Giving RFID a REST: Building a Web-Enabled EPCIS

Dominique Guinard
Institute for Pervasive Computing, ETH Zurich
and SAP Research Zurich
Email: dguinard@ethz.ch

Mathias Mueller
Software Engineering Group
University of Fribourg
Email: mathias.mueller@unifr.ch

Jacques Pasquier-Rocha
Software Engineering Group
University of Fribourg
Email: jacques.pasquier@unifr.ch

***Abstract*—The Electronic Product Code Information Service (EPCIS) is a standard which defines interfaces enabling RFID events to be captured and queried. The query interface, implemented with WS-* Web services, enables business applications to consume and share data within and across companies, to form a global network of independent EPCIS instances. However, the interface limits the application space to the rather powerful platforms which understand WS-* Web services. In this paper we propose seamlessly integrating this network into the Web by designing a RESTful (REpresentational State Transfer) architecture for the EPCIS. Using this approach, each query, tagged object, location or RFID reader gets a unique URL that can be linked to, exchanged in emails, browsed for, bookmarked, etc. Additionally, this paradigm shift allows Web languages like HTML and JavaScript to directly use RFID data to fast-prototype light-weight applications such as mobile applications or Web mashups. We illustrate these benefits with a JavaScript mashup platform that integrates several services on the Web (e.g., Twitter, Wikipedia, etc.) with RFID data to allow managers along the supply chain and customers to get comprehensive data about their products.**

I. Introduction

The EPC Network is composed of several standards addressing issues from the RFID (Radio Frequency IDentification) tags themselves (EPC standard) to readers infrastructure and the reading middleware [1]. The way tags are encoded, read and aggregated through the whole supply chain is addressed by these standards. Furthermore, to be able to query and use recorded RFID data (i.e., traces), the EPCIS standard (Electronic Product Code Information Services) acts as a global track and trace sharing infrastructure with several, potentially interconnected, EPCIS servers distributed around the world. The EPCIS provides a simple and lightweight HTTP interface for recording EPC events. A different approach is taken to querying for these traces by other applications because the EPCIS specifies a standardized WS-* (i.e., SOAP, WSDL, etc.) interface. The WS-* type of integration architecture is well adapted to combine business applications [2], [3]. For example, it can be used to integrate EPCIS data about the status of a shipment with an ERP (Enterprise Resource Planning) application.

However, WS-* applications are complex systems with high entry barriers and require developer expertise in the domain. Hence, they are not well adapted for more light-weight and ad-hoc application scenarios [2]. Furthermore, the WS-* protocols are known to be rather verbose. Moreover, they do not fully meet the requirements of resource-constrained devices such as mobile phones and wireless sensor/actuator networks often not providing WS-* server or even client stacks [4], [5]. As a consequence, these shortcomings limit the type of applications built on top of EPCIS servers to rather heavy-weight business applications fully supporting the WS-* procotols. This is unfortunate since track and trace applications are also relevant beyond the desktop. As an example, providing an out of the box mobile access to these data might be beneficial for many users such as mobile workers. Similarly, providing direct access to RFID traces to sensor and actuator networks could enable those to react to RFID events. Finally, allowing light-weight Web applications (e.g., HTML, JavaScript, PHP, etc.) to directly access these data would enable the vast community of Web developers to create innovative applications using RFID traces.

In this paper we propose an additional integration interface, or API (Application Programming Interface) for the EPCIS upon which a different range of applications can be built. We define the following as requirements for the new API:

1) It should **lower the entry barrier for developers** and foster rapid prototyping. This allows a wider range of developers, tech-savvy user (technologically skilled people) or researchers to develop on top of the EPCIS and contributes to helping the EPC Network developer community grow.
2) It should **offer a direct access to users**. Users should be able to access the data without the need for installing another software than the EPCIS itself. They should further have means to directly extract, share and save EPC events.
3) It should offer a more **light-weight** access to the data. This enables creating applications in which the EPCIS data are directly consumed by **resource-constrained devices** without requiring proxy or translation gateways.

To fullfill these requirements, our first contribution is to specify an API complentary to the WS-* interface and based on the REST architectural style. REST is actually core to the Web and uses URIs for encapsulating and identifying services on the Web. Unlike WS-* services using HTTP only as a transport protocol, the Web implementation of REST uses

HTTP as an application protocol. As a consequence RESTful services are directly usable with well-known Web languages (e.g., JavaScript, HMTL, PHP, Python, etc.) which lowers the entry barrier for developers. The Web currently accounts for one of the most active pools of developers[1]. Bringing RFID data closer to the Web creates opportunities for new applications. Just as tech-savvy users create Web 2.0 mashups by integrating several Websites to create new applications, they could re-use RFID events to create ad-hoc, innovative applications.

Furthremore, RESTful services can be directly accessed, tested, bookmarked, exchanged from the browser, a tool that is ubiquitously available and understood (at least superficially) by a vast number of people [6]. Finally, REST is known to be more light-weight [4] than WS-* services and thus more adapted to the vast majority of resource-constrained devices which support it through simple HTTP client libraries or higher-level REST client libraries.

In this paper we present the architecture and implementation of this additional RESTful API as a pluggable software component that can be deployed on top of any standard implementation of the EPCIS without obfuscating the original EPCIS WS-* interface.

Moreover, we want to illustrate how this paradigm shift unveils a broader range of applications using EPCIS data. Thus, our second contribution is the EPC Mashup Dashboard. This Web mashup platform allows exploring EPC related data and gathering timely information about tagged objects from several Web services such as Twitter, Wikipedia or Google Maps. Product or supply chain manager can use this tool as a business intelligence platform, helping them to better understand and visualize the supply chain. Likewise, customers can better understand and visualize where the goods came from, what they really are, what other people think about them, etc.

Finally, we evaluate the performance of the system. We look at the overhead created by the RESTful module and show that is it minimal.

Before looking at the architecture of the RESTful EPCIS, we briefly introduce the idea of the EPC Network as well as the basic concepts behind RESTful Web Services.

A. EPC Network and EPCIS

The EPC Network[2] is a set of standards established by industrial key players towards a uniform platform for tracking and discovering RFID tagged objects and goods. Fifteen standards are currently composing the EPC Network addressing every step required from encoding data on RFID tags to reading them and sharing their traces. We will focus on two of them as those are the most relevant in the context of this paper.

The first standard is the EPC Tag Data Standard (TDS). It defines what an EPC number is and how it is encoded on the tags themselves. An EPC is a world wide unique number. Rather than identifying a product class, like most barcode standards do, it can be used to identify the instance of a product. The TDS specifies eight encoding schemes for EPC tags. They basically contain three types of information: the manufacturer, the product class and a serial number. As an example in the tag (represented in its URI form): urn:epc:id:gid:2808.64085.88828, `2808` is the manufacturer ID, `64085` represents the type of product and `88828` an instance of the product.

One of the goals of the EPC Network is to allow sharing observed EPC traces. Thus, the network specifies a standardized server-side EPCIS, in charge of managing and offering access to traces of EPCs events. Whenever a tag is read it goes through a filtering process and is eventually stored in an EPCIS together with contextual data. In particular, these data deliver information about:

- The "what": what tagged products (EPCs) were read.
- The "when": at what time the products were read.
- The "where": where the products were read, in terms of Business Location (e.g., "Floor B").
- The "who": what readers (Read Point) recorded this trace.
- The "which": what was the business context (Business Step) recording the trace (e.g., "Shipping").

The goal of the EPCIS is to store these data to allow creating a global network where participants can gain a shared view of these EPC traces. As such, the EPCIS deals with historical data, allowing, for example, participants in a supply chain to share the business data produced by their EPC-tagged objects.

Technically speaking, a standard EPCIS is an application that offers three core features to client applications:

1) First it offers a way to capture, i.e., persist, EPC events.
2) Then, it offers an interface to query for EPC events.
3) Finally, it allows to subscribe to queries so that client applications can be informed whenever the result of a query changes.

There exist several concrete implementations of EPCISs on the market. Most of them are delivered by big software vendors such as IBM or SAP. However, the Fosstrak [1] project offers a comprehensive, Java-based, open-source implementation of the EPCIS standard.

II. Related Work

The great potential of the EPC network for researchers in the ubiquitous computing field has led to a number of initiatives trying to make it more accessible and open for prototyping. The authors of [1] initiated the Fosstrak project, which is to date the most comprehensive open-source implementation of the EPC standards. The Fosstrak EPCIS is an open-source implementation of a fully-featured EPCIS. This project is suitable for prototyping [1] but it implements the standard WS-* interface which closes the EPCIS to a number of interesting use cases such as direct use from simple Web languages or usage on resource constrained devices.

To overcome these limitations, researchers started to create translation proxies between the EPCIS and their applications.

[1] http://langpop.com

[2] http://epcglobalinc.org/standards/architecture

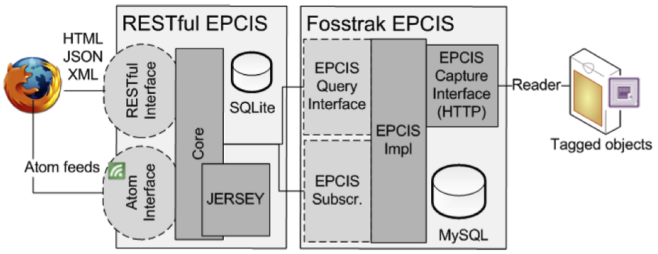

Fig. 1. Architecture of the RESTful EPCIS based on the Jersey RESTful framework and deployed on top of the Fosstrak EPCIS.

In [7] the authors present an implementation of such a proxy. The "Mobile IoT Toolkit" offers a Java servlet based solution that allows to request some EPCIS data using URLs which are then translated by a proxy into WS-* calls. This solution is a step towards our goal as it enables resource-constrained clients such as mobile phones to access some data without the need for using WS-* libraries. Nevertheless, the proxy is directly built on the core of Fosstrak and thus does not offer a generic solution for all EPCIS compliant system. Furthermore, the protocol used in this implementation as well as the data format is proprietary which requires developers to learn it first.

In the "REST Binding" project[3] a translation proxy is implemented, similarly to [7] it proposes using URLs for accessing the EPCIS data but these data are provided using the XML format specified in the standard. While this is an important improvement, the proposed protocol does not respect the REST principles but implements what experts sometimes call a REST-RPC style [8]. As the connectedness and uniform interface properties do not held, an EPCIS using this interface is not truly integrated to the Web [3], [8]. For instance, it does not offer alternative representations (e.g., JSON) and resources can't be browsed for.

III. RESTful EPCIS

The architectural principle that lies at the heart of the Web is the Representational State Transfer (REST) as defined by Roy Fielding [9]. It shares a similar goal with more well known integration techniques such as WS-* Web services (SOAP, WSDL, etc), which is to increase interoperability for a looser coupling between the parts of distributed applications. However, the goal of REST is to achieve this in a more lightweight and simpler manner. In particular, REST uses the Web as an application platform and fully leverages all the features inherent to HTTP such as authentication, encryption, scalability and caching. REST brings services "into the browser": resources can be linked and bookmarked and the results are visible with any Web browser, with no need to generate complex source code out of WSDL files to be able to interact with the service.

Traditionally, REST has been used to offer APIs on top of Websites. More recently it has been used to integrate real-world data such as data coming from sensors to the Web [10], [5]. This type of research is often clustered under the concept of "Web of Things" [11]. The core concept of these projects is to re-use and adapt Web standards and architectures to the needs of the physical world in order to create a "universal API for things". Most of these projects build upon and adapt the principles of Resource Oriented Architectures (ROAs) [8] which is a service architecture using REST and the HTTP protocol.

The goal of the project described in this paper is to create a Resource Oriented Architecture for the EPCIS. In this section we describe the core architecture supporting a large-scale Web-enablement of the EPCIS features taking a top-down approach. There are basically two ways to achieve this: either by creating a RESTful interface directly as part of a particular implementation of an EPCIS such as Fosstrak, or by creating a software module that can be plugged on top of existing EPCISs. Both architectures present advantages and drawbacks. We decided to create an independent software module as this allows the RESTful EPCIS to work on top of any standard EPCIS implementation. The direct drawback of this approach is that it creates a communication overhead. We will discuss this issue in greater details in Section V.

The resulting architecture is shown in Figure 1. The RESTful EPCIS is a module which core is using the EPCIS WS-* standard interface. It basically translates the incoming RESTful request into WS-* requests and takes care of reformatting the results into several Web formats such as HTML and JSON. As shown on the left of the picture, the typical clients of the RESTful EPCIS are different from the business applications traditionally connected to the EPCIS. The browser is the most prevalent of these clients. It can either directly access the data by means of URL calls or indirectly using scripted Web pages.

In the following sections we detail the architecture of the RESTful EPCIS, focusing on the implementation of the REST patterns and constraints.

A. RESTifying the Query Interface

As mentioned before, in the EPCIS standard, most features are accessible through a WS-* interface. To specify the architecture of the RESTful EPCIS we systematically took these WS-* features and applied the properties of Resource Oriented Architectures.

1) Addressability and Connectedness: All the services of a Resource Oriented Architecture are modeled with resources which are components of an application worth being uniquely addressed and linked to. Each resource gets a unique and resolvable address in the form of a URL. Thus, the first step a ROA design is to **identify the resources** an EPCIS should be composed of and to make them **addressable**. Looking at the EPCIS standard, we can extract a dozen resources. We focus here on the four main types:

1) Locations (called "Business locations" in the EPCIS standard): those are locations where events can occur, e.g.,:"C Floor, Building B72".
2) Readers (called "ReadPoints" in the standard): which are RFID readers registered in the EPCIS. Just as Business

[3]http://autoidlabs.mit.edu/CS/content/OpenSource.aspx

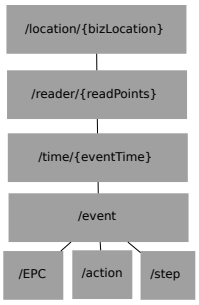

Fig. 2. Hierarchical representation of the browsable RESTful EPCIS resources.

Locations, readers are usually represented as URIs: e.g., `urn:br:maxhavelaar:natal:shipyear:incoming` but can also be represented using free-form strings, e.g.,: "Reader Store Checkout"

3) Events: which are observations of RFID tags, at a Business Location by a specific reader at a particular time.
4) EPCs: which are Electronic Product Codes identifying products (e.g., `urn:epc:id:sgtin:618018.820712.2001`), types of products (e.g., `urn:epc:id:sgtin:618018.820712.*`) or companies (e.g., `urn:epc:id:sgtin:618018.*`).

We first define a hierarchical clustering of resources based on the following URI template:
`location/{businessLocation}/reader/{readPoint}/time/{eventTime}/event`
More concretely, this means that the users begin by accessing the Location resources. Accessing the URL `http://.../location/` with the `GET` method retrieves a list of all Locations currently registered in the EPCIS. From there, clients can navigate to a particular Location where they will find a list of all Readers at this place. From the Readers clients get access to Time resources which root is listing all the Times at which Events occurred. By selecting a Time the client finally accesses a list of Events.

Each event contains information like its type, event time, Business Location, EPCs, etc. If a client is only interested about one specific field of an Event, he can get this information by adding the desired information name as sub-path of the Event URI. For example `EVENT_URI/epcs` lists only all the EPCs that were part of that Event. The resulting tree structure is shown in Figure 2, and a sample Event in Figure 3.

Furthermore, in a ROA all resources should be discoverable by browsing to facilitate the integration with the Web. Just as you can browse for Web pages, we should be able to find RFID tagged objects and their traces by browsing. Each representation of resources should contain links to relevant resources such as parents, descendants or simply related resources. This property of ROAs is known as "connectedness".

To ensure the connectedness of the RESTful EPCIS, each resource in the tree links to the resources below or to related resources. The links allow users to browse completely through the RESTful EPCIS where links act as the motor. Every available action is deduced by the set of links included. This way, people can directly explore the EPCIS from any Web browser, simply by clicking on hyperlinks and without requiring any prior knowledge of the EPCIS standard.

To ensure that the browsable EPCIS interface did not become too complicated, we limited the number of available resources and parameters. For more complex queries we provide a second, hierarchical, interface for which we map the EPCIS WS-* query interface to uniquely identifiable URIs. Each query parameter can be encoded and combined as a URI query parameter according to the following template
`/eventquery/result?param1=value1&...&paramN=valueN`
Query parameters restrict the deduced result set of matching RFID events. The RESTful EPCIS supports the building of such URIs with the help of an HTML form. If for example a product manager from Max Havelaar is interested in the events that were produced in Palmas, the following URL lists all events that occured at this business location:
`http://.../eventquery/result?location=urn:br:maxhavelaar:palmas:productionsite`
To further limit possibly very long search results, the query URI can be more specific. The manager might be interested only about what happened on that production site on the 4th of November 2009, which corresponds to the following URL:
`http:/../eventquery/result?location=urn:br:maxhavelaar:palmas:productionsite&time=2009-11-04T00:00:00.000Z,2009-11-04T23:59:59.000Z`
The HTML representation of this resource is illustrated in Figure 3.

To keep the full connectedness of the RESTful EPCIS, both the browsable and the query interface are interlinked. For example, the EPC `urn:epc:id:sgtin:0057000.123430.2025` included in the event of Figure 3, is also a link to the query which asks the EPCIS for all events that contain this EPC.

By implementing the addressability property we allow greater interaction with EPCIS data on the Web. As an example, since queries are now encapsulated in URLs, we can simply bookmark them, exchange them in emails and consume them from JavaScript applications. Furthermore, by implementing the connectedness property we enable users to discover the EPCIS content in a simple but yet powerful manner.

2) Uniform Interface: Finally, in a ROA, the resources and their services should be accessible using a standard interface defining the mechanisms of interaction. The Web implementation of REST uses HTTP for this purpose. We particularly focus on two aspects of the uniform interface here: the representation of resources, and the communication of errors.

a) Multiple Representation Formats: A resource is representation agnostic and hence should offer several representations (e.g., XML, HTML). HTTP provides a way for clients to

RESTful Path ID	ID	*Unique Path ID to represent the Event*		
Event Type	ObjectEvent			
Event Time	2009-11-04T10:21:39.000Z			
Time Zone Offset	+00:00			
Record Time	2010-02-26T15:04:59.000Z			
Business Location	urn:br:maxhavelaar:palmas:productionsite			
Read Point	urn:br:maxhavelaar:palmas:productionsite:outgoing			
Business Step	urn:epcglobal:epcis:bizstep:fmcg:shipping			
Action	OBSERVE			
EPC List				
EPC	urn:epc:id:sgtin:0057000.123430.2025	Company Prefix: 0057000	Item Reference: 123430	Serial Number: 2025
		Serialized Global Trade Item Number		
EPC	urn:epc:id:sgtin:0057000.123430.2026	Company Prefix: 0057000	Item Reference: 123430	Serial Number: 2026
		Serialized Global Trade Item Number		

Fig. 3. HTML representation of an EPC event as rendered by a Web browser. Every entry is also a link to the sub-resources.

retrieve the most adapted one. The RESTful EPCIS supports multiple output formats to represent a resource. Each resource first offers an HTML representation as shown in Figure 3 which is used by default for Web browser clients.

In addition to the HTML representation, each resource has also an XML and a JSON (JavaScript Object Notation) representation, which all contain the same information. The XML representation complies with the EPCIS standard and is intended to be used mainly for business integration. The JSON representation can be directly translated to JavaScript objects and is thus intended for mashups, mobile applications or embedded computers.

The choice of what representation to use is left to clients who can request it through the HTTP "content negotiation' mechanism[4]. Since content negotiation is built into the uniform interface, clients and servers have agreed-upon ways to exchange information about available resource representations, and the negotiation allows clients and servers to choose the representation that is the best fit for a given scenario.

b) Error Codes: The EPCIS standard defines a number of exceptions that can occur while interacting with an EPCIS. HTTP offers a standard and universal way of communicating errors to clients by means of "status codes". Thus, to enable clients, especially machines to make use of the exceptions defined by the EPCIS specification, the RESTful EPCIS maps the exceptions to HTTP status codes. An exhaustive list of error codes and their meanings for Resource Oriented Architectures can be found in [8].

B. Web-Enabling the Subscriptions

As mentioned before, standard EPCISs also offers an interface to subscribe to RFID events. Through a WS-* operation, clients can send a query along with an endpoint (i.e., a URL) and subscribe for updates. Every time the result of the query changes, an XML packet containing the new results is sent to the endpoint. While this mechanism is practical, it requires for clients to run a server with a tailored Web applications that listens to the endpoint and thus cannot be used by all users or cannot be directly integrated to a Web browser. To improve this, the RESTful EPCIS offers a RESTful subscription interface and a Web feed of the updates.

For this we use Atom. The Atom Syndication Format is an XML language specifying the syntax of Web feeds. Atom also specifies a RESTful protocol for publishing feed entries, the AtomPub protocol.

In the RESTful EPCIS, we propose an alternative interface for subscribing to RFID events using Atom as shown on the leftmost side of Figure 1. This way, end-users can formulate queries by browsing the RESTful EPCIS and get updates in the Atom format which most browsers can understand and directly subscribe to. As an example a product manager could create a feed in order to be automatically notified in his browser or any feed reader whenever one of his products is ready to be shipped from the warehouse. More concretely, this would result in sending an HTTP `PUT` request to `http://.../eventquery/subscription?reader=urn:ch:migros:stgallen:warehouse:expedition&epc=urn:epc:id:sgtin:0057000.123430.*`
Or, for a human client, clicking on the "subscribe' link present at the top of each HTML representation of query results. The product manager could then use the URI of the feed in order to send it to his most important customers for them to follow the goods progress as well. A simple but very useful interaction which would require a dedicated client to be developed and installed by each customer in the case of the WS-* based EPCIS.

C. Implementation

As shown in Figure 1, the core of the RESTful EPCIS is based on the Jersey[5] framework. Jersey is a software framework for building RESTful applications. It is especially interesting since it complies with the JAX-RS[6] (JSR 311) standard for building RESTful Web services.

Jersey is responsible for managing the resources' representations and dispatching HTTP requests to the right resource depending on the request URL. When correctly dispatched to the RESTful EPCIS Core, every request on the querying or browsing interface is translated to a WS-* request on the EPCIS. This makes the RESTful EPCIS entriely decoupled from any particular implementation of an EPCIS. However, for our tests we used the Fosstrak EPCIS.

For the subscription interface we used Apache Abdera, which is an open-source implementation of an Atom-Pub server. Thus, every time a client subscribes to a query, the RESTful EPCIS checks whether this feed already exists by checking the query parameters, in any order. If it is not the case it creates a query on the WS-* EPCIS and specifies the address of the newly created feed. As a consequence every update of the query is directly `POST`ed to the feed resource which creates a new entry using Abdera and stores it in an embedded SQLite[7] database.

[4]http://tinyurl.com/y89n7uk

[5]https://jersey.dev.java.net
[6]http://jsr311.dev.java.net
[7]http://www.sqlite.org

Jersey, Abdera and SQLite are packaged with the RESTful EPCIS core in a WAR (Web Application Archive) that can be deployed in any Java compliant Web or Application Server. We tested it successfully on both Glassfish[8] and Apache Tomcat[9].

IV. The EPC Dashboard Mashup

To better illustrate the new type of applications the RESTful EPCIS unveils we created the EPC Dashboard Mashup, an innovative Web mashup, that helps product, supply chain and store managers to have a live overview of their business at a glance. It can further help consumers to better understand where the goods are coming from and what other people think about them. The EPC Dashboard is based on the concept of widgets in which the event data are visualized in a relational, spacial or temporal manner.

The EPC Dashboard consumes data from the RESTful EPCIS. Usually these data are hard to interpret and integrate. The dashboard makes it simple to browse and visualize the EPC data. Furthermore, it integrates the data with multiple sources on the Web such as Google Maps, Wikipedia, Twitter, etc. To better understand the use of such a tool, let us first introduce two use-cases before looking at the applications' architecture.

A. Use Cases

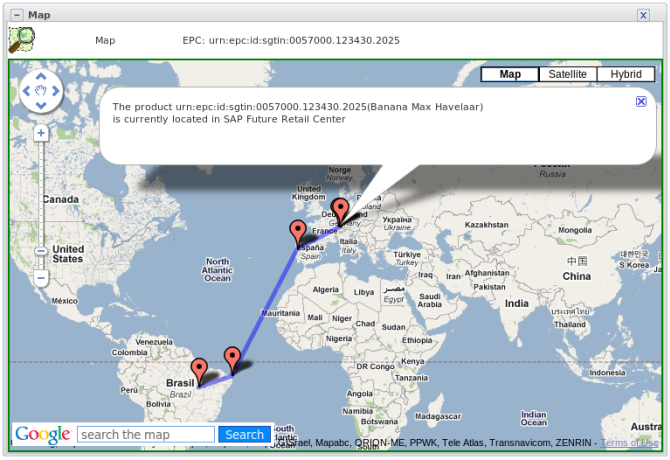

Fig. 4. The Maps widget is following the route of the banana tagged with the EPC urn:epc:id:sgtin:0057000.123430.2025.

Rachel, a customer, just bought Max Havelaar Bananas and Lindt Chocolate from a retail store "M" in Switzerland. She wants to know more about the Bananas. For that purpose, she opens the EPC Dashboard in her preferred browser. She activates the Product Description Widget and enters the EPC of the article. The EPC Dashboard now shows her a description of bananas and Max Havelaar extracted from Wikipedia. Likewise, the Product Video Widget provides her with video about planting of these bananas. She is further interested to know about where this particular banana has grown. Rachel activates the Map Widget and she can see on the map of Figure 4 where her banana originates from. In addition she also sees the route that the banana has taken from its origin to the M retail shop. She further prepares a Banana Split with the chocolate she just bought and shares the recipe on Twitter through the Product Buzz Widget.

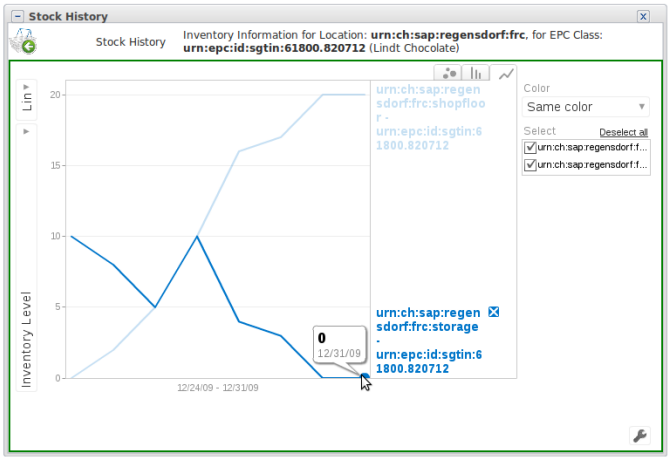

Fig. 5. The Stock History widget allows for looking at the flows of goods through the supply chain. Here the manager can see that all the available Lindt chocolate has been transfered to the shop, leaving an empty stock.

In the M store in Zurich, Andy, the product manager of chocolate products wants to check the recent inventory levels of the Lindt Chocolate. He also browses to the EPC Dashboard Mashup and opens the Stock History Widget with the corresponding inventory RFID reader. According to Figure 5, he discovers that there had been an increase demand for Lindt Chocolate and that the chocolate has almost entirely been transfered from the stock to the shop. He directly orders larger shipping contingents. He also subscribes to the feed listing the arrival of Lindt products in the stock using the entry gate reader. This way, a feed reader on his mobile phone and in his favorite browser will inform him when the ordered products have arrived. He is further interested in knowing why the Lindt products are so popular recently. Andy activates the Product Buzz Widget and sees the current Twitter messages related to Lindt Chocolates as shown in Figure 6, including Rachel's recipe. He can use this information for marketing analysis.

B. Mashup Architecture

The EPC Dashboard integrates several information sources. This information is encapsulated in small windows called widgets. The widgets combine services on the Web with traces coming from the RESTful EPCIS. The EPC Dashboard Mashup currently offers 12 widgets using different APIs and services. As an example, the Map Widget is built using the Google Maps Web API (see Figure 4), the Product Buzz Widget uses the Twitter RESTful API (Figure 6) and the Stock History Widget uses the Google Visualization API (Figure 5).

All widgets are connected to each other which means that actions on a given one can propagate the selection to the other

[8]http://glassfish.org
[9]http://tomcat.apache.org

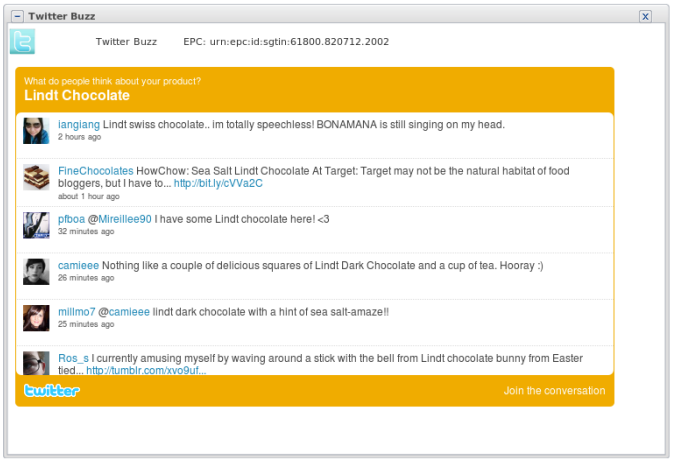

Fig. 6. The Product Buzz Widget extracts live opinions and information about particular products (here Lindt Chocolate) from Twitter.

widgets and changes their view accordingly. As such, widgets listen to selections and can make selections. This interaction is implemented using the observer pattern [12] where consumers (i.e., the widgets) register to asynchronous updates of the currently selected Locations, Readers, Time or EPCs. This architecture allows the creation and integration of other Web widgets with very little effort. The EPC Dashboard itself is a JavaScript application built using the Google Web Toolkit[10], a framework to develop rich Web clients. This has been possible because having a RESTful Interface upon the EPCIS which eases the development of mashups.

V. Evaluation

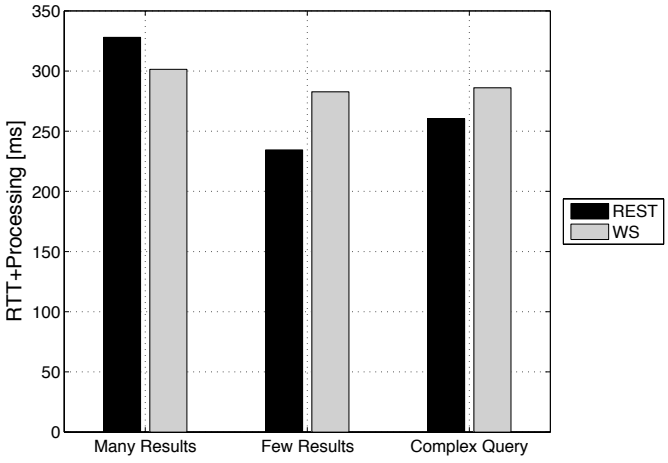

Fig. 7. Average RTT and processing time when using the WS-* interface and the REST interface for three types of requests each run 100 times. Standard deviations are as follow: 49, 77, 39, 11, 38, 12 ms.

As mentioned before, the RESTful EPCIS is an add-on to the standard EPCIS where each REST request is eventually translated to a (local) WS-* request. This results in an overhead that we evaluate here.

[10]http://code.google.com/intl/en/webtoolkit

The experimental setup is composed of a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of ram. We deploy Fosstrak and the RESTful EPCIs on the same instance of Apache Tomcat with a heapsize of 512 MB. We evaluate three types of queries all returning the standard EPCIS XML representation.

The first query (Q1, "Many Results" in Figure 7) requests all events recorded by the EPC, i.e., a small request returning a document of 30 KB with 22 events each composed of about 10 EPCs. In the second test (Q2, "Few Results"), is a query returning a document of 2.2 KB with only two results. The last test (Q3, "Complex Query") is a query containing a lot of parameters and returning 10 events. We test each of these queries asking for the standard XML representation. All queries are repeated 100 times from a client located on a machine one hop away from the server with a Gigabit ethernet connectivity. The client application is programmed in Java and uses a standard JAX-WS client for the WS-* calls and the standard Apache HTTP Client and DOM (Document Object Model) library for the REST calls.

As shown on Figure 7, for Q1 the RESTful EPCIS has an average overhead of 30 ms due to the computational power required to translate the requests from REST to WS-* and vice-versa. For Q2 and Q3 the REST requests are executed slightly faster (about 20 ms) than the WS-*. This is explained by three factors. First, since there are fewer results, the local WS-* request from the RESTful EPCIS is executed faster. Then, REST packets are slightly smaller as there is no SOAP envelope [4]. Finally, unmarshalling WS-* packets (using JAXB) on the client-side takes significantly longer than for REST packets with DOM. For Q3, similar results are observed. Overall, we can observe that the RESTful EPCIS creates a limited overhead of about 10% which is compensated in most cases by the relatively longer processing times of WS-* replies. This becomes a particularly important point when considering devices with limited capabilities such as mobile phones or sensor nodes as well as for client-side (e.g., JavaScript) web applications.

It is worth mentioning that the WS-* protocol can be optimized in several ways to better perform, for example by compressing the SOAP packets and optimizing JAXB. However as the content of HTTP packets can also be compressed this is unlikely to drastically change the results. Furthermore, because they encapsulate requests in HTTP `POST`, WS-* services cannot be cached on the Web using standard mechanisms. For the RESTful EPCIS however, all the queries are formulated as HTTP `GET` requests and fully contained in the request URL. This allows to directly leverage from standard Web caching mechanisms [9] which would importantly reduce the response times [4]. However a discussion on caching strategies is outside the scope of this article and left for future work.

VI. Conclusion and Future Work

In this paper we propose to extend the existing EPCIS architecture with a RESTful module. This literally bring RFID traces to the Web, every tagged product, reader, location,

etc. become fully addressable resources. Using the HTTP protocol tagged objects can be directly searched for, indexed, bookmarked, exchanged and feeds can be created by end-users. Furthermore, this enables exploring the EPCIS data simply by browsing them, which helps making sense of the data. We argue that this adds more flexibility to the types of applications that can be built on top of an EPCIS and opens the EPCIS API for fast-prototyping to the very large and active community of Web and mobile developers. We illustrate this by means of a JavaScript Mashup: the EPC Dashboard which is an easily extensible business intelligence interface for managers that reuses a number of Web APIs.

We further show that this added flexibility does not necessarily have to hinder the overall performances, deploying the RESTful EPCIS on the same machine as the WS-* EPCIS leads to satisfactory results while preserving the EPCIS-vendor independence.

However, there are still a number of challenges to tackle towards a global network of Web-enabled EPCIS. First of all, although global sharing is the vision of the EPC Network, very few real-world companies are willing to share their data beyond their own boundaries [13]. Thus, for a realistic global deployment there is a need for a more comprehensive and flexible sharing framework that allows a thinner-grained selection of the partners the data are shared with. Fortunately enough, this topic has been identified as an important issue and is currently being actively researched on and a standard at least partially addressing this issue is currently written by EPC Global[11]. This new standard, currently called "Discovery Services" is also expected to solve another important problem to enable global sharing: the need for a discovery service that allows looking for product traces accross several instances of EPCISs.

Since open-APIs for smart things such as the RESTful EPCIS emphasize the need for trust, access control and fine-grained sharing mechanisms, several Web of Things related projects move forward tackling these goals and the RESTful EPCIS could benefit from this research. In [14] the authors propose an architecture for sharing and controlling access to REST-enabled smart things based on Web-authentication through social networks. Such an architecture could also be applied to the RESTful EPCIS. This would enable companies to share their data through Web APIs only to their trusted networks of partners and customers. Nevertheless, while the RESTful EPCIS gains value as the EPC network grows, the presented solution can also be used on a local basis, connected to a limited number of EPCISs, for instance within a single company.

Contacts with the Fosstrak development team have led to positive feedback and we will release the RESTful EPCIS software and specification as an open-source module of the Fosstrak project, under the name of epcis-restadapter[12]. This will provide us with very valuable information on how to improve the software and a better overview of the performances and stability in the long run. It is also a good chance to create greater community of developers using the EPCIS and its RESTful API for their future Internet/Web of Things or Mobile prototypes and products.

[11] http://www.epcglobalinc.org/standards/discovery

[12] http://www.webofthings.com/rfid

ACKNOWLEDGMENT

The authors would like to thank Ivan Delchev, Vlad Trifa and Patrik Fuhrer for their contributions to the architecture of the RESTful EPCIS. Thanks also to Christof Roduner and Christian Floerkemeier for their very valuable expertise in the EPC Network standards and implementations.

REFERENCES

[1] C. Floerkemeier, M. Lampe, and C. Roduner, "Facilitating RFID development with the accada prototyping platform," in *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE Computer Society, 2007, pp. 495–500.

[2] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a Multi-Faceted metric for service design," in *Proc. of the 18th International World Wide Web Conference (WWW'09)*, Madrid, Spain, Apr. 2009.

[3] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. big web services: making the right architectural decision," in *Proc. of the 17th international conference on World Wide Web (WWW)*. New York, NY, USA: ACM, 2008, pp. 805–814.

[4] D. Yazar and A. Dunkels, "Efficient application integration in IP-based sensor networks," in *Proc. ACM of the First ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings (BuildSys)*, Berkeley, CA, USA, Nov. 2009.

[5] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST - a protocol for integrating sensor networks into the internet," in *Proc. of the Workshop on Real-World Wireless Sensor Network (SICS)*, Stockholm, Sweden, 2005.

[6] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic, "People, places, things: web presence for the real world," *Mob. Netw. Appl.*, vol. 7, no. 5, pp. 365–376, 2002.

[7] D. Guinard, F. von Reischach, and F. Michahelles, "MobileIoT toolkit: Connecting the EPC network to MobilePhones," in *Proc. of Mobile Interaction with the Real World at Mobile HCI (MIRW)*. Amsterdam, Netherlands: The University of Oldenburg, Sep. 2008.

[8] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, Inc., May 2007.

[9] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 115–150, 2002.

[10] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin, "pREST: a REST-based protocol for pervasive systems," in *Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, 2004, pp. 340–348.

[11] D. Guinard, V. Trifa, and 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, Nov. 2010.

[12] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Nov. 1994.

[13] P. Schmitt, "Adoption und diffusion neuer technologien am beispiel der Radiofrequenz-Identifikation (RFID)," Ph.D. dissertation, ETH Zurich. [Online]. Available: http://tinyurl.com/3ac5bfw

[14] D. Guinard, M. Fischer, and V. Trifa, "Sharing using social networks in a composable web of things," in *Proc. of the 1st IEEE International Workshop on the Web of Things (WoT 2010) at IEEE PerCom 2010*, Mannheim, Germany, 2010.