scenario, our framework considers users carrying devices with them that allows the environment to detect and track their presence. For the second scenario, the user interacts with the environment through a mashup editor he can use to create new applications using the available services or even previously created applications.

As explained above, in our base platform a mashup application is coded by means of an application manifest that specifies the services involved, the connections between them, and the data flow though the configuration or assembled by using the mashups editor; this editor is a Web-based system to create application manifests through simple activities such as dragging and dropping graphical representations of the involved services to accomplish the specification of configurations that support a desired scenario. In our interaction model, a user must carry a device that provides him with personal services (such as localization, contacts, notification, and others), runs mashup applications, and allows access to the infrastructure through the mashups editor. The personal applications and services are shown on the left sidebar of the editor (see Figure 6:1-3). The platform infrastructure automatically discovers the services that are provided on the user environment and the applications that are currently running (those are shown on the right sidebar, Figure 6:4-5)). Additionally to the basic personal services, the user can add external services such as his Facebook or Twitter account (Figure 6:3). The editor also allows interaction with remote environments (Figure 6:6-7). The area at the center of the editor (Figure 6:8) is the instantiation area where the services are dropped, interconnected, and then stored into an environment or in the personal device as mashup applications.

4 Case of study

To better illustrate how our base platform and user-interaction model work together let's consider the following scenario. In a living room, there are several devices that provide RESTful interfaces to their resources as a way to consume data or trigger functionality. The living room has three types of communication channels (see Figure 7): a 802.15.4-based protocol, a HomePlug network, and a Wi-Fi router. The lights in the room and the shade on the window communicate with the 802.15.4-based protocol, the home entertainment center and the ceiling lamp use HomePlug, and users use the Wi-Fi connection to talk with all the devices via the mashup editor on the device of users. All three-communication channels are connected together with a Wi-Fi/HomePlug/802.15.4 bridge in the same room. Each device is running a REST service that is developed following the characteristics of a service in our base platform. These are, each service provide access to resources through an RESTful API, implement one of the generic interfaces, and are broadcasting their availability into the service discovery mechanism. Additionally to the device services, there is one execution engine and a directory service.

As discussed above, the interaction model has two different approaches. In one, applications on the execution engines react to changes on the environments initialized by the presence of the user or other related resources. In the other, the user actively makes changes on the applications in execution engines by using the mashup editor. In this scenario there are multiple possible applications. One of them is one that turns out the lamp on the corner, closes the window shade, and dims out the lights in the room whenever the home entertainment center is playing a movie. Other possibility is one application that turns on the lights and closes the shade when the day starts growing dark. These applications are really simple and the user might be able to write the needed recipes by himself. But, there could be situations, for example, when the user purchases a new device, that could require the user to obtain a recipe from a website, or by having the assistance or a technician. Execution engines will use devices and services on application manifests but the user or any other entities can still use them.

Lets discuss the implementation of the scenario in which the entertainment center plays a movie, dims out the lights and closes the window shade. The first thing we need is to identify what is the dependency between devices. In this scenario, the starting point is the entertainment center, whenever the resource that contains the status of the device changes, we need to receive an event notification in order to continue other actions. So, before using the mashup editor to create the application manifest, we must be sure that we count with the proper service implementations. First, the entertainment center must provide a resource containing information about its current status through a REST interface. Second, we need at least one hub service available so we can subscribe to the updates of the entertainment center resource. And third, the light dimmer and the window shade must count each with a service accepting updates of its status resource so if we update it a physical action

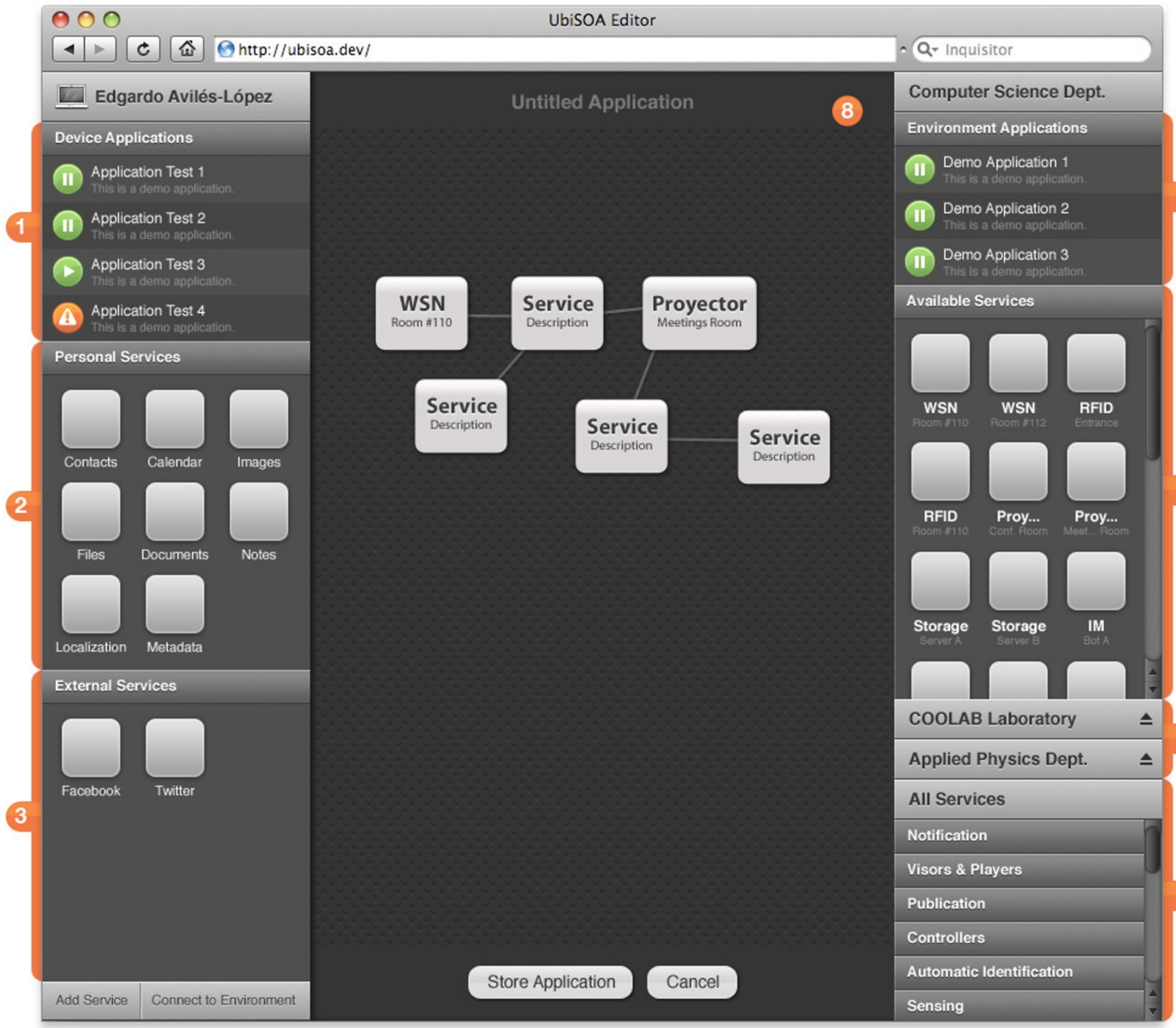

Figure 6 The mashup editor build upon the framework.

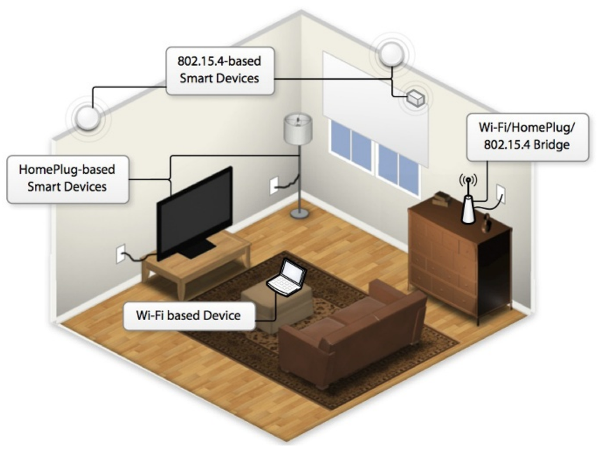

Figure 7 The devices involved in the case of study scenario.

will occur. Once the services are all running and detected by the service discovery mechanism, they will appear on the mashup editor the user carries with him. Here, he just needs to drag each service representation and to draw a line between the dependencies. Suppose this is the first time the user needs this dependency scenario. The editor will not find any recipes for the needed functionality. So, the user will need to write a recipe for each connection on the dependency tree. The needed recipes for this functionality are:

- The event subscription to the entertainment center status resource using any hub service available at the time on the environment.
- The request for the lights to dim out accordingly to the updated entertainment center status resource.

- The request for the window shade to close accordingly to the updated status resource.

We have implemented a working prototype[a] of our framework and a basic version of our mashup editor. The implementation of the different platform mechanisms are a combination of RESTful Web services written in PHP and Java (using the Restlet framework [23]) with a service discovery based upon Bonjour [24]. The editor is implemented on Webkit and uses JavaScript to handle the user interface. Recipes in our execution engine mechanism are an extension of PHP that includes methods and special objects to reduce the amount of source code written by users to define behavior. For example, to send a POST request to a REST service the user only needs to write `post($target, $data)`. The keyword `$target` refers to an special object that contains all the data related to the identification of a service, requests to the service discovery mechanism returns this kind of objects. The keyword `$data` refers to a resource representation that can be easily translated to a JSON or XML document as the recipe needs. Using the prototype of our infrastructure we have conducted other studies related to AAL applications and scenarios [25].

5 Related study

The following studies present some similarities with our proposals but differ in how services are handled towards composition or in how the user-interaction is when creating applications through mashups.

López de Ipiña et al. [26] present Sentient Graffiti (SG), a platform following the Internet of Things concept where users accompanied by mobile devices or Web browsers can browse, discover, search, annotate, and filter surrounding smart objects in the form of Web resources. In this infrastructure, users carry an SG client which they can use to add annotations to objects on spatial regions, they can also browse (physically or virtually) an environment to consume the available annotations. While this effort considers AmI environments, specifically, the ambient-intelligent systems requirements, it considers the services as passive entities, requiring the users to explicitly initiate the interaction with smart objects.

Vazquez et al. [27] present the concept of "social device" and talk about their study on implementing a couple of prototypes to evaluate the potential of their framework to support the integration of smart devices into the Web. Their focus differs to ours in the sense that the authors want to make devices interact between each other in the same way people interact through Web 2.0 social services. We are more focused on the interaction on site and in how users would interact with smart environments by easily creating mashups.

Guinard et al. [28] analyze the integration of real world devices with the Web by turning them into RESTful resources. They propose two methods of integration, in the first one, they describe how an actual Web server can be implemented on tiny embedded devices and, in the second one, when computational resources are too limited, they propose the use of an intermediate gateway that can offer a unified RESTful API to the devices by hiding the actual communication protocols used to communicate with them. They also exemplify how these services can be later used in mashup applications. Our contributions go beyond than just providing access to physical entities or simplifying the development of mashups from RESTful resources. The design of our framework takes into account special requirements for supporting AmI scenarios. Some of such requirements are: the need to deliver timely notifications on a particular context change, the possibility to deploy an application into the background of an environment, the incorporation of generic services into compositions for which the actual service to be used is unknown, and the possibility to involve an user's choice on runtime into the application flow. The same author has also experimented with a mashup editor for users to create applications [29]. However, the editor runs applications on a Web browser as it is based upon a Firefox plugin. Our mashups can run on specialized devices or in execution engines deployed on different environments, and more importantly, they not limited by the capabilities of JavaScript, this is, they can use different protocols or engines as a way to compose resources.

Zhang et al. [30] present the advanced Internet of Things (AIoT) paradigm based on the unified object description language (UODL) proposed by the same authors. UODL allows identifying and interconnecting every object and event with a standard format, making available all the information and events related to the objects, and make the control by management and third party entities easy and flexible. The AIoT paradigm and our framework both take care of identifying objects, handling discovery, and providing a deployment mechanism but the main difference is the way objects are modeled. Our framework is based upon REST and every single framework element is a service that provides resources via simple HTTP requests. Because of this, any existing third party service using HTTP as application protocol can be easily integrated into our framework. In the same way, our framework can directly use the same technologies for security and data privacy as the available on the traditional Web. Finally, AIoT doesn't directly involve the user in the environments where the applications are deployment, our framework is designed to allow users to crate or modify such applications following the mashups concept.

6 Conclusions

In this article, we have talked about how the Internet is progressing towards the Internet of Things and discussed two important problems, which are, how the resources and functionality of the different next generation services would be composed, and how the user interaction would be in terms of the access to the resources provided by the environment, and in terms of how they would create new applications for their particular support demands. To handle both issues we propose a base platform architecture and an user-interaction model based on the AmI applications and AAL concepts.

In order to corroborate our contributions, we have implemented a working prototype of our platform and used it to build test applications for a variety of scenarios. We also used the implementation to conduct a usability study with a small developers group consisting of engineers both experts and novel in Web programming. The test was based on the cognitive-dimensions framework [31] and the participants were requested to build an application using the mashup editor. Preliminary results show users have no problems identifying each of the platform elements and what is the role of each one in providing the final support of the requested applications.

The Web-based support for the Internet of Things seems to be the best way to promote interoperability and easiness of access to resources. Certainly, it is not always the best way for applications scenarios where quality of service, latency, and consumption of energy are important issues. Experimentation conducted with our prototype on different scenarios seems to support these thoughts. Fortunately, there is a very active research on protocols upon where the Internet of Things could rely upon. One of such protocols is CoAP [32] which allows low-end, 8-bit devices to use HTTP-like intercommunications. One of the many benefits of CoAP is that it is easily translatable into the full HTTP protocol enabling the valuable Web-based support.

With the introduction of our mashups-based platform architecture and user-interaction model we intend to provide a base upon where more advanced platforms and applications could be designed and built. We are working into conducting further evaluations to better assess the suitability of our base platform in providing support for smart environments.

Endnote

[a]The implementation is open and can be downloaded from: http://www.ubisoa.net/.

Acknowledgements
The study presented in this article was funded with grants from CONACyT, the Mexican Council for Science and Technology.

Competing interests
The authors declare that they have no competing interests.

Received: 3 October 2011 Accepted: 2 March 2012
Published: 2 March 2012

References

1. B Hartmann, S Doorley, SR Klemmer, Hacking, mashing, gluing: understanding opportunistic design. Pervasive Comput. **7**(3), 46–54 (2008)
2. M Alibinola, L Baresi, M Carcano, S Guinea, Mashlight: A Lightweight Mashup Frame-work for Everyone, in *Proceedings of the 18th international conference on World Wide Web, WWW'09*, (Madrid, Spain, 2009)
3. RT Fielding, Architectural styles and the design of network-based software architectures, (PhD thesis, University of California, Irvine, 2000)
4. L Richardson, S Ruby, *RESTful Web Services*, (O'Reilly Media, Inc., Sebastopol, USA, 2007)
5. Google Maps APIhttp://code.google.com/apis/maps
6. N Gershenfeld, R Krikorian, D Cohen, The Internet of Things. Sci Am. **291**(4), 76–81 (2004). doi:10.1038/scientificamerican1004-76
7. CR Schoenberger, The Internet of Things. Forbes Mag. **169**(6), 155–160 (2002)
8. JS Rellermeyer, M Duller, K Gilmer, D Maragkos, D Papageorgiou, G Alonso, The software fabric for the Internet of Things, in *Proceedings of the 1st international conference on the Internet of Things, IOT'08*, vol. 4952. (Springer-Verlag, Berlin, Heidelberg, 2008), pp. 87–104
9. ZigBee Alliance, ZigBee Smart Energy 2.0 Specification. http://www.zigbee.org/ (2011)
10. Pachube, Real-Time Open Data Web Service for the Internet of Things. http://pachube.com/ (2011)
11. D Guinard, V Trifa, E Wilde, A resource oriented architecture for the Web of things, in *Proc of IoT 2010 (IEEE International Conference on the Internet of Things)*, (Tokyo, Japan, 2010), pp. 1–8
12. W3C Ubiquitous Web Applications Grouphttp://www.w3.org/2007/uwa/
13. C Pautasso, O Zimmermann, F Leymann, RESTful Web services vs. "big" Web services: making the right architectural decision, in *WWW'08: Proceeding of the 17th interna-tional conference on World Wide Web*, (Beijing, China, 2008), pp. 805–814
14. MJ Handley, Web application description language (WADL), (W3c member submission, Sun Microsystems, Inc, 2009) http://www.w3.org/Submission/wadl/
15. Zero Configuration Networking (Zeroconf)http://www.zeroconf.org/
16. PubSubHubbub Projecthttp://code.google.com/p/pubsubhubbub
17. OASIS Web Services Business Process Execution Language (WS-BPEL)http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
18. C Pautasso, RESTful Web service composition with BPEL for REST. Data Knowledge Eng. **68**(9), 851–866 (2009). doi:10.1016/j.datak.2009.02.016
19. OMA EMML Documentationhttp://www.openmashup.org/omadocs/v1.0
20. E Aarts, H Harwig, M Schuurmans, *The Invisible Future: The Seamless Integration of Technology into Everyday Life*, (McGraw-Hill, Inc., NY, USA, 2002)
21. Weber W, Rabaey JM, Aarts EHL (eds.), *Ambient Intelligence* (Springer, New York, USA, 2005)
22. J Nehmer, M Becker, A Karshmer, R Lamm, Living assistance systems: an ambient intelligence approach, in *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, (Shanghai, China, 2006), pp. 43–50
23. Restlet, RESTful Web Framework for Java.http://www.restlet.org/
24. Apple Bonjourhttp://developer.apple.com/bonjour
25. E Avilés-López, JA García-Macías, I Villanueva-Miranda, Developing ambient intelligence applications for the assisted living of the elderly, in *Intl Conf on Ambient Systems, Networks and Technologies (ANT 2010)*, (Paris, France, 2010)
26. D López-de-Ipiña, JI Vazquez, J Abaitua, A Web 2.0 platform to enable context-aware mobile mash-ups, in *AmI'07: Proceedings of the 2007 European Conference on Ambient Intelligence*, vol. 4794. (Springer-Verlag, Berlin, Heidelberg, 2007), pp. 266–286
27. J Vazquez, D López-de-Ipiña, Social devices: autonomous artifacts that communicate on the Internet, in *The Internet of Things*, pp. 308–324http://dx.doi.org/10.1007/978-3-540-78731-0_20 (2008)
28. D Guinard, V Trifa, Towards the Web of things: Web mashups for embedded devices, in *WWW'09: Proceedings of the 18th international conference on World Wide Web*, (Madrid, Spain, 2009), pp. 678–683

29. D Guinard, Mashing up Your Web-enabled home, in *Adjunct Proc of ICWE 2010 (In-ternational Conference on Web Engineering), Vienna*. **6385**, 442–446 (2010)
30. L Zhang, N Mitton, Advance Internet of Things, in *iThings*, (Dalian, China, 2011) http://hal.inria.fr/inria-00634290
31. TRG Green, M Patre, Usability analysis of visual programming environments: a cognitive dimensions framework. J Visual Lang Comput. **7**, 131–174 (1996). doi:10.1006/jvlc.1996.0009
32. Constrained Application Protocol (CoAP)http://tools.ietf.org/html/draft-ietf-core-coap

doi:10.1186/1687-1499-2012-79
Cite this article as: Avilés-López and García-Macías: **Mashing up the Internet of Things: a framework for smart environments.** *EURASIP Journal on Wireless Communications and Networking* 2012 **2012**:79.