WebPlug: A Framework for the Web of Things

Benedikt Ostermaier*, Fabian Schlup*, Kay Römer*†
*Institute for Pervasive Computing, ETH Zurich, Zurich, Switzerland
Email: ostermaier@inf.ethz.ch, fschlup@alumni.ethz.ch, roemer@inf.ethz.ch
†Institute for Computer Engineering, University of Luebeck, Luebeck, Germany
Email: roemer@iti.uni-luebeck.de

***Abstract*—We present WebPlug, a framework for the emerging Web of Things. It consists of several building blocks which ease integration of things, including their sensors and actuators with the Web. For example, WebPlug supports versioning of and eventing for arbitrary Web resources, like sensor readings, thus simplifying the process of creating physical mash-ups. After providing an analysis that led to the development of WebPlug, we present selected aspects of its design. We complete the paper by presenting the application of our framework in a real-world scenario.**

I. Introduction

In the emerging *Web of Things*, the Web is extended from its original document-centric design to an application layer for the real world. In this yet-to-be-standardized concept, real-world objects like consumer devices are integrated into the WWW by representing them as Web resources, which can be accessed using lightweight APIs based on the REST principle [1]. Exposing real-world objects – including their attached sensors and actuators – as URLs enables novel application scenarios, like the search for real-world entities by their *current state* [2].

In the Web of Things, *sensors* and *actuators* play a central role, as they constitute the physical interface between the digital and the real world: They enable us to capture and change aspects of the real world, often in real time. The idea of providing end-users the ability to compose personal services based on physical objects, so-called *physical mash-ups*, is addressed in this paper. To this end, we developed a prototypical framework for the Web of Things which does not rely on a centralized infrastructure but rather introduces several building blocks. These components could be distributed among connected devices but could also be located in the *cloud*. The emphasis is on integrating sensors and actuators with the Web of Things. Due to space constraints, we can only cover selected aspects of our framework. Therefore, although considered important and also implemented in part, we have to omit aspects like self-description, discovery and security. For more information we refer to [3].

We start by providing a short analysis of the requirements of such a framework in the next section. We then introduce selected aspects of our framework in Sect. III and present the application of our framework in a real-world scenario in Sect. IV. The paper is completed with a brief survey on related work in Sect. V and an outlook in Sect. VI.

II. Analysis

The Web of Things should not be restricted to domain experts of specific application areas, but rather be open to the average user. We believe that looking at past or current sensor readings (e.g., *"Did I lock the garage door?"*) and creating *personal mash-ups* which include physical objects (e.g., *"I want to be alerted as soon as one of my plants is in a critical condition."*) will be a commonplace operation. By gently extending the Web to things, users could benefit from their already acquired knowledge and things could be integrated seamlessly with the existing Web infrastructure, like cloud services.

As one might often be interested not only in the current but also in past states of a sensor (e.g., *"When was the last time someone accessed my drawer?"*), we argue that there is a need to archive and make available past sensor readings. Additionally, filtering or querying past sensor states is required in order to be able to automate certain processes. The effort required to automate a process has to be minimized by providing a homogeneous resource-based Web framework instead of requiring the user to learn yet another Web API for the same purpose (e.g., retrieving past sensor readings) or even switching to another concept (e.g., SOAP).

Instead of creating complex web services for each specialized requirement, more complex mash-ups should be composed of simple building blocks, a concept which has proven its benefits with Unix pipes, for example. To this end, we believe that modelling everything as a resource is a good strategy for modularization, as it eases access and fosters the creation of mash-ups. URLs play an important role in a future Web of Things – we see them as vectors through information space, containing semantic hints for the user.

III. Design

We outline the most important concepts and design decisions in WebPlug that were guided by the goal to enable easy and safe composition of simple components (developed by multiple, independent parties) into complex applications.

978-1-4244-5328-3/10/$26.00 ©2010 IEEE

A. Basic Operations

In our system, we limit the use of HTTP verbs to PUT, GET, POST and DELETE, which map to creating, reading, updating, and deleting data (CRUD). Additionally, the HEAD operation is supported as a reduced version of GET, where only the headers of the corresponding GET response are returned. PUT creates a new resource which is specified by the supplied request URI and stores the enclosed data, whereas POST updates an existing resource. If a resource is versioned, each POST operation creates a new version of the resource. We do not introduce additional HTTP verbs as CalDAV [4] does, for example, in order to keep the system open for all clients.

B. Typed Resources

In a modular system as WebPlug, explicit typing of components helps to ensure meaningful compositions. Therefore, each resource in our framework has an associated resource type. It is not just descriptive, as in DCMI [5], but also specifies semantics and provides support for ontologies. The resource type specifies the behavior of the resource, its interface and may also provide a description of the real-world concept, if applicable. Resource types are identified by URLs and included in the headers of an HTTP request or response, specified by a header called `Resource-type`. The resource type is orthogonal to the well-known content type, which specifies only the type of representation of the resource. The concept of typed resources has a broader scope than HTTP's OPTIONS method and is more flexible: For example, one could imagine a service which would consider the `Resource-type` header in a PUT request and create a resource of the specified type at the given location. We currently support objects, collections, atomic values, text, quantities, references (URL pointer), and binary data. Resource types are also utilized during discovery, where the querying entity can automatically retrieve the type specifications of discovered resources. Due to space constraints, we omit the definition format of resource types.

C. Collections

Dealing with collections of values (such as sensor readings) is a common task in WebPlug, which therefore offers a collection resource that represents a set of items and can contain resources of any type.

Collections may be represented in a variety of formats, including Atom feeds, JSON arrays, HTML and CSV. In addition, special-purpose representations are possible, depending on the nature of the contained elements. For example, collections of numeric values can be represented graphically as a diagram depicting the values on a time axis according to the time they have been created or inserted into the collection, which is very helpful for sensor readings.

Each member of a collection is identified by a unique string that can be used to build its individual URL by appending the string as an additional segment to the collection's URL. Resources are added to the collection by POSTing them to the collection URL, which will return the newly created URL of the item in the `Location` header of the HTTP response. Items of a collection can be fetched, updated and deleted using GET, POST and DELETE with the item's URL. This is similar to the access patterns for collections proposed as part of the Atom Publishing Protocol (APP) [6]. GET and DELETE can also be used on the collection URL.

For example, when adding the URL *http://stuff/officekeys* to the collection *http://drawer/contents*, it could return *http://drawer/contents/23* as the URL for the item in the collection.

There is also an index-based view on the items of a collection, based on the order of their creation starting with index 1. The special indices *first* and *last* are also supported, each offering the possibility to be used with an optional offset in the form $first + n$ and $last - n$ respectively. In order to distinguish these special URLs from others, a prefix i: is used, followed by the index. Finally, the special property *count* can be accessed by adding the name as a segment to the collection's URL, yielding the number of items in the collection.

Assuming we want to identify the nineteenth and the penultimate item that have been added to a fridge, and the total number of items, we would write:

http://my.fridge.net/contents/i:19
http://my.fridge.net/contents/i:last-1
http://my.fridge.net/contents/count

D. Meta-URLs

Resources are often intimately connected with each other. For example, the history of a resource is intrinsically tied to the original resource. To this end, we introduce the *Meta-URL* concept to access such meta resources of a given resource by appending a well-defined term to the URL of the base resource, separated by the delimiter `@`, which acts as escape character.

For example, by appending the term `@history` to a given URL, we will address the list of past versions of the resource denoted by the base URL. In our use case, *http://acme/lamp/power@history* will provide the history of *http://acme/lamp/power*, i.e., the usage history of the lamp.

While this conflicts with the *opacity axiom* of URIs, which states that one should not interpret the internal contents of the URI, it adds advantages over the usual approach (which would be linking from the base resource). First, users can instantly deduce the Meta-URL from a given base URL in the browser. Second, and more importantly, Meta-URLs may also be stacked: *http://acme/lamp@accesscontrol@history* might provide a versioned list of past access control settings for *http://acme/lamp*.

E. History

In sensor-based applications, keeping track of a history of past sensor readings is a common task. Web feeds have been proposed [7] to be used to represent streams of sensor data. In our framework, this approach is taken one step further by introducing a generic concept of history keeping for any type of resource, whether it is a sensor measurement, a picture, or other document. It is based on the idea of each resource making available a list of its previous values in form of a separate resource, addressed by the Meta-URL composed of the base resource's identifier and the suffix `@history`. The history is a read-only collection type resource and is represented and accessed like any other collection in the system. For example, to access the penultimate reading of a temperature sensor, one could access *http://my.thermometer.net/temperature@history/i:last-1*.

F. Observing Resources

Figure 1. Example of how an observer is added to a resource (top) and how that resource notifies its observers when its value changes (bottom).

A key mechanism to enable composition of components is to inform a component about state changes (i.e., events) in another component. Similar to the concept of webhooks [8], our eventing mechanism is based on URL callbacks, where a URL registered with a specific *event* is *notified* as soon as this event occurs.

In our approach, an *event* is always the change of data identified by a web resource. *Observable resources* are resources which support subscriptions to changes in their associated data. *Observers* are URL callbacks which are subscribed to one ore more *observable resources*. Note that any resource which supports the POST operation may act as an observer and that multiple observers may be subscribed to a single observable resource. As soon as the data of an observable resources changes, all registered observers are *notified* by sending a POST request to the respective URLs. For each observable resource, there is an associated resource which represents the list of registered observers. The URL of this collection is a Meta-URL which can be deduced by appending the term *@observers* to the URL of the observable resource.

For example, consider an observable resource which reports the state of a door, *http://myhome.org/frontdoor/status*. As soon as the door is opened, the data associated with this resource changes from "closed" to "open" and all associated observers are notified. The list of observers for this sensor can be accessed at *http://myhome.org/frontdoor/status@observers*. As the list of observers is a collection resource, the interaction is well-defined: an observer is added by submitting its URL with a POST request to the list, and removed again by sending a DELETE to the URL of the entry (see Fig. 1 for an example of a subscription and event call). One can also retrieve the list of all observers using a GET request on the list.

Notifications come in two different forms, depending on the type of the observed resource. Simple types (including text and quantities) and references are passed *by value*, sending the new value directly to the observer as payload of an HTTP POST message. Objects, collections, and binaries, where the actual value is potentially large, are passed *by reference* by sending a URL pointer. The URL used for this purpose is pointing to the corresponding record in the history (e.g. *http://example.net/p1@history/37845* for notifications of a change in *http://example.net/p1*) because this is a stable address identifying the exact version of the value.

When sending out notifications to observers, the URL of the resource triggering the event is included in the message in form of the standard *Referer* header. This gives observers the possibility to determine where an update originates in the case they are subscribed to multiple resources.

Note that by introducing intermediary hubs (as in [9], for example), which relay a single notification to multiple observers, we expect the concept of URL callbacks to scale to large numbers of observers and notifications.

G. Expressions

Connecting components typically requires some glue logic. Expressions provide a simple form of such glue logic by enabling the evaluation of an arithmetic expression in the context of an object. The URL of such an expression is constructed appending the expression to the object URL. For example, the resource *http://my.fridge.net/temperature>10* will return true if *http://my.fridge.net/temperature* is currently above 10°C, false otherwise (the ">" sign needs to be escaped, which is done automatically by most browsers). Note that observers can also be registered with expressions. Due to space constraints, we omit further details on expressions.

H. Poller

In order to be able to provide push-functionality for non-observable resources, i.e., resources outside of our framework, we introduce a component called *poller*. A poller is a virtual entity that fetches a specified resource in the Web periodically and publishes its value as an own, observable property. Every time the resource is retrieved, its value is compared to the previous one and the registered observers

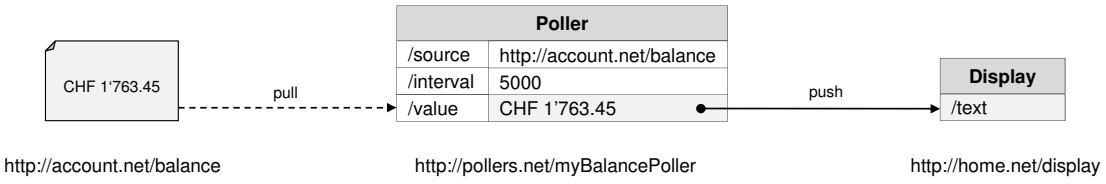

Figure 2. Illustration of a poller.

are notified if it has changed in the meantime. Pollers are designed as virtual objects with three properties corresponding to the *source* to be fetched, the *interval* in which to do it and the last *value* retrieved. Users interested in subscribing to a given Web resource which does not support the registration of observers, can use a poller to achieve their goal: they set the *source* property to the URL of the resource of interest, specify the desired polling interval in the *interval* property and add their observer to the *value* property (Fig. 2).

Pollers are active components that produce new events without receiving any from another component. In this sense, they are much like sensors and – in fact – can be thought of as virtual sensors that are "sensing" a Web resource in a given interval and make the result available as their output.

I. Evaluator

Raw sensor data typically needs to be processed to obtain a derived value which is then used as the input to another component. Examples include checking measurements against a threshold or the detection of motion in a series of images. Furthermore, there must be a way of combining multiple inputs from different sources into one aggregated result. The generic solution presented here is based on a virtual component called *evaluator*, which takes one or multiple inputs and produces a single output by evaluating an external function. It is designed as a virtual object with at least three properties: one or more inputs, one function reference, called *function*, and one evaluation result, called *value*. As soon as one of the input properties is updated with a new value, all the inputs are taken to reevaluate the specified function and the value of the result property is updated accordingly. As illustrated in Fig. 3, the input properties are registered as observers to the web data sources, the function is specified by a URL, and the target resource subscribes to the output of the evaluator.

Functions supported by the evaluator need to be stateless, RESTful services producing a result from a set of arguments without side effects. A function is evaluated by constructing a URL including the input parameters and retrieving a representation of this resource by an HTTP GET request.

An evaluator's function property is used to specify the template for the construction of the function URL. For this, the format known as URI templates [10] is used, where placeholders wrapped in curly brackets are included in a URL string. Evaluators offer the possibility to use any of its properties as a variable in the template by including a placeholder with its local path name. The following example illustrates a URL template for a service checking whether the temperature input property exceeds a given threshold. Every time the value of the temperature property changes, the evaluator would replace the placeholder with the new value and GET the constructed URL.

http://services/isOver?value={temperature}&threshold=23

Due to the possibility to use any property as a variable in the template, it is even possible to include the previous result, or a historic value of any property. The following example shows how one could check whether the last two versions of an input value named *temperature* are equal.

http://services.net/isEqual?a={temperature@history/last}
&b={temperature@history/last-1}

IV. Implementation

We implemented a prototype of our framework in Java, using the RESTlet engine. Additionally, we implemented a subset of the functionality on a Nokia N95 mobile phone, using Python for S60 (PyS60) and the Nokia Mobile Web Server.

Our framework supports *resource factories*, where an entire pool of virtual components like pollers and evaluators can be hosted. Such a pool behaves like a collection containing only virtual components of a given type. For example, in order to create a new poller, one POSTs its desired configuration to *http://wot/pollers* and the new poller might be created at *http://wot/pollers/534*, which is returned to the user.

We will now demonstrate the usage of our framework for a real-world example consisting of several components.

A. Application Example

Our objective is to create a simple motion detection system, using a standard webcam with our framework. To notify the user of any activity detected by the camera, his or her cell phone should signal it with a short vibration. It is obvious that in order to achieve the outlined functionality, we cannot connect the camera directly to the mobile but rather need some intermediary components.

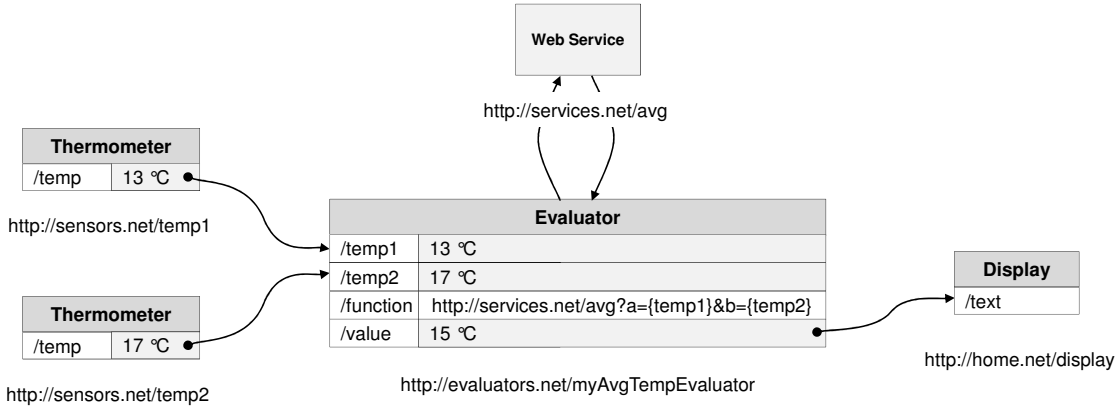

Figure 3. Illustration of an evaluator.

1) Components: The scenario consists of the following components:

- A standard webcam, which publishes its current image at a known URL.
- A poller, which creates an image sensor based on the images of the webcam, adding observability and history for its readings.
- An evaluator, which calls a RESTful web service with two subsequent images of the webcam as soon as a new image is present and publishes the result.
- A RESTful web service, which compares the similarity of two images which are given as parameters using URL-pointers. It returns a number between 0 (identical images) and 1 (completely different images), which serves as an activity metric.
- A RESTful mobile phone, which allows access to parts of its functionality using RESTful services.

2) Setup: Let us assume the webcam publishes its current image at *http://webcam.org/image.jpg*. At first, we have to create an instance of a poller and set its properties accordingly. Assuming the poller is located at *http://wot/pollers/001*, we set *http://wot/pollers/001/source* to *http://webcam.org/image.jpg*, and *http://wot/pollers/001/interval* to 500, using POST operations. The poller will then download the webcam image every 0.5 seconds and publish the latest image at *http://wot/pollers/001/value*. Additionally, observability and versioning are provided for the *value* property.

Next, we have to set up an evaluator for use with the image comparison service. Let us assume that the comparison service is called with *http://services.net/imgcmp?a=<image1>&b=<image2>*, where a and b are assigned the URLs of the images to compare. Let us further assume that the evaluator is created at *http://wot/evaluators/001*. We need to set its *function* property to a URI template which calls the image comparison service using the URLs of the two latest images of the webcam. This is done by setting *http://wot/evaluators/001/function* to

http://services.net/imgcmp?a={picture}
&b={picture@history/i:last-1}.

The definition of the function template automatically creates an input property for the evaluator called *picture*, which we connect to the output of the poller by registering an observer: *http://wot/evaluators/001/picture* is added to *http://wot/pollers/001/value@observers*. As soon as the property *picture* of the evaluator is updated, the component expands the URI template of the function and performs a GET operation on the resulting resource.

Recall that when observing complex resource types like images, notifications do not include serialized data but rather a permanent URL to the corresponding data. Therefore, as soon as the poller is publishing a new image, the evaluator's input property *picture* is updated with the URL of the latest image. When this happens, the evaluator will then evaluate the template, by replacing the template variables with the current values. In our example, this might lead to a call like

http://services.net/imgcmp
?a=http://wot/pollers/001/value@history/7364
&b=http://wot/pollers/001/value@history/32734,

for example (URL encoding omitted for readability). The evaluator's *value* property is then updated with the current activity measure of the webcam. In order to let the mobile phone vibrate as long as an activity is detected, we register an observer for the phone's vibration actuator by adding *http://myphone/vibrating* to the observer list at *http://wot/evaluators/001/value>0.1@observers*. This way, two events may be generated: one when the current output value of the evaluator exceeds 0.1 – in this case, "true", the result of the expression, is sent to the URL of the phone's

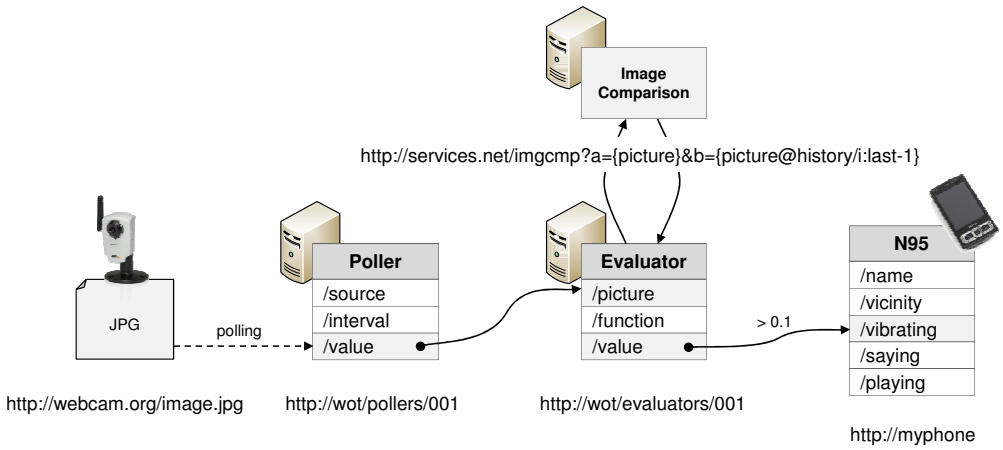

Figure 4. A physical mash-up which realizes a motion detection system based on a webcam.

vibration alert. As soon as the activity drops below the threshold, the state of the expression changes and therefore its result ("false") is POSTed to the phone's vibration actuator, causing it to stop.

The complete setup is depicted in Fig. 4. Note that it may be configured using the built-in web front-end of our framework as well as its RESTful API. As we can see from this example, the flexible use of *observers* allows a piped, distributed execution of the given task "in the cloud" – there is no central program which specifies this task, instead both program flow and state are distributed among the participating components.

V. Related Work

We are aware that many of the concepts considered in this paper have already been addressed by the Web community. However, we believe that our uniform URL-based approach provides significant benefits in the context of creating personal mash-ups in the Web of Things. For example, manipulations of collections are addressed by WebDAV [11] and the Atom Publishing Protocol [6]. Versioning is addressed by a WebDAV extension [12] and an attempt based on the Atom format [13]. However, we do not depend on an extension of HTTP or a single serialization format such as Atom in order to achieve this functionality. URL callbacks have been promoted by [8] and [9], for example. Our concept extends these approaches by enabling subscriptions on arbitrary resources, including those containing expressions.

VI. Conclusion and Outlook

We presented WebPlug, a framework for the Web of Things, which strives to provide unified concepts for interacting with real-world objects. WebPlug is currently work in progress, as we plan to implement and test additional concepts. Security is an important issue – the traditional username/password authentication scheme does seem adequate for a Web of Things, where devices interact among each another. Likewise, a discovery mechanism for elements in the Web of Things is required.

References

[1] E. Wilde, "Putting Things to REST," UC Berkeley, Tech. Rep. 2007-015, November 2007.

[2] B. M. Elahi, K. Römer, B. Ostermaier, M. Fahrmair, and W. Kellerer, "Sensor Ranking: A Primitive for Efficient Content-Based Sensor Search," in *IPSN '09*, San Francisco, CA, USA, 2009.

[3] F. Schlup, "Design and Implementation of a Framework for the Web of Things," Master's thesis, ETH Zurich, September 2009.

[4] C. Daboo, B. Desruisseaux, and L. Dusseaul, "Calendaring Extensions to WebDAV (CalDAV)," Internet RFC, March 2007. [Online]. Available: http://tools.ietf.org/html/rfc4791

[5] "DCMI Metadata Terms," Homepage, January 2008. [Online]. Available: http://dublincore.org/documents/dcmi-terms/

[6] J. Gregorio and B. de hOra, "The Atom Publishing Protocol," Internet RFC, October 2007. [Online]. Available: http://tools.ietf.org/html/rfc5023

[7] R. F. Dickerson, J. Lu, J. Lu, and K. Whitehouse, "Stream Feeds - An Abstraction for the World Wide Sensor Web," in *IOT*, ser. Lecture Notes in Computer Science, 2008.

[8] J. Lindsay, "Webhooks," Homepage. [Online]. Available: http://www.webhooks.org/

[9] B. Fitzpatrick, B. Slatkin, and M. Atkins, "PubSubHubbub Core 0.2," Homepage, 2009. [Online]. Available: http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.2.html

[10] D. Orchard, "URI template," Internet-Draft, September 2008. [Online]. Available: http://tools.ietf.org/html/draft-gregorio-uritemplate-03

[11] C. Daboo, "Extended MKCOL for Web Distributed Authoring and Versioning (WebDAV)," Internet RFC, September 2009. [Online]. Available: http://tools.ietf.org/html/rfc5689

[12] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead, "Versioning Extensions to WebDAV," Internet RFC, March 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3253.txt

[13] A. Brown, G. Clemm, and J. R. (Ed.), "Link Relations for Simple Version Navigation," Internet-Draft, November 2009. [Online]. Available: http://tools.ietf.org/id/draft-brown-versioning-link-relations-03.txt