Proceedings of the First International Workshop on RESTful Design

WS-REST 2010

Editors: Cesare Pautasso, Erik Wilde, and Alexandros Marinos

April 26, 2010

Raleigh, North Carolina

Table of Contents

- Session 1: *Introduction*
  - First International Workshop on RESTful Design (WS-REST 2010): **1**
    Cesare Pautasso, Erik Wilde, Alexandros Marinos
- Sessions 2 and 3: *REST Infrastructure*
  - Developers Like Hypermedia, But They Don't Like Web Browsers: **4**
    Leonard Richardson
  - Exploring Hypermedia Support in Jersey: **10**
    Marc Hadley, Santiago Pericas-Geertsen, Paul Sandoz
  - The Role of Hypermedia in Distributed System Development: **16**
    Savas Parastatidis, Jim Webber, Guilherme Silveira, Ian Robinson
  - Using HTTP Link: Header for Gateway Cache Invalidation: **23**
    Mike Kelly, Michael Hausenblas
  - Replacing Legacy Web Services with RESTful Services: **27**
    Charles Engelke, Craig Fitzgerald
- Session 4: *REST Research*
  - Towards a Practical Model to Facilitate Reasoning about REST Extensions and Reuse: **31**
    Federico Fernandez, Jaime Navon
  - A Formal Definition of RESTful Semantic Web Services: **39**
    Antonio Garrote Hernández, María N. Moreno García
  - A RESTful Messaging System for Asynchronous Distributed Processing: **46**
    Ian Jacobi, Alexey Radul
- Session 5: *Practical REST*
  - Developing a RESTful Mixed Reality Web Service Platform: **54**
    Petri Selonen, Petros Belimpasakis, Yu You
  - A RESTful Architecture for Adaptive and Multi-device Application Sharing: **62**
    Vlad Stirbu

First International Workshop on RESTful Design (WS-REST 2010)

Cesare Pautasso
Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland
cesare.pautasso@usi.ch

Erik Wilde
School of Information
UC Berkeley
Berkeley, CA 94720, USA
dret@berkeley.edu

Alexandros Marinos
Department of Computing
University of Surrey
Guildford, UK GU2 7XH
a.marinos@surrey.ac.uk

ABSTRACT

Over the past few years, the discussion between the two major architectural styles for designing and implementing Web services, the RPC-oriented approach and the resource-oriented approach, has been mainly held outside of traditional research communities. Mailing lists, forums and developer communities have seen long and fascinating debates around the assumptions, strengths, and weaknesses of these two approaches. The *First International Workshop on RESTful Design (WS-REST 2010)* has the goal of getting more researchers involved in the debate by providing a forum where discussions around the resource-oriented style of Web services design take place. *Representational State Transfer (REST)* is an architectural style and as such can be applied in different ways, can be extended by additional constraints, or can be specialized with more specific interaction patterns. WS-REST is the premier forum for discussing research ideas, novel applications and results centered around REST at the World Wide Web conference, which provides a great setting to host this first edition of the workshop dedicated to research on the architectural style underlying the Web.

Categories and Subject Descriptors

A.0 [**GENERAL**]: Conference proceedings

General Terms

Algorithms, Design, Languages, Standardization, Theory

Keywords

REST, HTTP, Web Architecture, SOA, Web Services, Service Design

1. INTRODUCTION

With the advent of *service orientation* and *Service Oriented Architecture (SOA)* as important new approaches for large-scale IT system design, it has become an important (and sometimes contentious) issue what to define as a "service". While many definitions in the scope of SOA stay on a very abstract level, eventually services need to be mapped to concrete IT architectures. In this area, there are two main design approaches. One approach has been to use a functional approach to services and define them in a way resembling a collection of *Messages* and *Remote Procedure Calls*; this approach has also been the underlying principles of existing middleware frameworks, such as CORBA. The other approach is to center the design on resources instead of functions; this approach has its main background in *Representational State Transfer (REST)* [3], the architectural style underlying the Web.

While discussions about REST and the advantages and limitations of this approach in the context of enterprise computing have gained some popularity, they were mostly conducted in email forums and discussion groups. The *First International Workshop on RESTful Design (WS-REST 2010)* has been planned to foster discussions around REST, its applications, and possible extensions or adaptations in the area of academic research. Interestingly, during the preparation time for the workshop, the `PATCH` method [1] for HTTP [2] has become a standard, which nicely illustrated the fact that REST (in that case, the uniform interface available through HTTP) is under active development. While we do not have any papers dealing with `PATCH` in the workshop, this topic alone (how to use the new method in RESTful designs, how to design representations for use with it, and how to expose/advertise these representations) would be a very interesting research area.

One of the main goals of WS-REST 2010 has been to bring application-oriented developers and academic research closer together, so that discussions about RESTful design can be both informed by real-world usage and constraints, and also benefit from research in the areas of information systems and information integration that has been ongoing for a long time. We believe that this first edition of the workshop has gathered a balanced and high-quality paper collection, selected from 28 submissions, setting a strong starting point for a workshop series that will be continued in the future.

2. PROGRAM

The main goal of WS-REST 2010 is to bring together practitioners and researchers. REST so far has mainly been discussed in more application-oriented forums, and in order to mirror this current situation and our goal to popularize RESTful design as a research topic and a design pattern

WS-REST 2010, April 26, 2010; Raleigh, NC, USA

in academia, we have decided to structure the workshop into three sessions, which are introduced with a thought-provoking keynote on *Willful Violations* by Sam Ruby.

2.1 REST Infrastructure

This session looks at how RESTful design can contribute to building infrastructures for open and adaptable service ecosystems. This means both a look at toolkits or frameworks which solve specific problems that occur in many RESTful designs, as well as a look at how the application of RESTful principles allows systems and frameworks to be designed in a way that makes the designs open and flexible.

- Leonard Richardson: *Developers Enjoy Hypermedia, But May Resist Browser-Based OAuth Authorization*
- Marc Hadley, Santiago Pericas-Geertsen and Paul Sandoz: *Exploring Hypermedia Support in Jersey*
- Savas Parastatidis, Jim Webber, Guilherme Silveira and Ian Robinson: *The Role of Hypermedia in Distributed System Development*
- Mike Kelly and Michael Hausenblas: *Using HTTP Link: Header for Gateway Cache Invalidation*
- Charles Engelke and Craig Fitzgerald: *Replacing Legacy Web Services with RESTful Services*

2.2 REST Research

This session looks at REST from a more fundamental perspective. While REST itself is defined by a number of constraints, it is informative and a possible road towards an evolution of the style to ask what constraints could be changed or added or removed and how that would affect the resulting style, and what patterns of information system design might not be adequately supported by the current style, and how that situation could be improved by changing, adding, or removing constraints. In addition general questions about how to combine, extend, and evolve RESTful service are within the cope of this session as well.

- Federico Fernandez and Jaime Navon: *Towards a Practical Model to Facilitate Reasoning about REST Extensions and Reuse*
- Antonio Garrote Hernández: *A Formal Definition of RESTful Semantic Web Services*
- Ian Jacobi and Alexey Radul: *A RESTful Messaging System for Asynchronous Distributed Processing*

2.3 Practical REST

REST infrastructure 2.1 and REST research 2.2 look at how to address patterns in RESTful design, or how to understand and/or evolve REST in general. In this last session, the focus is on specific applications of RESTful design principles, based on interesting case studies that highlight some of the lessons learned when applying REST.

- Petri Selonen, Petros Belimpasakis and Yu You: *Developing a RESTful Mixed Reality Web Service Platform*
- Vlad Stirbu: *A RESTful Architecture for Adaptive and Multi-device Application Sharing*

3. ORGANIZERS

- *Cesare Pautasso* is assistant professor in the new Faculty of Informatics at the University of Lugano, Switzerland. Previously he was a researcher at the IBM Zurich Research Lab and a senior researcher at ETH Zurich (Switzerland). His research focuses on building experimental systems to explore the intersection of model-driven software composition techniques, business process modeling languages, and autonomic/Grid computing. Recently he has developed an interest in Web 2.0 Mashups and Architectural Decision Modeling. He is the lead architect of JOpera, a powerful rapid service composition tool for Eclipse. His teaching, training, and consulting activities cover advanced topics related to Web Development, Middleware, Service Oriented Architectures and emerging Web services technologies. He is an active member of IEEE and ACM, where he has participated in more than 60 international conference program committees. He is has recently organized the *3rd International Workshop on Web APIs and Services Mashups at OOPSLA (Mashups'09)*. He is currently co-authoring a book on SOA with REST, to be published by Prentice Hall.
- *Erik Wilde* is associate adjunct professor at the UC Berkeley School of Information. He holds a diploma in computer science from the Technical University of Berlin, and a Ph.D. from the Swiss Federal Institute of Technology in Zürich, Switzerland. His general interest is Web architecture and Web-oriented information architecture. His research focus is on XML and related technologies, Web services and REST, and loosely coupled architectures for exposing data and services in easily usable and accessible ways.
- *Alexandros Marinos* is a researcher and doctoral candidate at the University of Surrey, at the Department of Computing. He has a special interest in digital ecosystems and is a researcher in the related EU-FP6 project OPAALS. With a background in web development, he has been endeavoring to bring the organizational paradigms of digital ecosystems and the architecture of the web together. As part of this effort, he has been involved in developing RETRO, a RESTful Transaction Model, and also a querying model for accessing relational databases through an HTTP-based API. Other interests include service composition on the web and RESTful service development using model-driven, declarative and rule-based approaches. He has received his MSc in Internet Computing from the University of Surrey in 2006.

4. PROGRAM COMMITTEE

The program committee of WS-REST 2010 did an excellent job of thoroughly reviewing all submissions to the workshop. Each submission was reviewed by at least three, often four PC members, which resulted in a fair reviewing process and a lot of helpful feedback for the authors. The program committee of WS-REST 2010 consisted of:

- Rosa Alarcon, Pontificia Universidad Catolica de Chile
- Subbu Allamaraju, Yahoo Inc., USA

- Mike Amundsen, USA
- Tim Bray, Sun Microsystems, USA
- Bill Burke, Red Hat, USA
- Benjamin Carlyle, Australia
- Stuart Charlton, Elastra, USA
- Joe Gregorio, Google, USA
- Michael Hausenblas, DERI, Ireland
- Ralf Kernchen, Centre for Communication Systems Research, University of Surrey, UK
- Rohit Khare, 4K Associates, USA
- Yves Lafon, W3C
- Francesco Lelli, University of Lugano, Switzerland
- Frank Leymann, University of Stuttgart, Germany
- Alexandros Marinos, Department of Computing, University of Surrey, UK
- Mark Nottingham, Yahoo Inc., Australia
- Cesare Pautasso, Faculty of Informatics, USI Lugano, Switzerland
- Ian Robinson, Thoughtworks, UK
- Richard Taylor, UC Irvine, USA
- Stefan Tilkov, innoQ, Germany
- Steve Vinoski, Verivue, USA
- Jim Webber, Thoughtworks, UK
- Erik Wilde, School of Information, UC Berkeley, USA
- Olaf Zimmermann, IBM Zurich Research Lab, Switzerland

5. REFERENCES

[1] Lisa Dusseault and James M. Snell. PATCH Method for HTTP. Internet RFC 5789, March 2010.

[2] Roy Thomas Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Internet RFC 2616, June 1999.

[3] Roy Thomas Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

Developers Like Hypermedia, But They Don't Like Web Browsers

Leonard Richardson
Canonical USA
leonardr@segfault.org

ABSTRACT

Although desktop developers often have trouble consciously understanding RESTful concepts like "hypermedia as the engine of application state", this does not prevent them from intuitively understanding client-side tools based on these concepts. However, I encountered unexpected developer resistance after implementing a security protocol I and other web developers had thought uncontroversial: the most common mechanism for authorizing OAuth request tokens. This developer resistance has implications for many web services that share their authentication credentials with a corresponding website.

Categories and Subject Descriptors

H.3.5 [Information Storage and Retrieval]: Online Information Services—Web-based services; H.5.4 [Information Interfaces and Presentation]: Hypertext/hypermedia.

General Terms

Human Factors

Keywords

REST, hypermedia, OAuth, OpenID, developer relations

1. INTRODUCTION

I am the lead developer of `lazr.restful`, a Python library for publishing RESTful web services in a Zope environment. The biggest `lazr.restful` site is Launchpad,[1] which hosts collaborative development for the Ubuntu Linux distribution, many of Ubuntu's component packages, and many unrelated open source software projects.

In late 2008 I told three stories[9] recounting my advocacy of RESTful design in the face of my colleagues' skepticism. A year later, I present two stories about everyday usage: what happens when outside developers start using a RESTful web service. The first story is about being proved right by your users; the second about what happens when the users rebel.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

2. EVERYBODY LOVES HYPERMEDIA

By general consensus, the most difficult RESTful constraint to grasp is "hypermedia as the engine of application state."[13] People who have trouble understanding HATEOAS in the context of web services understand it perfectly well in their everyday use of computers. Web browsers are based on HATEOAS. Ordinary computer users use an algorithm like this to accomplish something on a website: (I've translated the algorithm into RESTful terms.)

1. Retrieve a hypermedia representation of the home page.
2. Decode the representation to determine the current resource state.
3. Based on the representation's semantic cues, decide which hypermedia link or form is likely to bring you closer to your goal.
4. Click the link or fill out the form. Your browser will make another HTTP request and the result will be another hypermedia representation.
5. Go back to step 2 and repeat until the resource state is to your liking.

Although developers understand how the web works as well as non-developers, I've noticed two points of resistance when translating this algorithm into the world of web services. The first is in step 1, where the client starts at the well-known URI of the home page. Many developers prefer to use predefined rules to construct the URI of the object they "really" want to access, and go directly there. A simple, well-known example is the web service for the social bookmarking website del.icio.us, which describes its web service in a human-readable document, listing a number of URI "endpoints", each with a distinct function.[14] In violation of the HATEOAS principle, these useful URIs are nowhere to be found within the web service itself.

The second point of resistance is in step 3, with the idea that the state of a resource includes meta-information about its capabilities and its relationships to other objects. Some developers of web services prefer to keep this information (especially the information about capabilities) in human-readable form, and regard machine-readable hypermedia depictions as redundant.

Resistance to the HATEOAS principle is implicit in the design of many web services, and when I began work on the Launchpad web service, this resistance took the form of pushback from my colleagues. My perspective was not dismissed—I'd been hired specifically for my web service expertise—but I got a clear message that the Launchpad team's focus was on producing results, not exploring arcane theories.

"Results" in this context meant the kind of user-friendly development tools generally associated with SOAP/WSDL services. On the server side, it meant an easy way for developers to publish their existing data models as a web service. On the client-side, it meant a Python client which makes web service access idiomatically similar to local object access.

The competing vision for a Launchpad Python client was a library hard-coded with information about the Launchpad web service's "endpoints". This is a common design for custom web service clients, but it has one big disadvantage: these clients are brittle. They are written or generated based on a particular set of assumptions about the structure of the web service, and when the service changes, it violates those assumptions and the clients stop working.

Because of this, open source web service clients like `pyamazon` and `pydelicious` (for Amazon's ECS and the del.icio.us web service, respectively) have undergone serial changes of ownership. The web service changes and the library breaks, but the original client developer no longer has any active projects that use the web service. Someone with a more pressing need takes the project over, and the cycle repeats—or else the project is abandoned.

> "Originally written by Mark Pilgrim, I took over maintenance of the project in January 2004. and am now looking for somebody else who would be interested in taking over the maintenance of the project." [4]

> "pydelicious broke on the last del.icio.us API update and I was unable to contact the author so I'm posting the repaired code here going forward." [8]

Rather than pedantically explain the value of HATEOAS to my colleagues, I suggested that we exploit the hypermedia constraint to quickly ship a library that wouldn't have this problem. I proposed a client library whose exact capabilities would be determined by hypermedia served by the server. This client would present a Python interface corresponding to whatever hypermedia it received, analogous to the way a web browser displays a graphical representation of whatever hypermedia it receives.

My proposal became `launchpadlib`, a library that presents Launchpad as a densely interconnected network of Python objects similar to that found in an ORM library.[2] This network of objects corresponds exactly to the densely hyperlinked network of representations available from the web service.

In `launchpadlib`, the simplest way to get from one object to another is to follow a Python object reference. In web service terms, this corresponds to following a link. Save operations become PUT or PATCH requests, just as save operations in an ORM become database commands. Delete operations become DELETE requests. Here's some sample code:

```
>>> from launchpadlib import Launchpad
>>> service_root = Launchpad.login_with(
... "my application",
... "https://api.launchpad.net/beta/")
>>> my_account = service_root.me
>>> print my_account.name
Leonard Richardson
>>> my_account.name = "L. Richardson"
>>> my_account.lp_save()
```

Once `launchpadlib` was released I made an interesting discovery: a developer may have blind spots about the concept "hypermedia as the engine of application state", but they will use `launchpadlib` as if they did not have those blind spots. When developers ask me for help and send me code snippets, I see them using hypermedia as the engine of application state.

It would be hubristic to claim that `launchpadlib` is as easy to use as a web browser, but it's the same kind of tool as a web browser: a client programmed by hypermedia documents received from the server, presenting a number of possible next steps based on that hypermedia, each next step representing a change to the application state.

Whence this ease of use? Well, every `launchpadlib` session begins by constructing a "client" object. But this object doesn't just handle authentication and network details. It retrieves a hypermedia representation of the service's "home page". This automatically pushes the developer past step one of the HATEOAS algorithm, and past the first blind spot.

What about the second point of resistance, the reluctance to follow a link? The `launchpadlib` 'client' object offers a set of 'next steps' derived from the hypermedia representation. It's easy for a developer to load up a 'client' object in an interactive Python session and explore those 'next steps' by using the `dir()` command and following object references. This is the simplest way to explore the web service: it effectively turns `launchpadlib` into a web browser, allowing for "surfing".

The hypermedia algorithm is recursive, and once the developer follows one link, they might as well keep following links until they find what they're looking for. Once they're done exploring, the developer doesn't have to write any new Python code—they just have to clean up the code they wrote while exploring in the interactive session.

Launchpad's URIs do follow certain patterns: the URI path designating a 'person' resource is always "/~{username}". It's possible to craft a URL and load the representation of that resource directly into Launchpad, bypassing the normal workings of hypermedia. This is the equivalent of hacking the URI in your browser's address bar, and it's something a developer does a lot when they have a blind spot in step one of the HATEOAS algorithm. A web service that does not use hypermedia, like the del.icio.us service, assumes that a developers will write code to craft every URI their client accesses. When using the Launchpad web services, some developers do craft URIs for performance reasons, but I don't see it very often: it's easier to follow links from the 'home page'.

The very complexity of the Launchpad web service makes the hypermedia-based "surfing" style the more attractive option. If the

[2] The `launchpadlib` library is actually a thin Launchpad-specific wrapper around a more generic client library, `lazr.restfulclient`.

Launchpad web service only had a few kinds of resources, then a stripped-down, endpoint-based web service like the del.icio.us web service would be comprehensible. But actually the Launchpad web service has over sixty kinds of resources. Hypermedia is the best way to represent that diversity: it hides the parts you're not interested in behind links you didn't click. And it turns out developers love this style—as long as you don't tell them that the secret ingredient is "hypermedia as the engine of application state".

3. THE OAUTH REVOLT

Our developer-users ratified with their behavior our decision to write a hypermedia-based service and client. But an influential minority of our users rebelled against another one of our decisions, simply refusing to use the system as designed. I brokered a compromise which seemed promising enough to make it into an early draft of this paper, but which turned out to be untenable in the long term—untenable, it turns out, because of the way the World Wide Web embodies the HATEOAS constraint.

The Launchpad web service protects its resources with OAuth authentication[2], a request signing mechanism that depends on a 'access token' shared between client and server. An OAuth client like `launchpadlib` can obtain an access token just by asking the server, but an access token is useless until the end-user authorizes it, explicitly delegating some of their human authority to a computer program.

The OAuth standard does not define how the end-user is supposed to authorize an access token. We defined a protocol similar to the one used by other OAuth implementers like Twitter and Google, and similar to proto-OAuth mechanisms defined by providers like Flickr. [7, 1, 15] In our protocol, the `launchpadlib` application hands control over to the end-user's web browser and opens a web page on www.launchpad.net.

The web page explains to the end-user that an application 'foo' (the application that's using `launchpadlib`) wants access to their Launchpad account. The end-user may deny this request for access, may grant the application full access, or may grant limited access (like access to public data only). Once the end-user makes their decision, the access token is authorized (or revoked) and `launchpadlib` can begin using the Launchpad web service on the end-user's behalf (or not).

Here's the rationale for handing control over to the web browser: the end-user is in a tricky security situation. They're about to grant a third-party application access to their Launchpad account. We need to make it easy to distinguish between a legitimate delegation of authority and a phishing attack—an attempt to fraudulently obtain credentials by taking advantage of .people's natural tendency to give computers whatever information they ask for.

In my opinion, the best way to maintain the end-user's trust is to handle the authentication from their web browser. The browser is a trusted client. You already trust it with your passwords, and you trust your address bar not to lie about which server a web page came from. When the OAuth authentication process uses the web browser, the end-user can bring to bear all their experience in detecting web-based phishing attempts.

Launchpad bug #387297[11] summarizes what happened next. Several developers who were using `launchpadlib` in their third-party applications did not like this system, and routed around it. First they performed experiments, sniffing the HTTP interactions between `launchpadlib`, their web browser, and the Launchpad website. Then they wrote their own programs that asked for the end-user's username and password directly, and used screen-scraping and canned HTTP requests to simulate the browser-based authorization protocol we'd designed.

These developers weren't stupid. They knew how the system worked—they had to understand it in order to simulate it. They just didn't see the point. Stephan Hermann, a developer of a Launchpad desktop client called Leonov, wrote this about `launchpadlib`:[3]

> [T]he login and approval of `Launchpadlib` was a bit "strange" at the time when I looked at lplib. So I went and wrote a little wrapper class, which does the authentication and authorization (approval) automatically, without the need of a browser or interactive methods.

From Hermann's point of view, the API provider (my colleagues and me) did something "strange". Going along with our odd design would have inconvenienced his users. So he wrote a wrapper class that isolated the strange behavior. This let him tightly integrate Launchpad authorization into his native UI instead of jumping through bizarre browser-based hoops. The problem is, we designed the system specifically to prevent what Hermann wants to do.

Hermann isn't the only `launchpadlib` developer to rebel against our design. At least one other desktop application, Ground Control, uses a similar hack, and the ubuntu-dev-tools Ubuntu package includes a reusable script called `manage-credentials`.

The `manage-credentials` script takes a Launchpad username, password, and access level as command-line arguments. It logs in as the Launchpad user and grants itself a certain level of access. Here's the relevant code: [5]

```
# use hack
credentials =
    Credentials(options.consumer)
credentials =
    approve_application(credential,
      options.email,
      options.password,
      options.level,
      translate_api_web(options.service),
      None)
```

With this hack, the developer doesn't even have to let the end-user decide how much access they want to grant! Whoever calls `manage-credentials` can hard-code a certain value and get read-write access to the end-user's private data, without even telling the end-user there are other options.

Even this is not the worst of the desktop client depravity. While researching bug #387297, I heard of applications written before

Launchpad provided a web service, applications which crawled the end-user's Firefox profile looking for a saved Launchpad password. [M. Korn, personal communication] These were well-intentioned pieces of software designed to improve the end-user's interactions with Launchpad. But from an architectural standpoint, they were spyware.

Not all developers rebel against the browser-based security model. The F-Spot photo manager features integration with Flickr's web service, and rather than ask for your Flickr username and password, it tells you to click a button. Clicking the button opens up your web browser and begins Flickr's OAuth-like authentication protocol.

Similarly, many `launchpadlib` developers used `launchpadlib`'s browser-based authentication protocol without complaining (or at least without rebelling). But a prominent minority preferred to write hacks, to productize the hacks into scripts, and to include the scripts in official Ubuntu utility libraries.

I must admit that `launchpadlib` did not have the smoothest possible implementation of the credential-obtaining protocol (it's much better now). And our documentation doesn't shout out the security rationale for this protocol. It just says: "This lets your users delegate a portion of their Launchpad permissions to your program, without having to trust it completely."[10]

But even after reading an explanation of my point of view, Stephan Hermann preferred his original design, the one we tried to prohibit. In a comment on Launchpad bug #387297, he wrote:[11]

> Actually, I don't think there is a difference between trusting a webbrowser and an UI client... The approach with username + password is bad, but having no other chance to avoid a browser for ui clients, I think our leonov workaround is the best thing someone can do.

I interviewed Markus Korn, author of the `manage-credentials` script. He understands perfectly well how our OAuth protocol works; he just doesn't buy into the security rationale. When I asked him why he'd written `manage-credentials`, he told me: "The idea was to not bother the user with a web browser window when he is using a GUI." [M. Korn, personal communication]

I asked Korn what he would have said in his own defense if I'd confronted him while he was writing `manage-credentials`, telling him that he was subverting Launchpad's security model. He volunteered: "The user of my applications cares more about smoothness than security, because he trusts me as the developer of this app."[Korn, personal communication]

I also interviewed Martin Owens, the developer of the Ground Control desktop interface to Launchpad. He didn't think browser-based authentication was any more secure than a desktop application that asks the end-user for their Launchpad password: [M. Owens, personal communication]

> The web browser is a large application with arbitrary display and execution of code which comes from unknown and untrusted sources. It's got a very large attack area. It's got a fairly weak trust network... The user, I hope, would trust applications installed on their computer, especially if those applications are installed by default on the operating system CD...

It's not surprising that desktop developers put more trust in desktop applications than does a web developer like myself. If you install an application on your computer, you give that application as much implicit trust as you give your web browser. In a modern Linux environment, applications are typically open source and installed from trusted repositories, reducing the possibility that a given application will contain spyware.

Objectively speaking, Hermann, Korn, and Owens have the final say. The Launchpad web service was designed for toolmakers like them. Although hundreds of people use the Launchpad web service, they use it through applications like Leonov and Ground Control.

Because of this I decided to compromise with the toolmakers. I didn't like the idea of each desktop developer independently sniffing the token authorization protocol we'd designed for a web browser, and writing their own imitation browser to run through that protocol. Eventually one of those developers would make a mistake, leaking a Launchpad user's password or storing it in an insecure location. I also didn't like the way some of the imitation web browsers chose their own level of access to Launchpad, instead of leaving that decision up to the end-user.

My inspiration was `pinentry`, a suite of small desktop applications (part of the GnuPG project) which "read passphrases and PIN numbers in a secure manner." [6] The `pinentry` suite centralizes passphrase-gathering functionality in one simple, easily audited code base. I wrote a `pinentry`-like program, a canonical desktop application for taking the user's Launchpad password and authorizing an OAuth access token. Although this program duplicates the behavior of a web browser, I could at least keep every desktop developer from developing *their own* imitation browser.

I planned to package three different versions of this `pinentry`-like program with `launchpadlib`: one to blend in with GTK+ GUIs, one for Qt-based GUIs, and one for console applications. This would meet Korn's goal of "not bother[ing] the user with a web browser window." Instead of handing control over to the web browser, a developer would be able to hand control to a native desktop application that closely resembled their own desktop application.

I wrote a console-based application, `launchpad-credentials-console`, and was talking with desktop developers interested in writing GUI versions, when all my work was rendered obsolete by a change to the Launchpad website.

Up to this point, Launchpad was like most websites in having a special "login" page, which invited the end-user to type their username and password into an HTML form. My `launchpad-credentials-console` authenticated with Launchpad by constructing an HTML form submission and POSTing it to the appropriate Launchpad URL. Leonov, Ground Control, and

`manage-credentials` also authenticated with Launchpad using constructed form submissions.

In March 2010, the Launchpad login page disappeared. Launchpad users no longer give their username and password directly to Launchpad; they are now redirected to a OpenID provider, the Launchpad Login Service, and they authenticate with that OpenID provider. This is more convenient for the end-user, but it's disastrous for `launchpad-credentials-console` and all similar applications.

Consider a human being using their web browser to visit Launchpad, on the day after the old login page disappears. The home page still features an HTML link in the upper right-hand corner that says "Log In / Register". Clicking that link takes the end-user through a process in which they fill out HTML forms and submit them. At the end of the process, the end-user finds him or herself logged in to Launchpad.

On this day, the user's browser sends drastically different HTTP requests than the day before, but the differences between the old login procedure and the new one are encapsulated in hypermedia. When the end-user is using a hypermedia-aware client (ie. a web browser), the login system can change drastically and the user will not suffer anything worse than possible confusion. The "hypermedia algorithm" for obtaining a given application state still works.

Now consider a human being trying to run a `launchpadlib` script the day after the login procedure changes. At the crucial moment, when the end-user needs to authorize an OAuth request token, `launchpadlib` opens the end-user's web browser. The end-user is sent through the (new) login procedure and then gets a chance to authorize an OAuth request token or refuse authorization. Again, the change to the login procedure is encapsulated in hypermedia, which a web browser can always understand and a human being can always navigate.

Finally, consider someone trying to log in to Launchpad using `launchpad-credentials-console`. The end-user types their username and password into the console application, which sends a constructed HTML form submission. But Launchpad no longer recognizes that form submission! Launchpad is no longer in charge of handling login attempts; it can only redirect people to an OpenID provider. The `launchpad-credentials-console` program broke when the login procedure changed, in the same way `pyamazon` broke when Amazon's ECS service changed.

Leonov, Ground Control, `manage-credentials`, and `launchpad-credentials-console` all broke on the same day and for the same reason. These programs hid the workings of hypermedia ("arbitrary display... from unknown and untrusted sources") from the end-user, and now they're paying the price.

Can these applications be made to work again? In the short run, certainly. It's just HTTP. A developer can sniff the way browsers interact with the Launchpad Login Service, find the point at which the username and password are sent to the server, and teach a program to send the same request to the same URL.

In the long run, a hypermedia-oblivious program like `launchpad-credentials-console` cannot be made to work. The problem is not just that the login procedure might change again. We know the login procedure will change again, and it will change in a way that makes programs like `launchpad-credentials-console` impossible.

As of March 2010, Launchpad only accepts OpenID identifiers from one source: the Launchpad Login Service. In the future, Launchpad will be a full-fledged "relying party". End-users will be able to log in to Launchpad using an identifier from any OpenID provider: the Launchpad Login Service, the Ubuntu Single Sign On Service, Google, LiveJournal, MySpace, or any other.

When a user tries to log in to Launchpad, they will temporarily be redirected to their OpenID provider and asked to authenticate with their provider. Each provider has its own way of authenticating the end-user. Most OpenID providers use username-password combinations, as Launchpad used to and as the Launchpad Login Service does now, but each provider serves slightly different HTML forms and accepts slightly different HTTP requests. And nothing prevents an OpenID provider from authenticating with an x509 certificate, or defining an authentication procedure that makes the end-user solve a CAPTCHA or digitally sign a challenge string.

A web browser supports nearly any protocol for authorizing an OAuth access token. The differences between protocols are encapsulated in hypermedia and a human can navigate any protocol using the same, general hypermedia algorithm. Programming a hypermedia-oblivious client with all these possibilities is simply impossible. Once Launchpad's OAuth token authorization protocol allows authentication with an arbitrary OpenID provider, that authentication must take place in a web browser.

Desktop developers I've spoken with are understandably unhappy that their applications are broken. [Owens, personal communication.] But `launchpadlib`'s normal browser-based mechanism for authorizing OAuth tokens still works. The Launchpad team is working with Ubuntu desktop developers on a desktop-wide solution that should reduce the number of times an end-user has to use their web browser, but because it's very early in development, this paper is not a good place to discuss it.[12]

4. CONCLUSIONS AND RECOMMENDATIONS

When I designed the Launchpad web service, I expected one of my tasks would be developer education. After all, when designing the service I'd had to convince my colleagues of the benefits of RESTful design. But thanks to a hypermedia-aware client-side library, I found little conceptual resistance from developers. Our experience with web browsers shows that you don't have to understand "hypermedia as the engine of application state" to take advantage of it.

I did experience developer resistance to HATEOAS when it came to hypermedia experienced through the web browser. There, the problem wasn't the hypermedia; it was the browser. Although relatively few web services protect their resources with OAuth, I

predict that any web service that does will find its developers writing libraries along the lines of `manage-credentials`.

I propose a natural experiment: as I write, a client for the Twitter web service can authenticate its requests using an OAuth token, or by providing a Twitter username and password with HTTP Basic Auth. Twitter developers plan to deprecate Basic Auth starting in June 2010. [7] I predict that as Basic Auth is deprecated, client-side Twitter hackers will resist Twitter's OAuth token authorization protocol, just as client-side Launchpad hackers resisted Launchpad's similar protocol. How the Twitter developers will react to this resistance is an open question—especially if they ever intend to make Twitter an OpenID relying party.

It was frustrating to see `launchpad-credentials-console` suddenly break, along with all the other hypermedia-oblivious ways of authorizing Launchpad's OAuth tokens, but it also provided an object lesson in the value of hypermedia-aware clients. Regardless of desktop developers' reservations about the web browser, it's the only client that can authenticate with an arbitrary OpenID provider. More seriously, I think this problem illustrates a general obstacle towards OpenID adoption on websites that also provide web services.

Consider an alternate universe in which, by the beginning of 2010, Launchpad's web service somehow became as popular as Twitter's. Instead of four desktop clients that simulate a web browser to authenticate Launchpad's OAuth tokens, there would be dozens. Instead of an audience of a few hundred software developers, our web service would be used by millions of ordinary people.

What happens at this point if we decide to make Launchpad an OpenID relying party? We can't break dozens of clients and confuse millions of people. But the existing, hypermedia-oblivious clients won't allow a user to authenticate using an OpenID identity from (eg.) MySpace. Such a user would have to get a Launchpad account to use a web service client, defeating the purpose of making Launchpad an OpenID relying party.

When we saw that `launchpad-credentials-console` was broken, the Launchpad team had an internal discussion: should we fix the hypermedia-oblivious clients and forget about making Launchpad an OpenID relying party, or should we go ahead with our OpenID plans and abandon `launchpad-credentials-console`? This decision would have affected the entire Launchpad website, not just the web service.

For the sake of being good OpenID citizens, we decided to abandon the hypermedia-oblivious clients. The more popular a web service is, and the more hypermedia-oblivious clients there are in active use, the more difficult it will be to decide to make the corresponding website an OpenID relying party.

If your web service users authenticate using the same credentials they use on some corresponding website, give some thought to that site's future. If you protect your web service's resources with OAuth, you should decide now whether you ever want that site to be an OpenID relying party. If you use HTTP Basic Auth or some other authentication mechanism, make the same decision—and consider how your web service authentication mechanism might need to change.

5. ACKNOWLEDGMENTS

Thanks to Markus Korn and Martin Owens for explaining the desktop developer's perspective. Thanks also to my sister, Rachel Richardson, for formatting this paper.

6. REFERENCES

[1] Eric Bidelman
http://code.google.com/apis/gdata/articles/oauth.html

[2] E. Hammer-Lahav, Ed. "The OAuth 1.0 Protocol".
http://tools.ietf.org/html/draft-hammer-oauth-10

[3] S. Hermann, "Some internals...".
http://www.sourcecode.de/content/some-internals

[4] M. Josephson,
http://www.josephson.org/projects/pyamazon/

[5] M. Korn, "manage-credentials". Source code hosted in bzr repository. lp:ubuntu-dev-tools/manage-credentials.

[6] Werner Koch
http://www.gnupg.org/related_software/pinentry/index.en.html

[7] Raffi Krikorian et al.
http://apiwiki.twitter.com/OAuth-FAQ

[8] G. Pinero et al.
http://code.google.com/p/pydelicious/

[9] L. Richardson, "Justice Will Take Us Millions of Intricate Moves".
http://www.crummy.com/writing/speaking/2008-QCon/

[10] L. Richardson et al,
https://help.launchpad.net/API/launchpadlib

[11] L. Richardson, et al. "manage-credentials should not ask for Launchpad password directly".
https://bugs.edge.launchpad.net/launchpadlib/+bug/387297

[12] L. Richardson et al, "Trusted credential-management apps are broken and doomed",
https://bugs.launchpad.net/launchpadlib/+bug/532055

[13] J. Webber, "HATEOAS - The Confusing Bit from REST".
http://jim.webber.name/downloads/presentations/2009-05-HATEOAS.pdf

[14] Unknown author
http://delicious.com/help/api

[15] Unknown author
http://www.flickr.com/services/api/auth.spec.htm

Exploring Hypermedia Support in Jersey

Marc Hadley
Oracle
Burlington, MA, USA
marc.hadley@sun.com

Santiago Pericas-Geertsen
Oracle
Palm Beach Grdns, FL, USA
santiago.pericasgeertsen@sun.com

Paul Sandoz
Oracle
Grenoble, France
paul.sandoz@sun.com

ABSTRACT

This paper describes a set of experimental extensions for Jersey[9] that aim to simplify server-side creation and client-side consumption of hypermedia-driven services. We introduce the concept of *action resources* that expose workflow-related operations on a parent resource.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; H.4.3 [**Information Systems Applications**]: Communication Applications; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia

1. INTRODUCTION

The REST architectural style, as defined by Roy Fielding in his thesis [3], is characterized by four constraints: (i) identification of resources (ii) manipulation of resources through representations (iii) self-descriptive messages and (iv) hypermedia as the engine of application state. It is constraint (iv), hypermedia as the engine of application state or HATEOAS for short, that is the least understood and the focus of this paper. HATEOAS refers to the use of *hyperlinks* in resource representations as a way of navigating the state machine of an application.

It is generally understood that, in order to follow the REST style, URIs should be assigned to anything of interest (resources) and a few, well-defined operations (e.g., HTTP operations) should be used to interact with these resources. For example, the state of a purchase order resource can be updated by POSTing (or PATCHing) a new value for its *state* field.[1] However, as has been identified by other authors [5][7], there are *actions* that cannot be easily mapped to read or write operations on resources. These operations are inherently more complex and their details are rarely of interest to clients. For example, given a *purchase order resource*, the operation of setting its state to `REVIEWED` may involve a number of different steps such as (i) checking the customer's credit status (ii) reserving inventory and (iii) verifying per-customer quantity limits. Clearly, this *workflow* cannot be equated to simply updating a field on a resource. Moreover, clients are generally uninterested in the details behind these type of workflows, and in some cases computing the final state of a resource on the client side, as required for a PUT operation, is impractical or impossible.[2]

This paper introduces the concept of *action resources* and explores extensions to Jersey [9] to support them. An action resource is a sub-resource defined for the purpose of exposing workflow-related operations on parent resources. As sub-resources, action resources are identified by URIs that are relative to their parent. For instance, the following are examples of action resources:

```
http://.../orders/1/review
http://.../orders/1/pay
http://.../orders/1/ship
```

for purchase order "1" identified by `http://.../orders/1`.

Action resources provide a simplified hypertext model that can be more easily supported by generic frameworks like Jersey. A set of action resources defines—via their link relationships—a *contract* with clients that has the potential to evolve over time depending on the application's state. For instance, assuming purchase orders are only reviewed once, the `review` action will become unavailable and the `pay` action will become available after an order is reviewed.

The notion of action resources naturally leads to discussions about improved *client* APIs to support them. Given that action resources are identified by URIs, no additional API is really necessary, but the use of client-side proxies and method invocations to trigger these actions seems quite natural [7]. Additionally, the use of client proxies introduces a level of indirection that enables better support for *server evolution*, i.e. the ability of a server's contract to support certain changes without breaking existing clients. Finally, it has been argued [10] that using client proxies is simply more natural for developers and less error prone as fewer

[1]The choice of operation, such as POST, PUT or PATCH, for this type of update is still a matter of debate. See [6].

[2]For instance, when certain parts of the model needed by the workflow are not exposed as resources on the client side.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

URIs need to be constructed.

2. TYPES OF CONTRACTS

A contract established between a server and its clients can be *static* or *dynamic*. In a static contract, knowledge about the server's model is embedded into clients and cannot be updated without re-writing them. In a dynamic contract, clients are capable of discovering knowledge about the contract at runtime and adjust accordingly.

In addition, a dynamic contract can be further subdivided into *contextual* and *non-contextual*. Contextual contracts can be updated in the course of a conversation depending on the application's state; conversely, non-contextual contracts are fixed and independent of the application's state.

HATEOAS is characterized by the use of contextual contracts where the set of actions varies over time. In our purchase ordering system example, this contextual contract will prevent the `ship` action to be carried out before the order is paid, i.e. before the `pay` action is completed.

3. HUMAN VS. BOTS

In order to enable servers to evolve independently, clients and servers should be as decoupled as possible and everything that can change should be learned on the fly. The *Human Web* is based on this type of highly dynamic contracts in which very little is known *a priori*. As very adaptable creatures, humans are able to quickly learn new contracts (e.g. a new login page to access a bank account) and maintain compatibility.

In the *Bot Web*, on the other hand, contracts are necessarily less dynamic and must incorporate some static knowledge as part of the bot's programming: a bot that dynamically learns about some action resources will not be able to *choose* which one to use next if that decision is not part of its programming. It follows that at least a subset of the server's state machine—of which actions are transitions—must be statically known for the bot to accomplish some pre-defined task. However, if action descriptions (including HTTP methods, URI, query parameters, etc.) are mapped at runtime, then they need not be statically known. In fact, several degrees of coupling can be supported as part of the same framework depending on how much information is available statically vs. dynamically.

4. HYPERMEDIA IN JERSEY

Jersey [9] is the reference implementation of the Java API for RESTful Web Services [4]. In this section, we shall describe experimental extensions developed for Jersey to support HATEOAS, including a new client API based on Java dynamic proxies.

These Jersey extensions were influenced by the following (inter-related) requirements:

HATEOAS Support for *actions* and *contextual action sets* as first-class citizens.

Ease of use Annotation-driven model for both client APIs and server APIs. Improved client API based on dynamic generation of Java proxies.

Server Evolution Various degrees of client and server coupling, ranging from static contracts to contextual contracts.

Rather than presenting all these extensions abstractly, we shall illustrate their use via an example. The *Purchase Ordering System* exemplifies a system in which customers can submit orders and where orders are guided by a workflow that includes states like `REVIEWED`, `PAID` and `SHIPPED`.

The system's model is comprised of 4 entities: `Order`, `Product`, `Customer` and `Address`. These model entities are controlled by 3 resource classes: `OrderResource`, `CustomerResource` and `ProductResource`. Addresses are sub-resources that are also controlled by `CustomerResource`. An order instance refers to a single customer, a single address (of that customer) and one or more products. The XML representation (or view) of a sample order is shown below.[3]

```
<order>
    <id>1</id>
    <customer>http://.../customers/21</customer>
    <shippingAddress>
        http://.../customers/21/address/1
    </shippingAddress>
    <orderItems>
        <product>http://.../products/3345</product>
        <quantity>1</quantity>
    </orderItems>
    <status>RECEIVED</status>
</order>
```

Note the use of URIs to refer to each component of an order. This form of *serialization by reference* is supported in Jersey using JAXB beans and the `@XmlJavaTypeAdapter` annotation.[4]

4.1 Server API

The server API introduces 3 new annotation types: `@Action`, `@ContextualActionSet` and `@HypermediaController`. The `@Action` annotation identifies a sub-resource as a *named action*. The `@ContextualActionSet` is used to support contextual contracts and must annotate a method that returns a set of action names. Finally, `@HypermediaController` marks a resource class as a *hypermedia controller class*: a class with one more methods annotated with `@Action` and at most one method annotated with `@ContextualActionSet`.

The following example illustrates the use of all these annotation types to define the `OrderResource` controller.[5]

```
@Path("/orders/{id}")
@HypermediaController(
    model=Order.class,
    linkType=LinkType.LINK_HEADERS)
```

[3]Additional whitespace was added for clarity and space restrictions.

[4]This annotation can be used to customize marshalling and unmarshalling using `XmlAdapter`'s. An `XmlAdapter` is capable of mapping an object reference in the model to a URI.

[5]Several details about this class are omitted for clarity and space restrictions.

```
public class OrderResource {

    private Order order;

    @GET @Produces("application/xml")
    public Order getOrder(
      @PathParam("id") String id) {
        return order;
    }

    @POST @Action("review") @Path("review")
    public void review(
      @HeaderParam("notes") String notes) {
        ...
        order.setStatus(REVIEWED);
    }

    @POST @Action("pay") @Path("pay")
    public void pay(
      @QueryParam("newCardNumber") String newCardNumber) {
        ...
        order.setStatus(PAID);
    }

    @PUT @Action("ship") @Path("ship")
    @Produces("application/xml")
    @Consumes("application/xml")
    public Order ship(Address newShippingAddress) {
        ...
        order.setStatus(SHIPPED);
        return order;
    }

    @POST @Action("cancel") @Path("cancel")
    public void cancel(
      @QueryParam("notes") String notes) {
        ...
        order.setStatus(CANCELED);
    }
}
```

The `@HypermediaController` annotation above indicates that this resource class is a hypermedia controller for the `Order` class. Each method annotated with `@Action` defines a link relationship and associated action resource. These methods are also annotated with `@Path` to make a sub-resource.[6] In addition, `linkType` selects the way in which URIs corresponding to action resources are serialized: in this case, using link headers [11]. These link headers become part of the order's representation. For instance, an order in the `RECEIVED` state, i.e. an order that can only be reviewed or canceled, will be represented as follows.[7]

```
Link: <http://.../orders/1/review>;rel=review;op=POST
Link: <http://.../orders/1/cancel>;rel=cancel;op=POST
<order>
    <id>1</id>
    <customer>http://.../customers/21</customer>
    <shippingAddress>
        http://.../customers/21/address/1
    </shippingAddress>
    <orderItems>
        <product>http://.../products/3345</product>
        <quantity>1</quantity>
    </orderItems>
    <status>RECEIVED</status>
</order>
```

Link headers, rather than links embedded within the entity, were chosen for expediency. A full-featured framework would support both link headers and links embedded within entities but, for the purposes of this investigation, having links only in headers allowed for simpler, media-type independent, link extraction machinery on the client-side.

Without a method annotated with `@ContextualActionSet`, all actions are available at all times regardless of the state of an order. The following method can be provided to define a contextual contract for this resource.

```
@ContextualActionSet
public Set<String> getContextualActionSet() {
    Set<String> result = new HashSet<String>();
    switch (order.getStatus()) {
        case RECEIVED:
            result.add("review");
            result.add("cancel");
            break;
        case REVIEWED:
            result.add("pay");
            result.add("cancel");
            break;
        case PAID:
            result.add("ship");
            break;
        case CANCELED:
        case SHIPPED:
            break;
    }
    return result;
}
```

This method returns a set of action names based on the order's internal state; the values returned in this set correspond to the `@Action` annotations in the controller. For example, this contextual contract prevents shipping an order that has not been paid by only including the `ship` action in the `PAID` state.

Alternate declarative approaches for contextualizing action sets were also investigated but the above approach was chosen for this investigation due to its relative simplicity and expediency.

4.2 Client API

Although action resources can be accessed using traditional APIs for REST, including Jersey's client API [9], the use of client-side proxies and method invocations to trigger these actions seems quite natural. As we shall see, the use of client proxies also introduces a level of indirection that enables better support for server evolution, permitting the definition of contracts with various degrees of coupling.

Client proxies are created based on *hypermedia controller interfaces*. Hypermedia controller interfaces are Java interfaces annotated by `@HypermediaController` that, akin to the server side API, specify the name of a model class and the type of serialization to use for action resource URIs.

[6] There does not appear to be a need to use `@Action` and `@Path` simultaneously, but without the latter some resource methods may become ambiguous. In the future, we hope to eliminate the use of `@Path` when `@Action` is present.

[7] Excluding all other HTTP headers for clarity.

The client-side model class should be based on the representation returned by the server; in particular, in our example the client-side model for an `Order` uses instances of `URI` to link an order to a customer, an address and a list of products.

```
@HypermediaController(
    model=Order.class,
    linkType=LinkType.LINK_HEADERS)
public interface OrderController {

    public Order getModel();

    @Action("review")
    public void review(@Name("notes") String notes);

    @POST @Action("pay")
    public void pay(@QueryParam("newCardNumber")
        String newCardNumber);

    @Action("ship")
    public Order ship(Address newShippingAddress);

    @Action("cancel")
    public void cancel(@Name("notes") String notes);

}
```

The `@Action` annotation associates an interface method with a link relation and hence an action resource on the server. Thus, invoking a method on the generated proxy results in an interaction with the corresponding action resource. The way in which the method invocation is mapped to an HTTP request depends on the additional annotations specified in the interface. For instance, the `pay` action in the example above indicates that it must use a `POST` and that the `String` parameter `newCardNumber` must be passed as a query parameter. This is an example of a *static* contract in which the client has built-in knowledge of the way in which an action is defined by the server. RESTeasy [12] provides a client API that follows this model.

In contrast, the `review` action only provides a name for its parameter using the `@Name` annotation. This is an example of a *dynamic* contract in which the client is only coupled to the `review` link relation and the knowledge that this relation requires `notes` to be supplied. The exact interaction with the `review` action must therefore be discovered dynamically and the `notes` parameter mapped accordingly. The Jersey client runtime uses WADL fragments, dynamically generated by the server, that describe action resources to map these method calls into HTTP requests. The following shows the WADL description of the `review` action resource:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <doc xmlns:jersey="http://jersey.dev.java.net/"
    jersey:generatedBy="Jersey: ..."/>
  <resources base="http://localhost:9998/">
    <resource path="orders/1/review">
      <method name="POST" id="review">
        <request>
          <param type="xs:string" style="header"
            name="notes"/>
        </request>
      </method>
    </resource>
  </resources>
</application>
```

Essentially the WADL is used as a form to configure the request made by the client runtime. The WADL defines the appropriate HTTP method to use (`POST` in this case) and the value of the `@Name` annotation is matched to the corresponding parameter in the WADL to identify where in the request to serialize the value of the `notes` parameter.

The following sample shows how to use `OrderController` to generate a proxy to review, pay and ship an order. For the sake of the example, we assume the customer that submitted the order has been suspended and needs to be activated before the order is reviewed. For that purpose, the client code retrieves the customer URI from the order's model and obtains an instance of `CustomerController`.[8]

```
// Instantiate Jersey's Client class
Client client = new Client();

// Create proxy for order and retrieve model
OrderController orderCtrl = client.proxy(
    "http://.../orders/1", OrderController.class);

// Create proxy for customer in order 1
CustomerController customerCtrl = client.proxy(
    orderCtrl.getModel().getCustomer(),
    CustomerController.class);

// Activate customer in order 1
customerCtrl.activate();

// Review and pay order
orderCtrl.review("approve");
orderCtrl.pay("123456789");

// Ship order
Address newAddress = getNewAddress();
orderCtrl.ship(newAddress);
```

The client runtime will automatically update the action set throughout a conversation: for example, even though the `review` action does not produce a result, the HTTP response to that action still includes a list of link headers defining the contextual action set, which in this case will consist of the `pay` and `cancel` actions but not the `ship` action. An attempt to interact with any action not in the context will result in a client-side exception.

4.3 Server Evolution

Section 4 listed server evolution and the ability to support degrees of coupling as a requirement for our solution. In the last section, we have seen how the use of client proxies based on *partially* annotated interfaces facilitates server evolution.

An interface method annotated with `@Action` and `@Name` represents a *loose* contract with a server. Changes to action resource URIs, HTTP methods and parameter types

[8] `CustomerController` is a hypermedia controller interface akin to `OrderController` which is omitted since it does not highlight any additional feature.

on the server will not require a client re-spin. Naturally, as in all client-server architectures, it is always possible to break backward compatibility, but the ability to support more dynamic contracts—usually at the cost of additional processing time—is still an area of investigation. We see our contribution as a small step in this direction, showing the potential of using dynamic meta-data (WADL in our case) for the definition of these type of contracts.

5. RELATED WORK

In this section we provide a short overview of other REST frameworks that inspired our work. RESTeasy [12] provides a framework that supports client proxies generated from annotated interfaces. RESTfulie [8] is, to the best of our knowledge, the first public framework with built-in support for hypermedia.

5.1 RESTeasy Client Framework

The RESTeasy Client Framework follows a similar approach in the use of client-side annotations on Java interfaces for the creation of dynamic proxies. It differs from our approach in that it neither supports hypermedia nor the ability to map proxy method calls using dynamic information. That is, in the RESTeasy Client Framework, proxy method calls are mapped to HTTP requests exclusively using static annotations. These client-side annotations establish a tight coupling that makes server evolution difficult.

Despite these shortcomings, we believe their approach is a good match for certain types of applications, especially those in which servers and clients are controlled by the same organization. In addition, the use of client proxies simplifies programming and improves developer productivity. For these reasons, we have followed a similar programming model while at the same time provided support for hypermedia and dynamic contracts.

5.2 RESTfulie Hypermedia Support

This article [7] by Guillerme S. describes how to implement hypermedia aware resources using RESTfulie. In RESTfulie, action URIs are made part of a resource representation (more specifically, the entity) via the use of Atom links [14]. Even though we foresee supporting other forms of URI serialization (as indicated by the use of `linkType` in `@HypermediaController`), we believe link headers to be the least intrusive and most likely to be adopted when migrating existing applications into hypermedia-aware ones.[9]

Rather than providing an explicit binding between actions and HTTP methods, RESTfulie provides a pre-defined table that maps `rel` elements to HTTP methods. For example, an `update` action is mapped to `PUT` and a `destroy` action is mapped to `DELETE`. We believe this implicit mapping to be unnecessarily confusing and not easily extensible. Instead, as we have done in this paper, we prefer to make *action resources* first class and provide developers tools to define explicit mappings via the use of `@Action` annotations.

In RESTfulie, knowledge about action resources is discovered dynamically and, as a result, Java reflection is the only mechanism available to interact with hypermedia-aware resources.[10] So instead of writing `order.cancel()`, a developer needs to write:

```
resource(order).getTransition("pay").execute()
```

As explained in Section 3, this decision is impractical given that certain static knowledge is required in order to program bots. We believe that Java interfaces annotated with `@Action` and `@Name` provide the right amount of static information to enable bot programming and server evolution whilst not sacrificing the ease of use offered by client proxies.

6. CONCLUSIONS

In this paper we have introduced the notion of an action resource and, with it, described some experimental extensions for Jersey to support HATEOAS. Our analysis lead us into the exploration of innovative client APIs that enable developers to define client-server contracts with different degrees of coupling and improve the ability of servers to evolve independently. In the process, we also argued that there is a minimum amount of static information that is needed to enable programmable clients (bots) and showed how that information can be captured using client interfaces.

All the source code shown in the previous sections is part of a complete and runnable sample [15] available in the Jersey subversion repository. The solution proposed herein is experimental and is likely to evolve in unforeseen directions once developers start exploring HATEOAS in real-world systems.

We are currently evaluating support for entity-based action resources URIs. Support for Atom link elements (as has been proposed by other authors) is a likely avenue for exploration since their use, like that of Link headers, permits support of many different XML-based media types without requiring media-type specific machinery. Different ways in which contextual action sets are defined are being explored, as well as simplifications to the client APIs for the instantiation of dynamic proxies, especially those created from other proxies.

The authors would like to thank Martin Matula for his suggestions on how to improve the sample and Gerard Davison for reviewing our work and providing insightful comments.

7. REFERENCES

[1] B. Burke. *RESTful Java with JAX-RS*. O'Reilly, 2009.
[2] R. Chinnici and B. Shannon. Java Platform, Enterprise Edition (JavaEE) Specification, v6. JSR, JCP, November 2009. See http://jcp.org/en/jsr/detail?id=316.
[3] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Ph.d dissertation, University of California, Irvine, 2000. See http://roy.gbiv.com/pubs/dissertation/top.htm.

[9]For example, existing applications that use XML schema validation on the entities.

[10]The exact manner in which the RESTfulie runtime maps parameters in an HTTP request is unclear to us at the time of writing.

[4] M. Hadley and P. Sandoz. JAXRS: Java API for RESTful Web Services. JSR, JCP, September 2009. See http://jcp.org/en/jsr/detail?id=311.

[5] T. Bray. RESTful Casuistry. Blog, March 2009. See http://www.tbray.org/ongoing/When/200x/2009/03/20/Rest-Casuistry.

[6] R. Fielding. It is okay to use POST. Blog, March 2009. See http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post.

[7] G. Silveira. Quit pretending, use the web for real: restfulie. Blog, November 2009. See http://guilhermesilveira.wordpress.com/2009/11/03/quit-pretending-use-the-web-for-real-restfulie.

[8] RESTfulie. See http://freshmeat.net/projects/restfulie.

[9] JAX-RS reference implementation for building RESTful web services. See https://jersey.dev.java.net.

[10] Why HATEOAS? Blog, April 2009. See http://blogs.sun.com/craigmcc/entry/why_hateoas.

[11] Web Linking (draft). Mark Nottingham. http://tools.ietf.org/html/draft-nottingham-http-link-header-06.

[12] RESTeasy Client Framework. See http://www.jboss.org/file-access/default/members/resteasy/freezone/docs/1.2.GA/userguide/html_single/index.html#RESTEasy_Client_Framework.

[13] Web Application Description Language (WADL). Marc Hadley. See https://wadl.dev.java.net.

[14] Atom Syndication Format. See http://www.w3.org/2005/Atom.

[15] Jersey Hypermedia Sample. See http://tinyurl.com/jersey-hypermedia.

The Role of Hypermedia in Distributed System Development

Savas Parastatidis
Microsoft
savas@parastatidis.name

Jim Webber
ThoughtWorks
jim@webber.name

Guilherme Silveira
Caelum
guilherme.silveira@caelum.com.br

Ian S Robinson
ThoughtWorks
iansrobinson@gmail.com

ABSTRACT
This paper discusses the role of the REpresentational State Transfer (REST) architectural style in the development of distributed applications. It also gives an overview of how RESTful implementations of distributed business processes and structures can be supported by a framework such as Restfulie.

Categories and Subject Descriptors
D1.0 [**Programming Techniques**]: General. D.2.10 [**Design**]: Methodologies, Representation, D.2.11 [**Software Architectures**]: Patterns

General Terms
Design, Reliability, Experimentation.

Keywords
REST, Hypermedia, Distributed Applications, Distributed Computing, Web, Web services, Business Processes.

1. INTRODUCTION
Embracing HTTP as an application protocol puts the Web at the heart of distributed systems development. But that's just a start. Building RESTful distributed systems requires more than the adoption of HTTP and the remainder of the Web technology stack [1]. In order to develop a system that works in harmony with the Web, one needs to carefully model distributed application state, business processes that affect that state, distributed data structures which hold it, and the contracts and protocols that drive interactions between the consituent parts of the system. The key REST concept of hypermedia is a design pattern that can greatly help building software to meet these demands. It enables the construction of systems that can easily evolve, adapt, scale, and be robust to failures by taking advantage of the underlying Web infrastructure.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

To bootstrap our understanding of hypermedia, we first reintroduce REST's "Hypermedia as the Engine of Application State" (HATEOAS) principle, applied in a modern distributed systems environment. We then show how to use the HATEOAS principle to construct protocols as the building blocks for applications. Finally, we describe how an open source framework and runtime, called Restfulie, can implement such building blocks to support the development and deployment of RESTful systems.

2. HYPERMEDIA AS THE ENGINE OF APPLICATION STATE
If we think of an application as being computerized behavior that achieves a goal, we can describe an application protocol as the set of legal interactions necessary to realize that behavior. Application state is a snapshot of the execution of such an application protocol. The protocol defines the interaction rules that govern the interactions between participants in a system. Application state is a snapshot of the system at an instant in time.

Fielding coined the phrase "Hypermedia as the Engine of Application State," to describe a core tenet of the REST architectural style [2]. In this paper, we refer to HATEOAS as the "hypermedia constraint". Put simply, this constraint says that hypermedia drives systems to transform application state.

A hypermedia system is characterized by participants transferring resource representations that contain links according to an application protocol. Links advertise other resources participating in the application protocol. Links are often enhanced with semantic markup to give domain-specific meanings to the resources they identify. For example, in a consumer-service interaction, the consumer submits an initial request to the entry point of the service. The service handles the request and responds with a resource representation populated with links. The consumer chooses one of these links and interacts with the resource identified by the link in order to transition to the next step in the interaction, whereupon the process repeats. Over the course of several such interactions, the consumer progresses towards its goal. In other words, the distributed application's state changes.

Transformation of application state is the result of the systemic behavior of the whole: the service, the consumer, the exchange of hypermedia–friendly resource representations, and the acts of advertising and selecting links. On each interaction, the service and consumer exchange representations of resource state, which serve to alter application state.

2.1 Resource Representations

There is much confusion regarding the relationship between resources and their resource representations. Yet their reponsibilities are quite obviously different on the Web.

The Web is so pervasive that the HTTP URI scheme is today a common synonym for both identity and addressing. Resources must have at least one identifier to be addressable. Furthermore, although the terms "resource representation" and "resource" are often used interchangeably, it is important to understand that there is a difference and that there exists a one-to-many relationship between a resource and its representations. A representation is a transformation or a view of a resource's state at an instant in time as encoded in one or more transferable formats, such as XHTML, XML, Atom, JSON, etc.

For real-world resources, such as goods in a warehouse, we can distinguish between the ultimate referent, the "thing itself", and the logical resource encapsulated by a service. It's this logical resource which is made available to interested parties through its representations. By distinguishing between the physical and logical resource, we recognize that the ultimate referent may have many properties that are not captured in its logical counterpart, and which, therefore, do not appear in any of its representations. Of course, there are some resources, such as emails, where the ultimate referent is indistinguishable from the information resource. Semiotic niceties aside, we're primarily interested in representations of information resources, and where we talk of a resource or "underlying resource" it's the information resource to which we're referring.

Access to a resource is always mediated by way of its representations. That is, Web services exchange representations of their resources with consumers. They never provide access to the actual state of the underlying resources directly – the Web does not support pointers! URIs are used to relate, connect, and associate representations with their resources on the Web.

Since the Web doesn't prescribe any particular structure or format for resource representations, they may take the form of a photograph, a video, a text file, or comma-separated values. Given the range of options for resource representations, it might seem that the Web is far too chaotic a choice for integrating computer systems where fewer, structured formats – such as JSON, or XML formats like Atom – are preferred. However careful choice of representation formats can constrain the Web enough for computer-to-computer interactions through hypermedia-driven protocols.

3. STRUCTURAL HYPERMEDIA

We are all familiar with the use of hypermedia on the Web as a way to transition from one Web page to another. The use of hypermedia controls, or links, to enable the identification of forward paths in our exploration of the information space is what we call "structural hypermedia."

On the human Web, structural hypermedia is used for the representation of "linked" documents. A Web browser enables the transition from one document to another on demand. The approach respects the underlying network. Information is loaded as lazily as possible, and the user is encouraged to browse pages – traverse a hypermedia structure – to access information. Breaking information into hypermedia-linked structures decreases the load on a service by reducing the amount of data that has to be served. Instead of downloading the entire information model, the application transfers only the parts pertinent to the user.

Not only does this laziness reduce the load on Web servers, the partitioning of data across pages on the Web allows the network infrastructure itself to cache information. An individual page, once accessed, may be cached depending on the caching policy for the page and service. As a result, subsequent requests for the same page along the same network path may be satisfied using a cached representation, which in turn further reduces load on the origin server.

Importantly, the same is true of computer-to-computer systems: structural hypermedia allows sharing of information in a lazy and cacheable manner, thereby enabling the composition of business data from a distributed dataset in a scalable and performant manner. While this strategy may incur a cost in terms of transactional atomicity, this is not a major concern since contemporary distributed systems design tends to favor other models for consistent outcomes that don't sacrifice scalability.

As an example of a hypermedia-enabled resource representation format, we refer the reader to Atom [3] and its increasing use in business environments. Atom makes use of hypermedia controls to bring together lists of information, which in turn reference other business resources. Using Atom, we can create distributed, connected, and deduplicated datasets by composing and navigating Atom feeds across the Web.

4. MODELLING AND IMPLEMENTING DISTRIBUTED APPLICATION BEHAVIOR

If we can model distributed data structures or information using hypermedia, it's a logical assertion to suggest that we can do the same for behavior. An application makes forward progress by transitioning resources from one state to another, which affects the entire application state. Using hypermedia we can model and advertise permitted transitions. Software agents can then decide which possible forward steps they wish to activate based on their interpretation of the application state in the context of a specific business goal.

Observing that automata can take advantage of hypermedia means that computerized business processes can be modeled and implemented using HATEOAS and hypermedia-enabled resource representations. As parts of the same distributed system interact with one another, they exchange resource representations containing links, the activation of which modify the state of the application. The services sending those resource representations can dynamically change the included links based on their understanding of the state of the resources they control.

4.1 Domain Application Protocols

We promote the notion that a service supports a *domain application protocol* or DAP by advertising subsequent legal interactions with relevant resources. When a consumer follows links embedded in resource representations and subsequently interacts with the linked resources, the application's overall state changes.

DAPs specify the legal interactions between a consumer and a set of resources involved in a business process. They sit atop HTTP

and narrow HTTP's broad application protocol to support specific business scenarios. As we shall see, services implement DAPs by adding hypermedia links to resource representations. The links highlight other resources with which a consumer can interact to make progress through a business transaction.

In hypermedia systems, changes of application state resemble a workflow or business process execution, which implies we can build services that advertise workflow using hypermedia protocols. Hypermedia makes it easy to implement business protocols in ways that reduce coupling between services and consumers. Rather than understand a specific URI structure, a consumer need only understand the semantic or business context in which a link appears. This reduces an application's dependency on static metadata such as URI templates or WADL [4]. As a consequence, services gain a great deal of freedom to evolve without breaking consumers (since consumers are loosely bound to the service via its supported media types and link relations only).

A domain application protocol is associated with a contract that describes its behavior. In turn, a contract represents a collection of protocols, each of which consists of HTTP idioms, entry point URIs for the application, media types, and link relations. A media type is a collection of hypermedia representation formats (Figure 1).

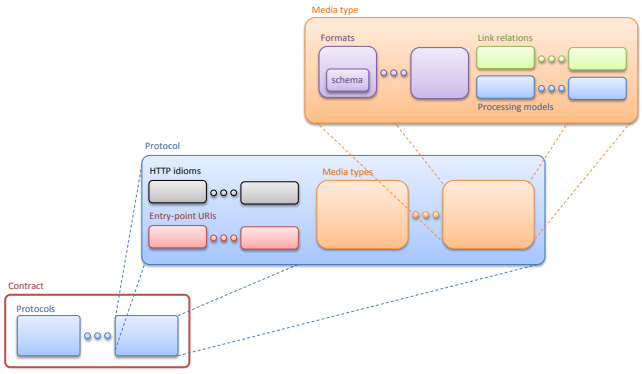

Figure 1. Contracts are composed of protocols. A protocol consists of a collection of media types, URI entry points, and HTTP idioms. A media type is a collection of resource representation hypermedia formats.

Services should ensure that any changes they introduce do not violate contracts with existing consumers, which would break their DAP. Whilst it is fine for a service to make structural changes to the relationships between its resources, semantic changes to the domain application protocol, and changes to the media types and link relations used may change the contract and break existing consumers. The Web is not a license to be a bad citizen.

4.1.1 Contracts

Contracts are a critical part of any distributed system since they prescribe how disparate parts of an application should interact. They typically encompass data encodings, interface definitions, policy assertions, and coordination protocols. Data encoding requirements and interface definitions establish agreed mechanisms for composing and interpreting message contents to elicit specific behaviors. Policies describe interoperability preferences, capabilities and requirements—often around security and other quality of service attributes.

Coordination protocols describe how message exchanges can be composed into meaningful conversations between the disparate parts of an application in order to achieve a specific application goal.

The Web breaks away from traditional thinking about upfront agreement on all aspects of interaction for a distributed application. Instead, the Web is a platform of well-defined building blocks from which distributed applications can be composed. Hypermedia can act as instant and strong composition glue.

Contracts for the Web are quite unlike static contracts for other distributed systems. They compose media types with protocols that extend the capabilities of a media type into a specific domain.

Currently, there is no declarative notation to capture all aspects of a contract on the Web. While technologies like XML Schema allow us to describe the structure of documents, there is no vocabulary that can describe everything. As developers, we have to read protocol and media type specifications in order to implement applications based on contracts.

4.1.2 Media Types

The core of any contract on the Web is the set of media types that a service supports. A media type specification sets out the formats (and any schemas), processing model, and hypermedia controls that services will embed in representations.

There are numerous existing media type specifications which we can select to meet the demands of our service. Occasionally we may create new media types to fit a particular domain. The challenge for service designers is to select the most appropriate media type(s) to form the core service contract.

On entering into the contract, consumers of a service need simply agree to the format, processing model and link relations found in the media type(s) the service uses. If common media types are used (e.g. XHTML or Atom) widespread interoperability is readily achievable since there are many existing systems and libraries that support these types.

We believe that an increase in the availability of media type processors will better enable us to rapidly construct distributed applications on the Web. Instead of coding to static contracts, we will be able to download (or build) standard processors for a given media type and then compose them together.

Often that's as far as we need to go in designing a contract. By selecting and composing media types, we've got enough collateral to expose a contract to other systems. However we need not stop there, and can refine the contract by adding protocols.

4.1.3 Protocols

On the Web, protocols extend the base functionality of a media type by adding new link relations and processing models.

A classic example of protocols building on established media types is the Atom Publishing Protocol [5]. AtomPub describes a number of new link relations, which augment those declared in the Atom syndication format. It builds on these link relations to

create a new processing model for the management of Atom feeds and entries.

Where media types help us with the interpretation and processing of the formats, link relations help us understand why we might want to follow a link. A protocol may add new link relations to the set provided by the existing media type(s). It may also augment the set of HTTP idioms used to manipulate resources in the context of specific link relations.

Services designers can also use independently defined link relations, such as those in the IANA link relation registry, mixing them in with the link relations provided by media types and protocols to advertise specific interactions.

4.2 Putting Everything Together – The Restbucks Coffee Shop

We are now armed with enough information to start building distributed applications using hypermedia. A consuming application can use the entry point of a service to start the interaction. From there, the DAP will guide the interactions by embedding links in the resource representations that are exchanged. The consumer may jump from one service to another, making forward progress in the entire business process along the way, as Figure 2 illustrates.

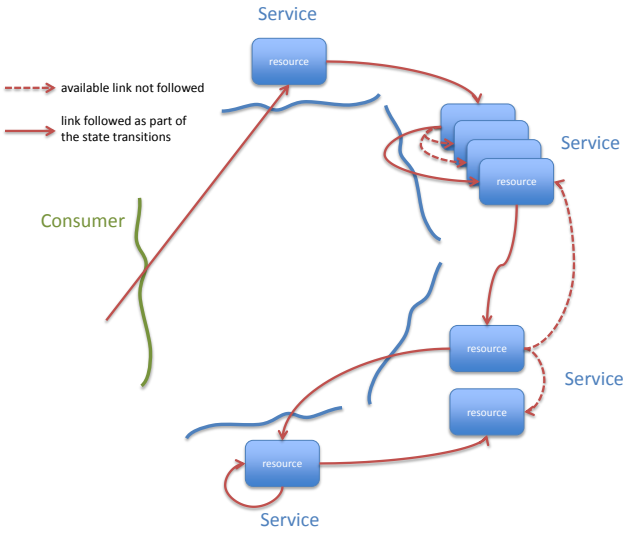

Figure 2. Following a DAP in a business transaction.

In order to illustrate the principles described in this paper, we have designed the "Restbucks" online coffee service. The inspiration for our problem domain came from Gregor Hohpe's observation on how a busy coffee shop works. In his popular blog entry,[1] Hohpe talks about synchronous and asynchronous messaging, transactions, and scaling the message-processing pipeline in an everyday situation. We liked the approach very much and as believers that "imitation is the sincerest form of flattery," we adopted Gregor's scenario.[2]

[1] http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html

[2] In fact we liked the scenario so much that we put it at the heart of our forthcoming book "REST in Practice" (O'Reilly 2010).

The work on Restbucks illustrates the way in which business processes could be modeled and implemented using Hypermedia and the Web technology stack. For example, Figure 3 illustrates the ordering service of Restbucks and how it makes use of HTTP verbs to progress a business transaction.

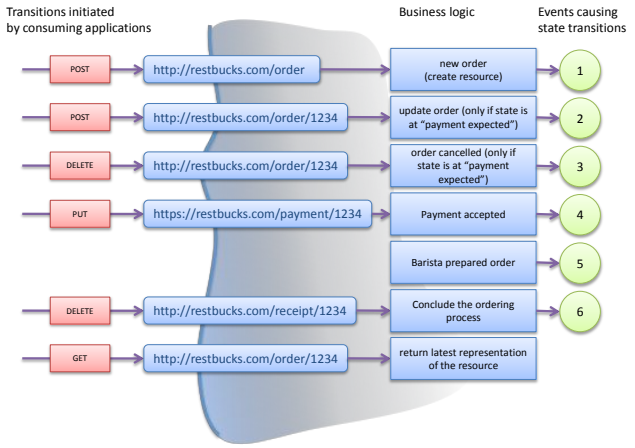

Figure 3. The supported HTTP interactions with the Restbucks ordering service, how they connect to the backend business logic, and the state transitions of the order resource.

The Restbucks DAP only describes the entry point for an order. The rest of the URIs will be returned in the resource representations. For example, the HTTP response to the POST request for a new order will contain a resource representation with links that allow the consumer to update or delete the order, submit payment, or check the status of the order. The Restbucks media types define the format of the representations and the supported semantics for the links. Ultimately, the interactions cause the "order" resource to transition between a set of states (Figure 4). Of course, the state of the "order" resource is only part of the entire application's state at any particular point in time.

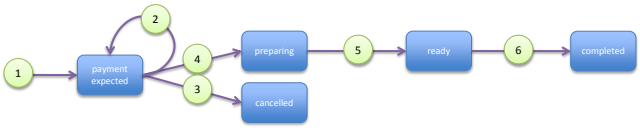

Figure 4. The state machine of an order resource.

4.3 URIs and Loose Coupling

We have explicitly identified the need for an entry point into a service, which is identified as part of a DAP description. Applications that do not use hypermedia to navigate through a business process or structural information tend to use out of band mechanisms (e.g. URI Templates) to advertise the existence of resources. In turn, this leads to tight coupling and makes it very difficult for applications to evolve and change.

5. THE RESTFULIE HYPERMEDIA FRAMEWORK

Restfulie is a software development and runtime framework that emphasizes structural and behavior hypermedia [6]. Restfulie makes it easy for developers to apply the REST architectural principles and implement them in a manner that uses the Web as an application platform.

5.1 Restfulie's Architectural Tenets

Unlike Web development frameworks that attempt to hide the primitives of the underlying distributed application platform and promote type and/or contract sharing, Restfulie explicitly promotes loose coupling between services and their consuming applications. In Restfulie, there is no type or static contract sharing. Instead, its API is built around the principles of content type negotiation, hypermedia, and domain application protocols.

The Restfulie framework supports the seamless incorporation of well-known and custom hypermedia media type formats in the development process. It promotes content type negotiation so that consumers and services can dynamically agree on the best (hypermedia) resource representation for their interactions. It is unique amongst other Web application development frameworks in that it extracts resource relations and possible state transitions from exchanged representations and exposes them through its API. By requiring developers to deal explicitly with hypermedia concepts, it guides and helps them in building truly RESTful systems.

Jersey [7], RESTEasy [8], and other similar frameworks require the use of programmatic annotations in order to associate HTTP verbs and URIs with business logic. This results to rigid early binding. We believe that early, static binding is suboptimal for the dynamic, Web-based, scalable, loosely-coupled distributed applications of today. In contrast, Restfulie allows relations and transitions to be dynamically discovered, which is a core feature of linked, semantically rich systems. As well as providing this late binding mechanism, Restfulie also provides support for applications that have prior knowledge of a service's links, formats, and protocols.

5.2 Resources and Transitions

Restfulie's architecture is very much platform agnostic, which is to be expected given that it is based on the REST principles that we discussed in the previous sections. To date, Restuflie has been implemented in Rails, Java, and .NET, and a port to Erlang is underway. In this paper, we focus on the Rails implementation.

First we show how the ordering service of our Restbucks coffee shop can be built in Rails with Restfulie. We concentrate on the management of an order's state, as per the state machine shown in Figure 4. Our service is going to consume and serve "order" resource representations, a typical example of which is shown in Figure 5.

```
<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>2.0</cost>
  <status>payment-expected</status>
</order>
```

Figure 5 An example of a Restbucks order

We create an `Order` model in Rails (Figure 6) to represent the order's state machine. Each state restricts the actions available to consumers of the model. For example, one can "cancel" the order only if the order is in the "unpaid" state. Also note that in some cases an action may result to the order's (or some other resource's) transition to a new state. For example, when the order is in the "unpaid" state, a payment can be issued, which is implemented by a different resource. Rails allows us to model the resource's state transitions in an intuitive manner. Restfulie makes sure that the model will be properly mapped to hypermedia-friendly primitives.

```
class Order << ActiveRecord::Base
  acts_as_restfulie do |order, t|
    t << [:self, :action => :show}]
    t << [:retrieve, :id => order, :action => :destroy}] if
order.is_ready?
    t << [:receipt, :order_id => order, :controller =>
:payments, :action => :receipt}] if order.status=="delivered"
    if order.status=="unpaid"
      t << [:cancel, :action => :destroy}]
      t << [:payment, :action => :create, :controller =>
:payments, :order_id => order.id}]
      t << [:update]
    end
  end
end
```

Figure 6 Declaring an Order in Restfulie

Restfulie will manage the lifecycle of an Order resource based on the Rails model of Figure 6. It automatically enables/disables the appropriate HTTP idioms on the resource, as shown in Figure 3, and includes the appropriate hypermedia controls in the exchanged resource representations. These controls allow agents to transition from one resource to another or from one resource state to another (legitimate) state. Using these controls, agents can make forward progress in the modeled business process—which in this case is the ordering process.

5.3 Media Types

Restfulie and Rails enable us to model an order's lifecycle in isolation of the resource representation's hypermedia format. As a result, we can leverage HTTP's content negotiation so that consumers of the ordering service can indicate their preferred format for the order-related interaction. Of course, the ordering service has to be configured to support the chosen hypermedia format. It's up to the service implementer to choose how many representation formats they wish to support and how the order resource and the state machine should be mapped to each format.

The Rails code in Figure 7 shows how we can modify the model of Figure 6 to make Restfulie aware of our custom media type, `application/vnd.restbucks+xml`, for an `Order` resource representation. Our customer media type defines the resource representation format for an order, and uses `atom:link` elements to represent hypermedia controls.

```
class Order << ActiveRecord::Base
  media_type 'application/vnd.restbucks+xml'
  acts_as_restfulie do |order, t|
    ...
end
```

Figure 7 Declaring a media type for a resource representation

Once the code of Figure 6 has been executed, and assuming the resource is in the "payment expected" state, an HTTP `GET` request to `http://restbucks.com/order/5` will return a custom media type representation like the one of Figure 8. Note the Atom links, which are automatically included by Restfulie to indicate the possible ways the recipient of the resource representation could make forward progress in the business interaction, as per the hypermedia principles we discussed in previous sections.

```
<order>
  <created-at>2010-01-09T15:18:29Z</created-at>
  <location>take-away</location>
  <status>unpaid/status>
  <updated-at>2010-01-09T15:18:29Z/updated-at>
  <cost>10/cost>
  <items>
    <item>
      <created-at>2010-01-09T15:18:29Z/created-at>
      <drink>latte/drink>
      <milk>whole/milk>
      <size>large</size>
      <updated-at>2010-01-09T15:18:29Z/updated-at>
    </item>
  </items>
  <atom:link rel="self" href="http://restbucks.com/orders/5"/>
  <atom:link rel="cancel"
             href="http://restbucks.com/orders/5"/>
  <atom:link rel="payment"
             href="http://restbucks.com/orders/5/payment"/>
  <atom:link rel="update"
             href="http://restbucks.com/orders/5"/>
</order>
```

Figure 8 Custom media type representation of an order

When generating a representation, Restfulie will check the order's resource state and its state machine in order to generate the set of relations as links, which are then translated to the appropriate hypermedia controls supported by the chosen hypermedia format.

5.4 Restfulie and the Web

Restfulie supports all those Web-related idioms one might expect from a modern Web application framework. Some of the major features include:

- Support for the Create, Retrieve, Update, and Delete operations that are common in a resource-oriented distributed application;
- Functionality so that developers can easily incorporate distributed state management logic in the actions of their models. For example, Restfulie automatically adds ETag and Last-modified headers in all the responses it generates.
- Automatic action filtering based on the model description. For example, an order "cancel" operation (modeled as an HTTP DELETE) will be automatically rejected if the order is in any other state apart from "payment expected".

5.5 Building Consumers with Restfulie

In addition to enabling the implementation of hypermedia services, Restfulie also provides support for building consumers of such services. Restfulie's primitives expose to developers a hypermedia-based model that allows a client to interact with RESTful services.

A consumer application will initiate the interaction with a service through a well-known URI. As per the HATEOAS principles discussed in this paper, the hypermedia-enabled resource representations received will contain enough information for the client to make forward progress in the business interaction. Restfulie's primitives enable developers to interact directly with these hypermedia controls.

5.5.1 Entry point

Entry points are typically accessed through a GET request, which retrieves a list of resources that can be acted upon, or a POST/PUT, which creates an initial resource. Figure 12 shows an example of the latter, where an HTTP POST request with '`application/vnd.restbucks+xml`' content is sent to the Restbucks ordering service.

```
def create_order(what = "latte")
Restfulie.at('http://restbucks.com/orders').
          as('application/vnd.restbucks+xml').
          create(new_order(what))
end
```

Figure 9 Sending a POST request to the Restbucks ordering service

5.5.2 Making forward progress

Restfulie extracts the hypermedia controls from the received responses and exposes them to developers, who can then programmatically process the links and associated relations. Activating well-understood hypermedia link relations will result in the appropriate HTTP request being sent to the service. For example, activating the link associated with the well-known "delete" relation will cause an HTTP DELETE request. A service's contract will define the appropriate HTTP verb that should be used for domain-specific relations. For example, when creating a payment resource through a "payment" relation, one needs to send a POST request with the appropriate content type, as Figure 10 illustrates.

```
order.request.as('application/vnd.restbucks+xml').
              pay(payment(order.cost), :method => :post)
```

Figure 10 Following hypermedia links

6. CONCLUSION

By embracing hypermedia, we realistically begin to expose business protocols over the Web. This milestone is important because of the significant benefits, in terms of loose coupling, self-description, scalability, and maintainability, conferred by the constraints of the REST architectural style.

All of this comes at rather a modest design cost when compared to non-hypermedia services. This is encouraging, since it means the effort required to build and support a robust hypermedia service over its lifetime is comparable to that associated with building services that share metadata out of band using URI templates or WADL. It's certainly a better proposition than tunneling through the Web using POX.

Our experimentation with Restfulie (and other contemporary frameworks) has delivered existence proofs that such an approach is practical. We strongly encourage others to experiment with the approach.

7. REFERENCES

[1] **W3C.** Architecture of the World Wide Web, Volume One. *W3C*. [Online] December 15, 2004. http://www.w3.org/TR/webarch/.

[2] **Fielding, Roy.** *Architectural Styles and the Design of Network-based Software Architectures (PhD Thesis)*. s.l. : University of Irvine, California, 2000.

[3] **IETF.** Atom Syndication Format. [Online] http://tools.ietf.org/html/rfc4287.

[4] **W3C.** Web Application Description Language. [Online] August 31, 2009. http://www.w3.org/Submission/wadl/.

[5] **IETF.** The Atom Publishing Protocol. [Online] October 2007. http://tools.ietf.org/html/rfc5023.

[6] **Restfulie.** http://restfulie.caelum.com.br/

[7] **Jersey**. https://jersey.dev.java.net/

[8] **RESTEasy**. http://jboss.org/resteasy/

Using HTTP Link: Header for Gateway Cache Invalidation

Mike Kelly
Independent Researcher
Jersey, Channel Islands
mike@mykanjo.co.uk

Michael Hausenblas
Digital Enterprise Research Institute (DERI)
National University of Ireland, Galway
IDA Business Park, Lower Dangan, Ireland
michael.hausenblas@deri.org

ABSTRACT

Gateway caches are intermediary components for reducing demands on destination servers, and therefore operational costs of a system. At scale, particularly with the advent of on-demand infrastructures such as EC2, etc., maximising cache efficiency translates into cost efficiency, resulting in a competitive advantage. In this position paper, we initially discuss advantages and limitations of HTTP caching mechanisms. We then propose to use HTTP `Link:` headers to maximise the efficiency of gateway (or reverse proxy) caching mechanisms and discuss early findings.

1. INTRODUCTION

Caches are essential components of the Web—as a RESTful ecosystem [3]—both concerning *scalability* and to compensate the *stateless* aspect of its architecture [5]. A gateway (or reverse proxy) is an intermediary component through which requests and responses flow. Gateway caches are types of gateways that implements a mechanism capable of managing requests and responses in such a way that it is able to serve responses to client requests on behalf of a destination server.

With the advent of on-demand infrastructures such as Amazon's EC2[1], maximising gateway cache efficiency translates into cost efficiency, resulting in a competitive advantage.

In this position paper we review basic (HTTP built-in) mechanisms in Section 2, and describe how to use HTTP `Link:` header to perform cache invalidation in Section 3. Eventually, in Section 4, we conclude our current work.

2. BASIC MECHANISMS

HTTP provides standardised mechanisms with which clients, servers, and gateways can negotiate and control cached representations. These existing mechanisms however are either sub-optimal in terms of system processing and governing efficiency (i.e. expiration and validation based caching), or are not capable enough to be relied on exclusively (i.e. cache invalidation). This proposal aims to augment HTTP's cache invalidation mechanism so as to increase its expressive capability and become a caching strategy that is feasible to leverage exclusively.

[1] `http://aws.amazon.com/ec2/`

WS-REST 2010, April 26 2010, Raleigh, NC, USA

Other, related discussions have been taken place in the REST community[2] and the microformats community[3].

2.1 Expiration-based caching

Part 6 [2] of the upcoming HTTPbis specification provides the basic framework and defaults concerning caching. Caches are controlled by including an expiration timing as part of the cacheable response, for example:

```
Cache-Control: public, max-age=600
```

Pros:

- Straight-forward to implement;
- No contact with destination server until expiration

Cons:

- It is *inefficient*; refresh must always be controlled against worst case, causing the destination server to be contacted more than necessary;
- One needs to manage caching rules; What should the rules be? Where are rules stored? How to deal with rules that change?

2.2 Validation-based caching

Also commonly referred to as the *conditional-GET* [2]. It is an approach in which requests include a validate-able value from a previous response which a destination server can use to establish whether the client possess a fresh or stale object. If client's object is still fresh server can respond with empty `304 Not Modified` response. The validate-able values are:

`Last-Modified: (datetime)`
and
`If-None-Match: (etag)`

Gateway caches can use this technique in order to ensure freshness of the cached objects it is serving to clients.

Pros:

- Ensures freshness, and
- the `304` responses are small.

Cons:

- Contacting the server for each request, which must be processed and evaluated;
- Not available in every environment and not always reliable [1].

[2] `http://www.mnot.net/blog/2006/02/18/invalidation`
[3] `http://microformats.org/discuss/mail/microformats-rest/2006-March/000187.html`

2.3 Combining Expiration and Validation

Both of the above methods may be employed together so that a gateway cache will simply assume an object is fresh for set period, after which the object must be revalidated.

Pros:

- Expiration can be used to reduce validation checks.

Cons:

- Still suffering from "worst case" inefficiency, however with a `304` Not Modified response, the cache can update its caching metadata using any Cache-Control metadata in the response, thereby mitigating some of the inefficiency;
- One still has to deal with the management of caching rules.

2.4 Cache Invalidation

Section 2.5 of Part 6 of the upcoming HTTPbis specification [2] defines *Request Methods that Invalidate*, listing `PUT`, `DELETE` and `POST` as HTTP methods that must cause a cache to invalidate the request URI.

Typically, cache invalidation would happen in the way as shown in Fig. 1.

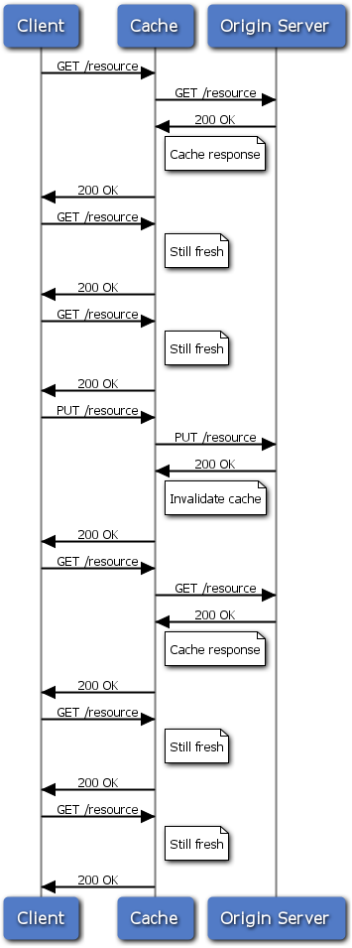

Figure 1: Example invalidation sequence.

The *Web Cache Invalidation Protocol* (WCIP) [6]), proposes to use invalidation and updates to keep changing objects up to date in Web caches, hence enabling proxy caching and content distribution of large amounts of frequently changing Web objects.

We like to stress that this is typically used as a bonus mechanism on top of Validation+Expiration, rather than the primary mechanism. Empirical evidence suggests that this is due to two specific problems revolving around pragmatism and trade-offs in resource granularity and identification: (i) the *composite problem*, and (ii) the *split-resource problem*.

2.4.1 *The composite problem*

Exposing a collection resource as a list of links requires several subsequent requests to render each item, considered as too "chatty" for a common requirement.

If one "overlaps" item and collection resources by embedding items within collections, it would result in collection resource's state containing several dependencies spread across all of the items. This breaks the power of the uniform interface regarding cache invalidation, as shown in the following:

```
> GET /composite-collection
< 304 Not Modified
> PUT /composite-collection/item
< 200 OK
> GET /comsposite-collection
< 304 Not Modified
```

2.4.2 *Split-resource problem*

Serving multiple representations from one URI (content negotiation, content-type, vary) makes it impossible to provide a link to a specific representation. Hence, one could propose to assign dedicated URIs for each representation of a resource. This would then allow to explicitly linked to them.

As representations are now also separate resources in their own right—given that URIs are opaque—this creates, however, invisible dependencies and breaks cache invalidation, again:

```
> GET /resource.xml
< 304 Not Modified
> GET /resource.json
< 304 Not Modified
> PUT /resource.xml
< 200 OK
> GET /resource.xml
< 200 OK
> GET /resource.json
> 304 Not Modified
```

2.5 The HTTP Link: Header

Web Linking [7, 4] defines a framework for typed links independently of a particular serialisation or application. The HTTP header-field for conveying typed links was defined in an earlier HTTP version, however—due to a lack of implementation experience—removed from HTTP/1.1 (RFC2616), resulting in an undefined status for the `Link:` header.

The proposed standard [7] now "specifies relation types for Web links, and defines a registry for them ...", for example (from [7]) ...

```
Link: </>; rel="http://example.net/foo"
```

... indicates that the root resource is related to this resource with the extension relation type `http://example.net/foo`.

3. LEVERAGING THE UNIFORM INTERFACE

We propose to address the above discussed issues by leveraging HTTP's uniform interface, its basic cache invalidation principals, and `Link:` headers. The goal is to achieve a more efficient solution for cache invalidation, which we will call *Link Header-based Invalidation of Caches* (LHIC) in the following, which has the following characteristics:

- Maximum efficiency;
- Having items cached for the longest possible period;
- Avoiding to contact the destination server;
- Governing the caching rules.

HTTP `Link:` headers allow us to express resource dependencies, increasing self-descriptiveness of messages, which can be leveraged to enable cache invalidation for the above problems. Establishing standardised link relations and defining the associated invalidation mechanisms for common cases of resource dependency will create necessary platform on which generic tooling can be developed. However, there is no reason this technique could not be leveraged and caching mechanisms implemented in systems immediately.

We have identified two different approaches—labelled LHIC-I and LHIC-II— to managing resource invalidation dependencies, these are discussed in the following.

3.1 LHIC-I

Issue in the success response to an invalidating request, i.e., "pointing out" the affected resources:

```
> PUT /composite-collection/item
< 200 OK
< Link: </composite-collection>;
rel="http://example.org/rels/dependant"
```

3.2 LHIC-II

Express them initially in the cacheable responses of each resource, i.e., a resource with invalidation dependencies defines them in its cacheable response (one could describe this mechanism as "latching" onto another resource):

```
> GET /composite-collection.html
< 200 OK
< Link: </composite-collection>;
rel="http://example.org/rels/dependsOn"

> GET /composite-collection.xml
< 200 OK
< Link: </composite-collection>;
rel="http://example.org/rels/dependsOn"

> GET /composite-collection.json
< 200 OK
< Link: </composite-collection>;
rel="http://example.org/rels/dependsOn"

> PUT /composite-collection
< 200 OK
```

The above should invalidate all of the following:

```
/composite-collection.html
/composite-collection.xml
/composite-collection.json
```

3.3 Comparison

LHIC-I has the benefit of granting greater and more dynamic control of invalidation to the origin server. However, there are drawbacks:

- In order to secure the mechanism from Denial-of-Service attacks, in which any/all cached objects could theoretically indicated for invalidation, it is likely that a same-domain policy would have to be adopted;
- Invalidation of this kind cannot "cascade", that is, any "inherited" dependencies are not visible to this mechanism, since the response will only contain information about "first level"-dependencies.

In contrast, LHIC-II does not suffer the above drawbacks, since cacheable responses control their own dependencies the mechanism eliminates the possibility of DoS attack. Also, "cascading invalidation" is enabled because a cache can query the cached objects to determine the "dependency graph" for a given invalidation. The drawback of LHIC-II is, however, that this mechanism significantly increases complexity/requirements of the cache.

Concluding, the best approach is to implement both methods and ensuring same domain policy for the first method. This grants dynamic control to origin server and retains mechanism for invalidation cascade.

4. CONCLUSION

This approach provides a more efficient mechanism for gateway caching than is currently possible using existing methods. There are, however, some points for consideration. In some situations resource state could be changed from outside of the uniform interface, some solutions to that problem might be:

- Introduce Expiration and Validation again;
- Invalidation will still reduce risk of staleness, allowing more "relaxed" expiration rules, and therefore increasing efficiency;
- Extremely "volatile" resources should be isolated and not cached at all.

Any invalidation logic should be applied only when invalidating requests receive a successful response, i.e., `PUT`, `POST`, `DELETE` requests that receive 2xx/3xx responses. This prevents erroneous or malicious requests unnecessarily invalidating cache.

Currently, we do not consider the implications of the proposed strategies for the overall caching infrastructure, including browser caches and 3rd party intermediary caches; we plan to address this issue in our future work.

Acknowledgment

Our work has partly been supported by the Science Foundation Ireland and the European Commission under Grant No. 231335, FP7/ICT-2007.4.4 iMP project. Eventually, we want to thank the reviewers for their valuable comments.

5. REFERENCES

[1] L. R. Clausen. Concerning Etags and Datestamps. In *Proceedings of the 4th International Web Archiving Workshop*, 2004.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Y. Lafon, M. Nottingham, and J. Reschke. HTTP/1.1, part 6: Caching. Internet Draft, Expires: April 29, 2010, IETF HTTPbis Working Group, 2009.

[3] R. Fielding and R. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[4] E. Hammer-Lahav. Link-based Resource Descriptor Discovery. Internet-Draft, 23 March 2009, IETF Network Working Group, 2009.

[5] I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation 15 December 2004, W3C Technical Architecture Group (TAG), 2004.

[6] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. Internet Draft, March 2001, IETF, 2001.

[7] M. Nottingham. Web Linking. Internet-Draft, January 19, 2010, IETF Network Working Group, 2010.

Replacing Legacy Web Services with RESTful Services

Charles Engelke
Info Tech, Inc.
5700 SW 34th Street
Gainesville, FL 32607
+1 (352) 381-4400
charles.engelke@infotechfl.com

Craig Fitzgerald
Info Tech, Inc.
5700 SW 34th Street
Gainesville, FL 32607
+1 (352) 381-4400
craig.fitzgerald@infotechfl.com

ABSTRACT
In this paper, we describe issues encountered in designing and implementing a set of RESTful services to extend and replace web services that have been in commercial use since 1998. Applicability of REST to the service requirements, suitability of available tools, and interoperability between multiple clients and servers are discussed.

Categories and Subject Descriptors
D.2.12 [**Software Engineering**]: Interoperability.

General Terms
Performance, Design, Reliability, Security, Standardization, Languages, Legal Aspects.

Keywords
REST, web services, digital signatures, cryptography, sealed bidding, interoperability.

1. INTRODUCTION
Info Tech develops software and provides services to the transportation industry. In 1992, Info Tech introduced applications to facilitate electronic data interchange for construction bids. In 1998, this capability was enhanced to use the Internet, digital signatures, and cryptography to completely replace traditional paper bids with Internet sealed bids.

The Internet bidding system was designed around a custom bid submission protocol created specifically for that purpose. The architecture was essentially a set of remote procedure calls tunneled over HTTP POST requests. This protocol has proved to be extremely reliable and successful. Approximately 40,000 bids totaling US $120 billion were submitted via this protocol to over 30 US state departments of transportation in 2009.

Although it has met the core requirements well, the bid submission protocol has not been easy to extend for new requirements or for use in other application domains. A new, more general purpose protocol using the REST[2] architecture was designed and developed. The new protocol is extremely simple in design, adding minimal new rules to normal HTTP verb behaviors. Despite its simplicity, it satisfies the business needs for the very specialized problem domain of signed, sealed bidding extremely well.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

2. FUNCTIONAL OVERVIEW
2.1 Existing Services
The original service consists of two web services, with each providing multiple functions. One web service provides all functions bidders use: submit a bid, list a requester's submitted bids, and withdraw a bid. The other service provides all functions an agency that receives bids requires: list all submitted bids, fetch public keys for signature verification, and fetch all submitted bids.

Each web service operates in a similar manner. Requests are made via an HTTP POST request whose body has the Content-Type multipart/form-data. One form element specifies the operation requested and others contain necessary request data. Even in the case of application errors, the web services' responses always have status code 200 OK unless there is a system error outside the control of the application itself. The body of the response is plain text and consists of named sections with headers, formatted similarly to a Windows .INI file. The content of this body includes the actual success or failure of the request in one section, and requested information in other sections.

No user authentication is used for the requests themselves. Instead, the request body is authenticated with a digital signature. If a request to submit a bid for "Acme Roads" is digitally signed by someone authorized to act on behalf of "Acme Roads", it is accepted, otherwise it is rejected. For that reason, requests can be replayed. The application services are designed to detect and reject such replays.

2.2 Existing Service Issues
The web services are partitioned based on kinds of requesters, not kinds of requests. Among other potential design issues this can cause it has resulted in similar operations (such as listing bids, either for the individual bidder or all submitted bids for an agency) being segregated into separate services.

The services themselves take little advantage of HTTP's architecture, which would be a good fit to the problem space. Instead, authentication, error reporting, and data typing were all designed from scratch for the application. And since the service protocol is unique to the application, interoperability with other clients and services is more difficult than it might otherwise be.

However, the existing services are stable, reliable, and proven over more than a decade of use. Hundreds of thousands of bids worth half a trillion dollars have been handled by the services with no bid ever lost or altered. As long as the functional and operational requirements remained unchanged, there was no reason to change the services.

2.3 New Functional Requirements

Supporting more state transportation agencies, different kinds of agencies, and different business areas has led to desires for new behaviors in the system. These in turn have affected the functional requirements.

- Cryptographic protocol. The existing system uses the OpenPGP protocols for digital signing and sealing of bids. Some state governments have implemented their own PKI systems, and would like to use them for digital signing. This requires changing to the PKCS cryptographic protocols.
- Bids submitted in multiple parts. A single bid must now be submitted in a single transaction. Some agencies wish to add significant data to the bid, making it large enough to take several minutes or more to submit. Since bid submission deadlines are firm, and activity as the deadline approaches is under considerable pressure, it would be advantageous to allow a bidder to submit a bid in pieces. The more stable parts could be submitted hours or even days before the deadline, making the submission near the deadline smaller.
- New types of clients. There is currently a single supported client, which is a Windows program the user must download and install. There is considerable interest in a web-browser based client. Info Tech has developed such a client but it cannot operate with the legacy protocol[1]. (A U.S. patent is pending on this kind of client.)
- Managing data other than bids. The same firms that submit bids to transportation agencies often have to send other kinds of digital data. This data often must be signed, and sometimes needs to be sealed until it reaches the agency itself. Unlike bids, there is rarely a rigid deadline after which submission is disallowed, but in other respects this is similar to sealed bidding, and it be would valuable for the bid submission services to also support these kinds of operations.

3. NEW SERVICE DESIGN

The new suite of web services has been made more general than the original ones. The suite and the protocol that defines it are code-named Elephant. Any service that implements the Elephant protocol allows clients to transfer information between each other in a highly secure manner. One possible application of this protocol is for bid submission.

3.1 Bid Submission Use Case

A transportation agency publicly announces requests for bids. Bidders prepare their bids and deliver them to the agency in a sealed envelope prior to a published deadline. Shortly after the deadline, the agency opens the envelopes in public and reads and processes the bids.

3.2 Elephant Protocol Overview

The new protocol permits resources, such as bids, to be submitted to an Elephant server where they will be held safely until the deadline for opening bids has passed. At that time, the agency requesting bids can fetch the submitted bids, open and process them.

An empty collection of resources can be established on an Elephant server via an HTTP POST or PUT request. Metadata included in HTTP headers of the creation request specify who can add resources to the collection, the deadline for adding, changing, or deleting resources, and rules for digitally signing and cryptographically sealing all resources.

Resources are submitted to a collection with an HTTP POST or PUT request. Metadata with information about the resource that has meaning to the users of the service (as opposed to the Elephant server itself) can be specified through HTTP headers on the POST or PUT request.

Resources in a collection are enumerated by making an HTTP GET request to the collection, and individual resources in the collection are retrieved via HTTP GET requests to the URIs that were listed in the enumeration.

The Elephant server not only accepts, holds, and delivers resources, it optionally protects their integrity by only allowing submissions, changes or deletions prior to the deadline, withholding access to resources from the owner of the collection until after the deadline, and refusing to accept resources that aren't properly signed and encrypted as specified by the collection owner.

3.3 Elephant Applicability to Bid Submission

A transportation agency can use Elephant for Internet bidding by establishing a resource collection to hold submitted bids, and advertising that collection's URI when publishing a request for bids. The collection would be created with a deadline for new additions to it that matches the bid submission deadline, and with rules specifying how bids submitted to the collection must be digitally signed and cryptographically sealed.

Bidders use software programs to create and submit their bids. To submit the bid over the Internet, the bidding software would package the bid as a resource and submit it to the Elephant server to be added to the collection at the advertised URI. The server would accept it only prior to the established deadline for that collection, and only if it is properly signed and sealed according to the rules established when the collection was created.

The agency eventually retrieves and processes the bids by issuing a request to the Elephant server for all the bids in the collection at the published URI. Again, the Elephant server will only satisfy this request if the deadline allows it. In this case, the deadline must have already passed in order for the agency to retrieve the bids.

Use of the deadline property is fundamental in maintaining the integrity of "sealed" bids. This ensures that the agency cannot see bids until they are to be made public to all. The cryptographic sealing rules further protect the bid even from disclosure to whomever controls the Elephant server by specifying that all submitted bids must be encrypted to a certificate whose matching key is known only to the transportation agency.

3.4 Resource Classes

The Elephant protocol has been designed in a Resource-Oriented RESTful architecture[6]. The resources themselves are generally opaque to the protocol and any Elephant server. The clients that

interact with the service see meaning in the resources and their representations, but Elephant simply faithfully accepts, stores, and returns them unchanged. In the Elephant protocol, these fundamental resources are called **Things**.

Elephant also supports resources that are collections of Things. Such a resource was central to the Bid Submission Use Case described above. Collections of Things are called **Places** in Elephant.

Places and Things are the only two kinds of resources needed for the expected use cases, but a third resource type, a collection of Places, is also supported. This resource is called a **Space**. Spaces are used simply to allow different organizations to manage Places without dealing with naming collisions with each other.

Every resource in the system has a canonical URI that never changes. For convenience, these URIs are organized hierarchically. For example, the URI of a Place will consist of the URI of its containing Space, followed by a suffix. This naming convention is natural and easy to use, but is not central to the design of the service.

3.5 Operations

All Elephant operations are mapped directly to HTTP verbs. A client designer who understands HTTP verbs well will find the Elephant protocol easy to understand and use[3].

HTTP GET is used to retrieve a resource, HTTP POST creates a resource in a specified collection, HTTP PUT creates or updates a specified resource, HTTP DELETE deletes a resource, and HTTP HEAD retrieves information about a resource rather than the resource itself. No other HTTP verbs are used in the protocol.

The resource itself is placed in the body of the request or response, and information about the resource is put in request or response headers. HTTP headers are either standard HTTP headers or Elephant-specific headers that have meaning only to clients and servers that understand the Elephant protocol. Elephant-specific header names will be prefixed by X-Elephant-.

Functional behavior follows expected behavior of the HTTP verbs, with additional restrictions imposed by the Elephant protocol. The most important of those restrictions concern who can perform which operations, and when.

3.6 Functional Behavior

Every operation that creates a resource on an Elephant server must be authenticated. The authenticated user becomes the owner of the created resource. The service itself has a built-in user authentication mechanism and others can be plugged in, and the creator of a collection resource can specify (through Elephant headers when the collection is created) what authentication mechanism should be used for operations on resources within that collection. For example, when a transportation agency creates a Place for bids to be submitted, the agency can specify that operations on resources within that Place be authenticated against a particular LDAP server.

Authorization for operations depends on whether an authenticated user created the affected resource or the collection the resource belongs to. No finer control of access is currently supported. The desired use cases can be satisfied with this coarse mechanism, allowing the protocol to remain simple and easy to use.

Every Place (collection) must have a property called **Policy** associated with it that defines the rules for who can create, modify or access resources within the collection. The two most important policies are Publish and Deposit. The Publish policy is intended to allow a collection owner to make selected resources generally accessible. Only the owner of a collection with the Publish policy can create resources in it, but anyone can access those resources. The Deposit policy is the one that enables most of the use cases. Any authenticated user can create a resource in a collection with the Deposit policy, but only the owner of the created resource or the owner of the collection itself can fetch the resource.

Collections with Deposit policy can have time related restrictions imposed when they are created. A deadline can be specified o limit the latest time a resource can be added to or updated in the collection and the earliest time the collection owner can access the elements of the collection.

Deposit collections can also have cryptographic rules imposed on resources created within them. One optional rule can require that all resources created or updated in the collection must be digitally signed; the rule can also specify one or more root certificates that must have issued the certificate used to digitally sign the resource. Another rule can require that all resources be encrypted to a set of recipients with specified certificates, prior to being sent. The cryptographic rules all require use of specific resource representations and formats according to PKCS standards[4,5].

These restrictions specifically enable digitally signed, sealed bidding, but are more generally useful as well. For example, sensitive information such as social security numbers can be passed from contractors to transportation agencies without any possibility of disclosure to the intermediate Elephant server. Contractual documents can be forced to be digitally signed before they are accepted for delivery to other parties.

3.7 Example: Creating a Collection for Receiving Bids

An agency can create a bid deposit Place with an HTTP PUT request. The agency must have access to an Elephant Space. If the Space's URI is http://host.name/SpaceName, it can create a Place within that Space named ProjectXYZBids with a single HTTP PUT, such as the one below:

```
PUT /SpaceName/ProjectXYZBids HTTP/1.1
Host: host.name
Authorization: Basic dXNlcjpwYXNz
X-Elephant-Policy: Deposit
X-Elephant-Deadline: 2010-12-25T15:00:00Z
X-Elephant-Encryption-Certs:
   http://host.name/SpaceName/Certs/bidofc.pem
X-Elephant-Signature-Certs:
   http://host.name/SpaceName/Certs/root.pem
X-Elephant-Authentication-Scheme:
   LDAP server=some.domain search="dc=bidder"
```

This request will establish a bid deposit location with a specified deadline, and only accept submissions that are digitally signed with a certificate from a certificate authority with the specified root certificate. Submissions should also be encrypted so that only the owner of the "bidofc.pem" certificate can decrypt them.

Any request that creates or modifies a resource, or accesses one that is not in a Publish Place, must be authenticated. Since this request creates a Place resource, it requires authentication,

provided via the Authorization header. The credentials provided are verified according to the rules established for the Space containing this new Place. The new Place will have Elephant property Authentication-Scheme set for it, so requests involving Things in this new Place will be verified according to the rules given by that property. In this example, that means that the credentials will be verified against a particular LDAP search.

4. DEVELOPMENT EXPERIENCE

Because the existing protocols have been found to be difficult to support on some platforms or in some languages, especially within web browsers using JavaScript, multiple Elephant servers and clients were developed. Every combination of client and server was checked to make sure that the design of the protocol wasn't subtly biased in favor of or against any expected targeted platform.

4.1 Platforms Supported

Servers were developed using Ruby and the Ruby on Rails web framework with the Mongrel web server on both Linux and Microsoft Windows, and in Perl using the CGI framework with the Apache web server on both Linux and Microsoft Windows. Multiple clients were created:

- Windows Forms program written in C#, running under the .NET common language runtime, and using the .NET managed cryptographic libraries.
- JavaScript program running under Internet Explorer 7 and Internet Explorer 8 using the Windows CryptoAPI libraries accessed through the CAPICOM ActiveX control.
- JavaScript program running under Firefox and Google Chrome on Windows, not supporting cryptographic rules.
- Command line program written in Perl using OpenSSL cryptographic libraries, running under Linux and Microsoft Windows.
- Command line program written in Ruby using OpenSSL cryptographic libraries, running under Microsoft Windows.
- Manual use of the open source Curl and OpenSSL command line utilities, with no additional software layers, in both Linux and Microsoft Windows.

Each combination operated properly. A preliminary version of an Elephant server was also successfully created using Python on the Google AppEngine framework. This early version worked properly with a variety of test clients, but time and resource constraints prevented updating it to the final version of the protocol and testing it with the full set of developed clients.

4.2 Technology Issues

The JavaScript clients were restricted in their support of the cryptographic rules because there are no standard JavaScript libraries to support them. Internet Explorer allows JavaScript to access functions via ActiveX objects, so the JavaScript client in that environment was able to implement the full set of Elephant operations. Clients in Mozilla Firefox and Google Chrome implemented the entire protocol except for the cryptographic operations.

Ruby on Rails and Perl CGI under Apache server frameworks had issues in allowing Elephant server software to deal with HTTP requests other than GET and POST. They also tended to add or modify headers of both requests and responses. In each case, the high level functionality provided by the framework had to be bypassed in order to operate on the raw request to make behavior exactly conform to the Elephant protocol specification. Newer versions of the frameworks released since this project was originally developed have greatly improved in this respect.

5. LESSONS LEARNED

The functional requirements of bid submission have turned out to be extremely well served by a Resource-Oriented RESTful architecture. The Elephant protocol is easy to implement on both the client and server sides, and has shown itself to be technology neutral.

Generalizing the functional requirements when defining the service has resulted not only in a good facility for bid submission, but also many other common requirements in our targeted industry. Elephant has been used to support signed delivery of confidential contractor employee information to transportation agencies, and provide shared "file cabinets" of documentation on construction materials. The small set of rules in the Elephant protocol are rich enough to enable many highly useful applications.

6. ACKNOWLEDGMENTS

The original electronic bidding system mentioned above was developed jointly by Info Tech, Inc., the Georgia Department of Transportation, and the American Association of State Highway and Transportation Officials. The existing Internet bidding protocol is owned by Info Tech, Inc. It was originally developed for, and with the assistance of, the Georgia Department of Transportation. Their help is greatly appreciated. It not only aided the creation of the software itself, but resulted in tools that have had an enormous economic impact on transportation construction bidding in the United States.

7. REFERENCES

[1] Engelke, C. E., Fitzgerald, C. L. and Orduz, R. D. 2008. *System and Method of Electronic Information Delivery.* US Patent Application number 12/148357.

[2] Fielding, R. T. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

[3] Gourley, D., Totty, B., Sayer, M., Aggarwal A., et al. 2002. *HTTP: The Definitive Guide*. O'Reilly Media, Sebastopol, CA.

[4] Housley, R. 2004. Cryptographic Message Syntax (CMS). RFC 3852. The Internet Society.

[5] Jonsson, J. and Kaliski, B. 2003. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447. The Internet Society.

[6] Richardson, L. and Ruby, S. 2007. *RESTful Web Services*. O'Reilly Media, Sebastopol, CA.

Towards a Practical Model to Facilitate Reasoning about REST Extensions and Reuse

Federico Fernandez
Pontificia Universidad Catolica
Vicuña Mackenna 4860
Santiago, Chile
(562) 4545885
fmfernandez@gmail.com

Jaime Navón
Pontificia Universidad Catolica
Vicuña Mackenna 4860
Santiago, Chile
(56-2) 354 4440
jnavon@ing.puc.cl

ABSTRACT

We believe that there is a need for a practical model to visualize the structure and design rationale of REST, so researchers can study more easily the reutilization of this architectural style or parts of it, to the design of software solutions with different requirements than those of the early WWW.

In this work we propose the utilization of extended influence diagrams to represent the structure and design rationale of an architectural style. The model is evaluated qualitatively by showing how a diagram of REST, populated with information extracted from the doctoral dissertation that introduced the term, is helpful to gain a better understanding of the properties and limitations of this style, and to reason about potential modifications for applications with different goals than those of the early WWW.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – *methodologies, representation.*

General Terms

Documentation, Design, Standardization, Theory.

Keywords

REST, architectural styles, software architecture, representational state transfer, web service, restful, software design, network-based software, influence diagrams, extended influence diagrams.

1. INTRODUCTION

Due to the great success of the WWW architecture not only in scalability, but also in simplicity and flexibility, there is increasing interest in understanding and applying the principles behind the WWW architecture to solve problems of other domains.

One source considered authoritative about these principles is the doctoral dissertation of Dr. Roy Fielding ("Author" for the rest of this paper), who contributed remarkably to the protocols that constitute the WWW architecture. The dissertation was titled "Architectural Styles and the Design of Network-Based Software Architectures" [9] ("Dissertation" for the rest of this paper), and is about designing network-based software architectures through the utilization of an abstraction called "architectural style".

In Chapter 5 of the Dissertation, the Author introduces REST (Representational State Transfer) as the architectural style designed to satisfy the requirements of the World Wide Web.

During the last decades, researchers and practitioners have been studying how to apply REST to solve different problems than those of the early WWW. For example, there are publications about the applicability of REST to build better web applications [23] and as an alternative to Web Services for the realization of SOA in the enterprise [22]. Debates have aroused about the advantages and disadvantages of using REST over other approaches to design network-based software. For example, we have seen debates about REST vs SOAP [20], REST vs Web Services [22], and Resource Orientation vs. Service Orientation [21].

However, in many papers the definitions introduced in the Dissertation are not used correctly. For example, REST is not compared to other architectural styles, but to specific technologies or protocols. The Author created a blog where he clarifies confusions about the term and encourages practitioners and researchers to use it properly [5][6][7]. Unfortunately, to do that, researches would need: 1) New names to describe the many specific software architectures built on top of the REST architectural style and 2) A description of alternative software design approaches as architectural styles (i.e. a mapping between the abstractions used by the researchers in their work and the ones contained in REST). Both requirements are hard to satisfy today, because a practical model to describe architectural styles is missing. The models available to describe a network-based architectural style are too ambiguous or too detailed to be used in practice.

In this work we propose the utilization of extended influence diagrams to visualize the structure and design rationale of an architectural style, so researchers can study more easily the reutilization of one architectural style or parts of it, to the design of new architectural styles for problems with different requirements than the original style was built for.

We will show how the extended influence diagrams, together with a visual modeling tool are useful to visualize REST structure and design rationale, and to study potential extensions and reusability of this style to other problem domains.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

2. ANALYSIS OF CURRENT REST MODELS

2.1 Models of the REST Design Rationale

In the Dissertation, the Author describes the REST design rationale using: a REST derivation tree (Figure 1), a matrix of styles vs. architectural properties (Table 1), and text. These can be seen as three models of the REST design rationale (Figure 5).

In order to understand the REST design rationale, one has to study the aforementioned models, and it is very hard to reason about modifications or applications of the style to other contexts. More importantly, it is cumbersome to communicate any change to an audience.

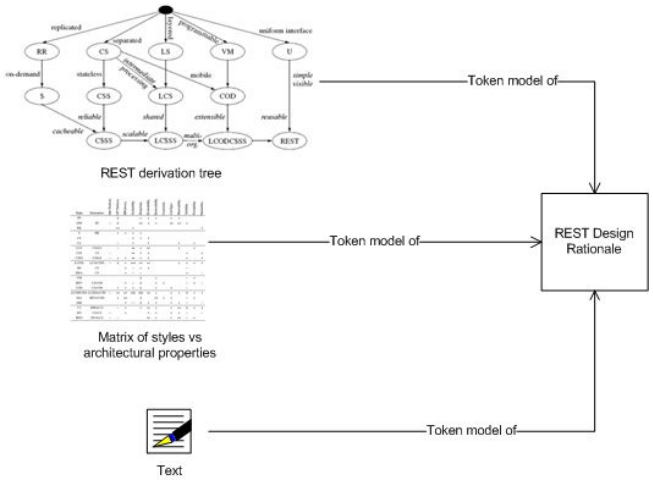

Figure 1: Current models of REST design rationale

Table 1: Matrix of architectural styles vs. software qualities

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF			±			+	+	+		+	+			
UPF	PF	–	±			++	+	+		++	++	+		
RR			++		+									+
$	RR		+	+	+	+								
CS					+	+	+							
LS			–		+		+				+		+	
LCS	CS+LS		–		++	+	++				+		+	
CSS	CS	–			++	+	+					+		+
C$SS	CSS+$	–	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	–	±	+	+++	++	++				+	+	+	+
RS	CS			+	–	+	+					–		
RDA	CS			+	–	–						+		–
VM						±		+				–	+	
REV	CS+VM			+	–	±		+	+			–	+	–
COD	CS+VM		+	+	+	±		+		+		–		
LCODC$SS	LC$SS+COD	–	++	++	+±+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		–	+	
EBI				+	– –	±	+	+		+	+	–		–
C2	EBI+LCS		–	+		+	++	+		+	++	±	+	±
DO	CS+CS	–		+			+	+		+	+	–		–
BDO	DO+LCS	–	–				++	+		+	++	–	+	

What we would like to have is one unified model of the REST design rationale that we could manipulate to visualize changes in a concise manner. We will describe the requirements of such a model in the next section, and we will explain why we decided to use something called "extended influence diagrams".

Figure 2 shows the REST influence diagram in context.

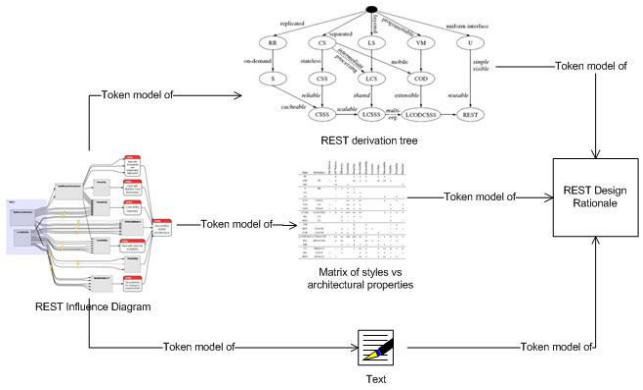

Figure 2: Our proposed model of REST design rationale

2.2 Models of REST Structure

To describe REST (not its design rationale), the Author defines the architectural elements (instances of components, connectors and data) present in REST, and uses a process view to show some possible configurations of these elements (Figure 3).

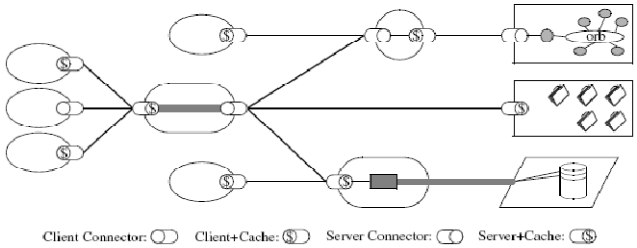

Figure 3: Process view of REST

The model is useful to show some sample configurations of the architectural elements that are defined for REST (client, server, proxy, cache, tunnel, resolver, resource, representation, etc), but it is not very useful to reason about making modifications to REST.

REST is better understood as a coordinated set of constraints that has to be respected by any architecture to be called "RESTful". However, there is no concrete model for this intuitive notion, which is what we would like to have to visualize the impact of adding or removing constraints to REST (Figure 4).

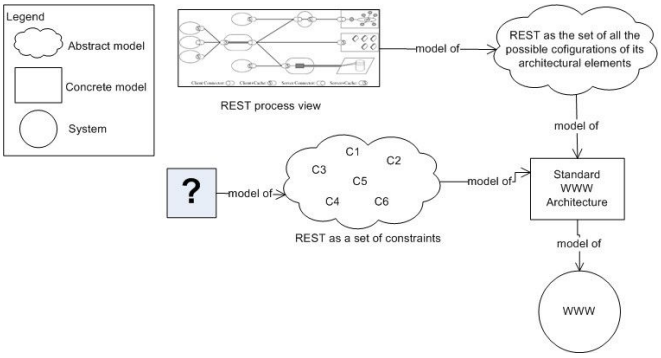

Figure 4: Current Models of REST Structure

We will propose a new model of REST structure in the next section.

3. TOWARDS A PRACTICAL MODEL OF REST

In this section we explain the models we propose to describe the structure and design rationale of REST. The goals of these models are to facilitate the understanding of REST and the reasoning about its potential reuse for applications with different requirements than those of the original WWW.

3.1 Defining the Structure of REST as a Sequence of Architectural Decisions

We will describe REST by using the concept of *architectural decision*. We will define it in the following manner:

An architectural decision is *a named set of constraints that can be added to an architectural style. The result of adding an architectural decision to an architectural style is another architectural style*.

A corollary of this definition is that a given architectural decision can only be made over certain architectural styles. For example, we could say that the architectural decision called: "Stateless CS Interactions" only makes sense if applied over the Client-Server architectural style. What we win with the introduction of this concept is that we can now describe an architectural style as a sequence of architectural decisions. Figure 5 shows this model of REST in context.

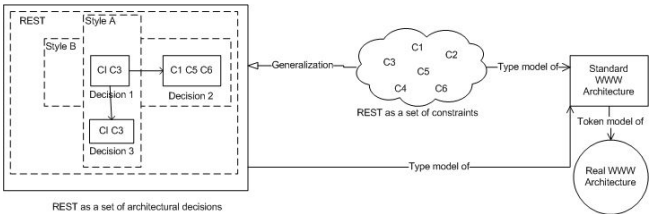

Figure 5: Our proposed model of REST structure

The difference with the "derived style" approach introduced in the Dissertation is that we make every component of an architectural style explicit. Therefore, the properties of the architectural style become the cumulative properties of its individual architectural decisions.

The components of any architectural style described in this way are visible, which is a property required by any "practical" model of the REST design rationale.

3.2 Requirements of a Practical Model of the REST Design Rationale

As it was explained before, we would like to have a unified model of the REST design rationale that we could manipulate to visualize changes in a concise manner. More specifically, the user should be able to:

a) Visualize and understand how each one of the architectural decisions of REST impacts the set of goals that guided its design (R1).

b) Visualize and understand the changes caused in the induced properties if new architectural decisions are added to REST, or if existing decisions are replaced for others (R2).

c) Visualize the set of alternative architectural decisions for each decision in REST (R3).

d) Easily modify the REST model to visualize and understand how each one of the architectural decisions of REST would impact a different set of goals (R4).

In addition, there is one non functional requirement that has to be met:

e) The model should be a loyal representation of the Dissertation in order to be accepted by the stakeholders (R5).

3.3 Candidate Solutions

We explored several alternatives before deciding to go with the solution presented in the next section. First, we considered extending the classification framework presented in Chapter 3 of the Dissertation, by adding rows in the top that would group the different properties into broader categories, as a hierarchy of desired properties. This approach would satisfy the visualization part of R1 and R2, and maybe R4, but it would not improve understandability, since the user would have to read through the Dissertation to understand the meaning of the plus and minus signs.

Secondly, we considered representing REST as a set of documents, one per architectural decision, containing a description of the decision and the alternatives considered [14] but we discarded this approach because it wouldn't have satisfied R1, R2 and R4.

Finally, we considered to use graphical models. After some research, we decided to use a subset of the extended influence diagrams defined in [16]. For our purposes it was enough to use the utility, chance and decision nodes, and just one type of arrow that would mean different things depending on the type of the connected nodes. As we are not trying to make a model capable of probabilistic reasoning, the resulting graphical notation is very close to the one explained in [4] for software development based on non-functional requirements.

The visibility part of R1 and R2 would be met using collapsible graphical elements. The understandability part would be met by adding as many kinds of boxes as necessary to trace the impact of

a decision over the set of goals. R4 would be satisfied by drawing hierarchies of goals from the most general ones, to those represented by software qualities. R5 would be satisfied by extracting all the knowledge to define the elements of the diagram and their relationships from the Dissertation, and documenting all these definitions with comments in the diagram. The only requirement that would not be satisfied is R3, but the ability to trace causality from decisions to goals should help the user identify possible alternatives for each decision.

In the next section, we will explain how we built a practical model of REST using extended influence diagrams.

4. ANALYZING REST WITH EXTENDED INFLUENCE DIAGRAMS

In order to make the diagrams, we relied on a flexible visual modeling tool [11]. As it also served as a utility to browse, zoom, collapse and expand parts of the produced diagrams, we decided to adapt the syntax of the extended influence diagrams described in [16] to the types of nodes and arrows available in the modeling tool.

Figure 6 shows the resulting syntax. There are three node types and three relationship types.

The node types are:

a) Decision nodes: represent architectural decisions.

b) Utility nodes: represent the desired goals of the target architectural style. They are impacted by each architectural decision. In the case of the REST influence diagram, the utilities are extracted from Chapter 2 and 4 of the Dissertation.

c) Chance nodes: they describe the chains of causal effects starting from the decisions and ending on the utility nodes.

The three relationship types are causal relationship, definitional relationship and temporal relationship. As the modeling tool has only one kind of arrow, the relationship type is defined by the nodes being connected and the color of the arrow:

a) An arrow connecting a decision with a chance means that the decision has a causal effect over the chance. A black arrow means a positive effect, while a grey arrow means a negative effect.

b) An arrow connecting a chance with a utility means that the chance has a causal effect over the utility. Only a black arrow, meaning a possible effect, is allowed.

c) An arrow connecting two chances means that the first chance has a causal effect over the second one. Only a black arrow, meaning a possible effect, is allowed.

d) An arrow connecting two decisions means that one decision was made before the other.

e) An arrow connecting two utilities means that one utility is a child of the other. The parent utility is comprised of its constituent children.

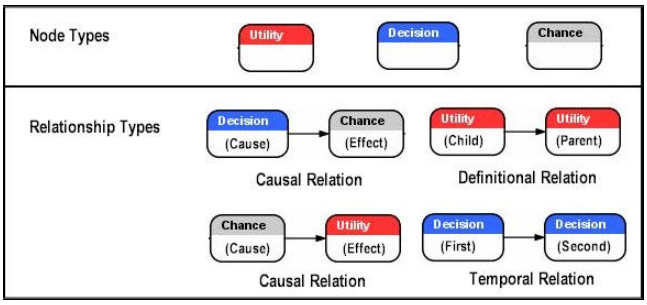

Figure 6: The syntax of the REST influence diagram

In order to improve visibility, we didn't use arrows to connect decision nodes most of the time. Instead, we described the sequence of architectural decisions that defines REST as nested decision sets. This improved visibility at the cost of reduced understandability and expressiveness. The visibility is improved because the arrows going from Decision to chances are easier to draw and follow. Understandability is reduced because if a decision is outside a given set, it means that it was made after the decisions inside the set, but if two or more decisions are in the same set, the user can't know which was done before the other, and has to get this information from another source (e.g. comments). The expressiveness of the diagram is reduced because only some sequences of decisions can be grouped as architectural styles. Figure 7 shows an example of how a group of temporally related architectural decisions would be represented in our extended influence diagram, using the aforementioned modeling tool.

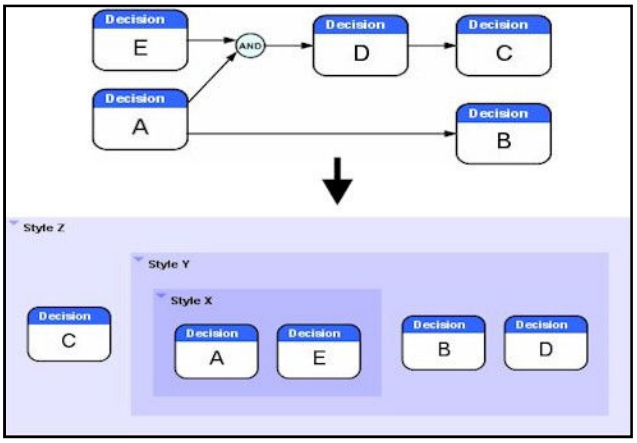

Figure 7: Example of how we draw sequences of architectural decisions

Figure 8 shows a small part of the influence diagram built for REST. The utility nodes were extracted from Chapter 2, and Chapter 4. Each decision represents an architectural style used in the derivation of REST, and each chance corresponds to a reason why a given decision affects a given utility. The chances were extracted from Chapters 2, 3 and 5.

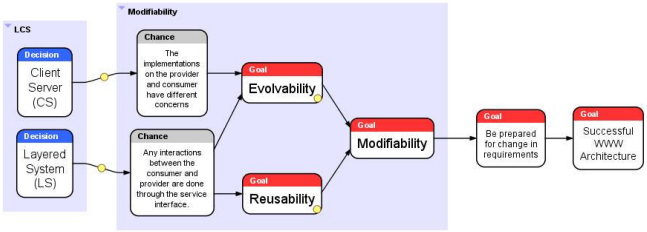

Figure 8: Sample of REST influence diagram

The shaded rectangles in the diagram have the ability to collapse its contents while maintaining the arrows going from the group to the outside. Figure 9 shows the result of collapsing both named rectangles. The little circles that are present in some arrows and boxes denote commentaries. We used them to copy/paste contents from the Dissertation to document nodes and their relationships.

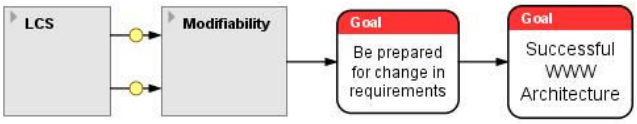

Figure 9: Sample of REST influence diagram, collapsed

Figure 10 shows a condensed view of the REST influence diagram.

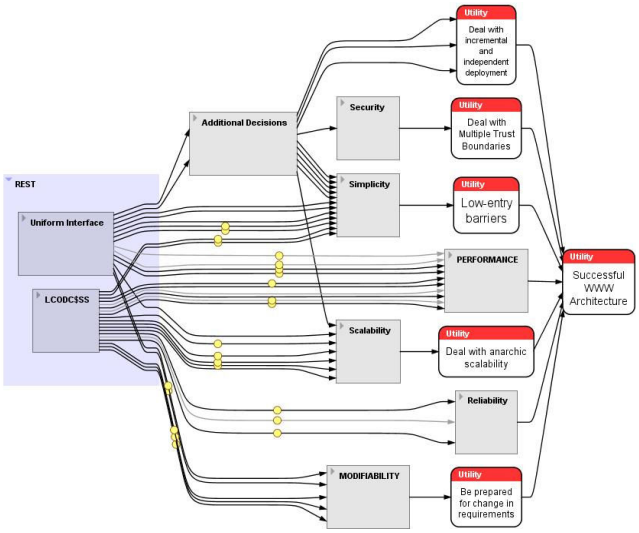

Figure 10: Condensed view of REST influence diagram

In the following paragraphs we comment the process we followed to define the utility, chance and decision nodes of the diagram.

4.1 Utilities

If we deleted all the chances and decisions from the diagram and expanded all the utility nodes, we would get a horizontal version of the tree of Figure 11. The figure shows the hierarchy of utility nodes that we inferred from the Dissertation. These utility nodes were extracted from Chapter 2 and 4. Relating general network based software qualities with the goals of the WWW architecture was not straightforward. Furthermore, while developing the diagram, we decided to eliminate some redundant utility nodes (portability), transform others into chances (visibility), divide some utilities into lower level utility nodes (simplicity), and rearrange the hierarchy of utility nodes (UP performance, Net Performance and Net Efficiency). All these changes were for the sake of making the diagram easier to understand and reuse.

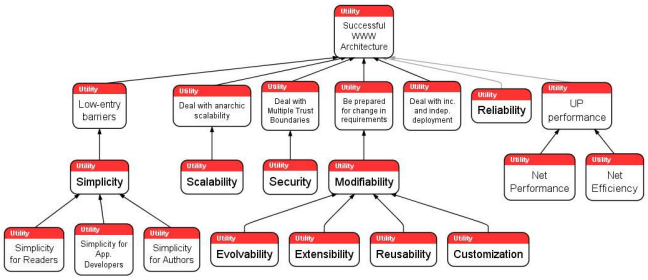

Figure 11: Goals of the standard WWW architecture

The chances were extracted through an exhaustive revision of Chapters 2, 3 and 5. Once we identified the reason why a decision was related to a utility node, we created a chance between both nodes, or reused an already existent one. In order to maximize the reuse of chances, we tried to generalize each chance as much as possible. For example, in Chapter 3 the description of the CSS style includes the following statement: "Scalability is improved because not having to store state between requests allows the server component to quickly free resources and further simplifies implementation." From this statement, we extracted the following chance node: "Servers need to maintain little knowledge about clients," which would later be reused by another decision: "Hypermedia elements are unidirectional".

4.2 Decisions

Initially, every decision node corresponded to one of the styles used in the derivation of REST. However, we soon found out that we had to use finer grained decisions. We then started to divide styles in smaller parts. Figure 12 shows the decisions that compose the Uniform Interface in the REST influence diagram. The decisions that we had to add are: Standard Representations, Standard Operations and Hypermedia as the User Interface.

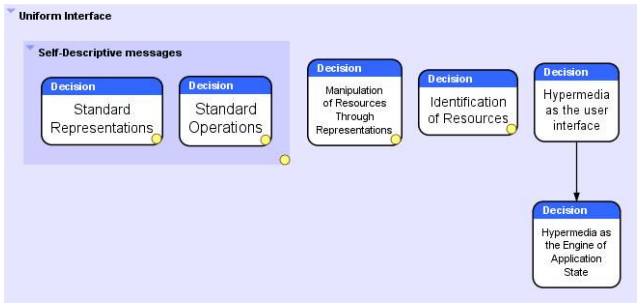

Figure 12: Architectural decisions of the uniform interface

4.3 Decisions Beyond REST

While developing the diagram, we also extracted some decisions and corresponding chances that were not explicitly explained as parts of the REST derivation. Some of these decisions are shown in Figure 13. They can be considered constraints that can be added to REST to characterize a more restricted architectural style. For example, the decision to define an idempotent operation to request a representation of a resource (i.e. GET in HTTP) could be seen as a constraint that can be added to REST to define a new architectural style. Thus, the influence diagram can facilitate the

task of checking what happens if one adds more constraints to REST. All what is needed is to write the constraint as a decision node, find the chances and utilities that it impacts, and reason about its usefulness.

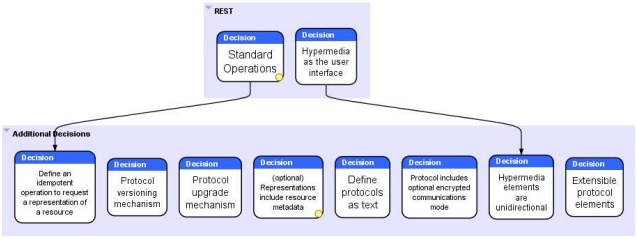

Figure 13: Architectural decisions outside REST

5. EVALUATION

As a preliminary evaluation of the proposed model, in this paper we show 1) how the model facilitates the visualization of some important aspects of the represented architectural style (REST in this case), and 2) show an example of how it can be used to reason about potential modifications to the architectural style being studied.

5.1 Insights about REST obtained from the Extended Influence Diagrams

Some interesting insights about REST can be immediately obtained by simply looking at the condensed view of Figure 10. Others require further inspection of the diagram:

a) Simplicity, Scalability, and Modifiability are the main properties of REST. The goals of Simplicity and Scalability are those that receive more positive effects from REST. They are followed by modifiability and performance, thought performance is also affected negatively by some decisions of REST.

b) Security is not affected by REST. One interesting conclusion is that the goal of Security is not tackled directly by REST. It is only treated in a lower abstraction level, and only by one chance: "Protocol includes optional encrypted communications mode" which in turn affects the sub-utility node: "network-level privacy".

c) Reliability is partially affected by REST. In addition, it is interesting to note that although the property of Reliability has four chances extracted from Chapter 2, only two are positively affected by REST, and both comes from the Client-Stateless-Server style.

d) REST was designed for human users. Thought REST clients can be robots, it was clear while extracting the chances and decisions that the Author was privileging the case of the users being human. This is manifested, for example, in the chances affecting the goal of Simplicity, that we had to divide into: "Simplicity for Authors," "Simplicity for Readers," and "Simplicity for Application Developers," who are the users introduced in chapter 4 of the Dissertation. One practical consequence of this is that, if we would like to make an assessment of how REST induces simplicity in SOA, we should start by classifying the users of SOA and twisting and redistributing the chances related to the sub-utilities of Simplicity in the REST diagram, into the sub-utilities defined for SOA.

5.2 Using the REST Influence Diagram to Identify the Missing Properties of ROA

The Resource Oriented Architecture is defined in [23] as "a way of turning a problem into a RESTful web service: an arrangement of URIs, HTTP, and XML that works like the rest of the Web, and that programmers will enjoy using". Another term used for the same purpose is Web Oriented Architecture, which is used by some IT consultants to name the application of the WWW standard protocols and proven solutions for the construction of distributed software systems [13].

Recently, with the blog posts mentioned before [6][7][8] it became clear that these approaches are not following all of the REST constraints, so their practitioners started to discuss ways to follow all of the REST constraints and to debate how important is to follow all of them.

The REST influence diagram is a good tool to identify what properties are really missing by not following all the constraints. In order to accomplish this task, we took the REST influence diagram and deleted all Decision nodes that are not relevant to the study. These Decisions are those represented by the LC$SS style (Table 1) and by those architectural decisions used for the development of HTTP and URI but not corresponding to the core of REST (the "Additional Decisions" in Figure 13). The resulting diagram is shown in Figure 14.

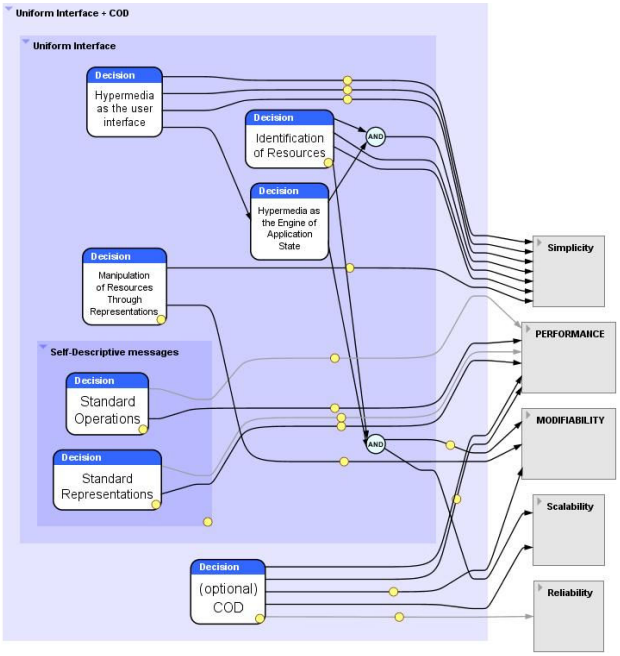

Figure 14: Properties of COD and the uniform interface

ROA lacks only one decision: "Hypermedia as the engine of application state" and places another decision as optional: "Hypermedia as the user interface" (by just recommending that the resource representations should be "connected" by links).

Now, if we focus only in the arrows going from these two decisions we get the diagram of Figure 15. Clearly, the constraint

"Hypermedia as the user interface" was only added to REST as a means to lower the WWW entry barriers for human readers and authors. Thus it is logical that it didn't make much sense for developers thinking about using REST for machine to machine interaction.

On the other hand, "Hypermedia as the engine of application state" affects two chances that would be kept in the diagram even if we instructed the reader to think that the user is now a machine, and these chances impact Scalability and Evolvability. So why did the ROA proponents miss this constraint? Because respecting that constraint in the case of machine to machine integration means that we have to develop representations containing semantic hypermedia, something that would affect negatively the utility of "Simplicity for Application Developers" and "Performance" if the Influence Diagram of REST were made for the goals of machine to machine integration. Thus, it happens that this additional effort is not considered worthwhile when compared to the benefits of Scalability and Modifiability.

Therefore, the answer to the original question is that by not following the hypermedia constraint, the architectures conforming to ROA are less scalable and modifiable than architectures conforming to REST. Exactly how less scalable and modifiable cannot be inferred from the Dissertation. Therefore, the tradeoff of adding these constraints to ROA at the cost of lower simplicity and performance must be the subject of future research.

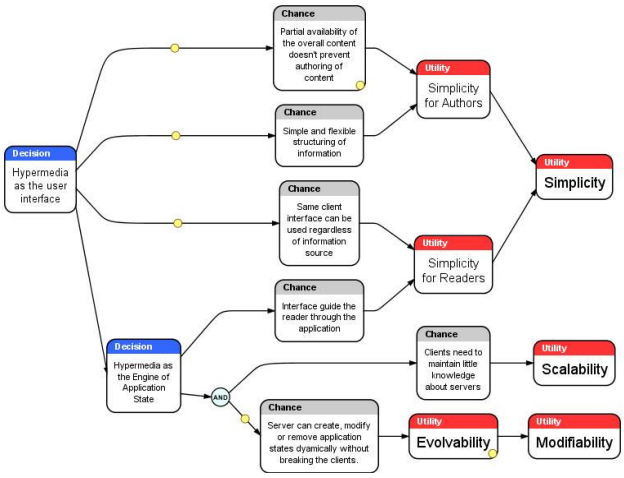

Figure 15: Properties of Hypermedia

6. DISCUSSION

The answers that could be obtained in section 5.2 using the REST influence diagram may not be surprising to REST practitioners, but the use of the influence diagram can guarantee that the response is made considering only the knowledge included in the Dissertation and nothing else. [1]

The second valuable property is that the model can be easily extended by adding decisions and chances. For example, one could change both decisions in Figure 15 by one group called "Semantic hypermedia as the engine of application state," and add to that group decisions like "Machine-readable hypermedia as the client interface" and "Shared semantic data model between Client and Server," thus starting to build a new architectural style that reuses part of REST structure and design rationale. In addition, one can reason about applying REST to other problem domains by modifying the hierarchy of utilities to reinterpret the software qualities used in the derivation of REST. For example, if we change "Successful WWW Architecture" for "Successful SOA" in Figure 11, all the other utilities would change their meaning, and we would be forced to change the content of the chances and to re-consider every arrow from decision to chance and chance to utility, but the decisions comprising REST would remain the same.

On the other hand, the example also illustrates some shortcomings of our model. First, the chances are possibly too verbose, and it is hard to define them in such a way that guarantees their reuse with future decisions (like it would be the case of a chance like "lines of code" or "memory size"). Second, if an inexperienced reader reads through the node names, he won't understand much, unless good comments are provided. One way to solve these problems would be to define primitive architectural decisions, which would be used to model any architectural style. Such effort would improve the understandability of the diagrams, by specifying once the semantics of the decision names, and also the chances that are compatible with each decision.

7. REFERENCES

[1] Allamaraju, S. (2008). Describing RESTful Applications. Retrieved May 4, 2009, from http://www.infoq.com/articles/subbu-allamaraju-rest.

[2] Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., et al. (2007). Damia: a data mashup fabric for intranet applications. En Proceedings of the 33rd international conference on Very large data bases (pp. 1370-1373). Vienna, Austria: VLDB Endowment.

[3] API Dashboard: ProgrammableWeb. (2009). Retrieved June 10, 2009, from http://www.programmableweb.com/apis.

[4] Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (1999). Non-Functional Requirements in Software Engineering. Springer.

[5] Dillon, T., Wu, C. y Chang, E. (2008). Reference Architectural Styles for Service-Oriented Computing. En Network and Parallel Computing (pp. 543-555).

[6] Fielding, R. (2008a, March 22). On software architecture » Untangled. Retrieved June 2, 2009, from http://roy.gbiv.com/untangled/2008/on-software-architecture.

[7] Fielding, R. (2008b, October 20). REST APIs must be hypertext-driven. Retrieved June 2, 2009, from http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[1] This is one of the most valuable properties of this model. We are not saying that the only truth about REST is and will always be contained in the Dissertation, but if we are interested in knowing anything about the design rationale for REST from the Dissertation, the Diagrams are a great way to save time and avoid misunderstandings. Ultimately, the right REST influence diagram can be reset to whatever is the correct representation of this style.

[8] Fielding, R. (2009, March 20). It is okay to use POST » Untangled. Retrieved June 2, 2009, from http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post.

[9] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. University of California, Irvine.

[10] Floyd, I. R., Jones, M. C., Rathi, D. y Twidale, M. B. (2007). Web Mash-ups and Patchwork Prototyping: User-driven technological innovation with Web 2.0 and Open Source Software. En Proceedings of the 40th Annual Hawaii International Conference on System Sciences (pág. 86). IEEE Computer Society.

[11] Flying Logic. (s.d.). Sciral. Retrieved June 1, 2009, from http://flyinglogic.com/.

[12] Gall, N. (2008, November 10). Why WOA vs. SOA Doesn't Matter. Retrieved June 2, 2009, from http://www.itbusinessedge.com/cm/community/features/interviews/blog/why-woa-vs-soa-doesnt-matter/?cs=23092.

[13] Hinchcliffe , D. (2008). What Is WOA? It's The Future of Service-Oriented Architecture (SOA). Dion Hinchcliffe's Blog - Musings and Ruminations on Building Great Systems. Retrieved January 11, 2008, from http://hinchcliffe.org/archive/2008/02/27/16617.aspx.

[14] Jansen, A. & Bosch, J. (2005). Software Architecture as a Set of Architectural Design Decisions. En 5th. Pittsburgh, PA, USA.

[15] Jhingran, A. (2006). Enterprise information mashups: integrating information, simply. En Proceedings of the 32nd international conference on Very large data bases (pp. 3-4). Seoul, Korea: VLDB Endowment.

[16] Johnson, P., Lagerström, R., Närman, P., & Simonsson, M. (2007). Enterprise architecture analysis with extended influence diagrams . Information Systems Frontiers, 9(2-3). doi: 10.1007/s10796-007-9030-y.

[17] Kühne, T. (2006). Matters of (Meta-) Modeling. Software and Systems Modeling, 5(4), 369-385. doi: 10.1007/s10270-006-0017-9.

[18] Lublinsky, B. (2007, September 1). Defining SOA as an architectural style. IBM Developer Works Technical Library. Retrieved April 29, 2009, from http://www.ibm.com/developerworks/architecture/library/ar-soastyle/

[19] Mehta, N. R., & Medvidovic, N. (2003). Composing architectural styles from architectural primitives. SIGSOFT Softw. Eng. Notes, 28(5), 347-350. doi: 10.1145/949952.940118.

[20] Muehlen, Z., Nickerson, J., & Swenson, K. (2004). Developing web services choreography standards—the case of REST vs. SOAP. Decision Support Systems, 40(1). doi: 10.1016/j.dss.2004.04.008.

[21] Overdick, H. (2007). The Resource-Oriented Architecture. En Services, 2007 IEEE Congress on (págs. 340-347). doi: 10.1109/SERVICES.2007.66.

[22] Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. "big"' web services: making the right architectural decision. En Proceeding of the 17th international conference on World Wide Web (pp. 805-814). Beijing, China: ACM. doi: 10.1145/1367497.1367606.

[23] Richardson, L. y Ruby, S. (2007). Restful web services (p. 446). O'Reilly.

[24] Vinoski, S. (2008a). RPC and REST: Dilemma, Disruption, and Displacement. Internet Computing, IEEE, 12(5), 92-95. doi: 10.1109/MIC.2008.109.

[25] Vinoski, S. (2008b). Convenience Over Correctness. Internet Computing, IEEE, 12(4), 89-92. doi: 10.1109/MIC.2008.75.

[26] Vinoski, S. (2007). REST Eye for the SOA Guy. IEEE Internet Computing, 11(1), 82-84.

[27] Wong, J. & Hong, J. I. (2007). Making mashups with marmite: towards end-user programming for the web. En Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 1435-1444). San Jose, California, USA: ACM. doi: 10.1145/1240624.1240842.

[28] Xu, X., Zhu, L., Liu, Y., & Staples, M. (2008). Resource-oriented business process modeling for ultra-large-scale systems. En Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems (pp. 65-68). Leipzig, Germany: ACM. doi: 10.1145/1370700.1370718.

A Formal Definition of RESTful Semantic Web Services

Antonio Garrote Hernández
University of Salamanca
Salamanca, Spain
agarrote@usal.es

María N.Moreno García
University of Salamanca
Salamanca, Spain
mmg@usal.es

ABSTRACT

In this article a formal model applying REST architectural principles to the description of semantic web services is introduced, including the discussion of its syntax and operational semantics. RESTful semantic resources are described using the concept of tuple spaces being manipulated by HTTP methods that are related to classical tuple space operations. On the other hand, RESTful resources creation, destruction and other dynamic aspects of distributed HTTP computations involving coordination between HTTP agents and services are modeled using process calculus style named channels and message passing mechanisms.
The resulting model allows for a complete and rigorous description of resource based web systems, where agents taking part in a computation publish data encoded according to semantic standards through public triple repositories identified by well known URIs. The model can be used to describe complex interaction scenarios where coordination and composition of resources are required. One of such scenarios taken from the literature about web services choreography is analyzed from the point of view of the proposed model. Finally, possible extensions to the formalism, such as the inclusion of a description logics based type system associated to the semantic resources or possible extensions to HTTP operations are briefly explored.

1. INTRODUCTION

In recent years, semantic web services [1] and RESTful [2] web services have been two important topics for researchers and practitioners in the field of distributed web computation, although they have remained largely unrelated.
RESTful web services have achieved a great success in their application to actual web development but limited research have been undertaken in their formalization and expressiveness. On the other hand, ongoing effort in the standardization of semantic web services technology draws heavy influence from WS-* standards and retains strong RPC semantics. Incipient proposals for developing more RESTful semantic web services like hRESTS [3] or SA-REST [4] are working to bring the world of semantic web services closer to the field of RESTful services. Nevertheless, different foundations for the development of semantic web services like triple space computing models have also been proposed [5]. This model shows remarkable similarities with the architectural principles of HTTP and REST. This model can be used to describe a kind of RESTful semantic web services characterized by the following points:

- A RESTful semantic resource consists of a set of triples stored in a certain shared memory repository accessible by processes taking part in a computation.
- The triple space can be accessed through an associated URI that can be linked from related resources.
- The triple space containing the resource's triples can be manipulated using HTTP requests issued to the resource URI according to REST semantics.

This kind of RESTful web services may be applied to a wide range of actual web resources from a XHTML page with embedded RDFa [6] triples, to a web application connected to a triple store. It is also conceptually similar to the semantic web standard for the SPARQL protocol [sparqlprot].

In this paper a formal calculus for distributed computation using this kind of RESTful semantic web services will be described. This formalism merges aspects from triple space computation and other formalisms like process calculi [7] to build a generic abstraction of the computational model underlying RESTful semantic web services distributed computation.

2. SEMANTIC RESOURCES AND TRIPLE SPACES

The basis of the calculus is the manipulation of semantic meta data represented as triples. Each triple has subject, predicate and object, as described by the Resource Description Framework W3C's recommendation [10]. Components of a triple (v) can consist of URIs (μ) or literals (λ). Every computation described in the calculus consists in the manipulation of triples stored in shared data spaces known as triple spaces [5] (θ_i) by a set of distributed processes $(P, Q, ...)$.
This model of distributed computation known as *generative communication* [11] was pioneered by the Linda system for distributed computation. Linda original operations are

WS-REST 2010, April 26, 2010; Raleigh, NC, USA

$$
\begin{array}{lcl}
P & ::= & 0 \mid T \mid P|P \mid !P \mid if\, T\, ?\, P.P \mid x ::= T \\
T & ::= & rd(\theta_i, p) \mid in(\theta_i, p) \mid out(\theta_i, v) \mid swap(\theta_i, p, v) \mid \\
 & & rdb(\theta_i, p) \mid inb(\theta_i, p) \mid notify(\theta_i, \rho, v) \\
 & & \\
\theta & ::= & \{ \text{ triple spaces } \} \\
\rho & ::= & \{in, out\} \\
\mu & ::= & \{\text{URIs}\} \\
\lambda & ::= & \{\text{literals}\} \\
p & ::= & \{\text{patterns}\} \\
v & ::= & \{\text{values}\} = \{\mu\} \cup \{\lambda\} \cup < p, v > \cup < p, \theta_i >
\end{array}
$$

Table 1: triple space operations syntax

used to describe the different ways in which a triple space can be manipulated. Symbol rd designates an operation for reading triples from a triple space without removing them, in symbol for reading and removing the triples , and out symbol defines an operation for inserting new triples in the triple space. Blocking versions of rd and in operations are also defined. In addition to these operations, the $notify$ [12] operation, which allows processes to be notified when other process manipulates the triple space, and the atomic $swap$ operation [13], combining read and write in a single operation, are also defined.

A process in the calculus is defined as a finite sequence of operations over the triple space. Parallel composition of processes, replication and a simple type of conditional based on the number of triples recovered from a triple by an operation space and simple name matching of values are defined.

Triple space operations accept as arguments sets of triples (v) or patterns (p) that must be matched against triples in the triple space. No particular pattern mechanism is chosen in this definition of the calculus. It can be assumed that patterns are a built using a subset of the SPARQL [14] query language with basic name substitution [16]. A pattern can be matched against a collection of tuples ($< p, v >$) or a tuple space ($< p, \theta_i >$) obtaining as a result a collection of bindings for the pattern variables. Each of these bindings, together with the original pattern, can be transformed into a set of matched tuples or the empty set.

Operational semantics for this formal notation follows previous formalizations of Linda type systems [11]. We have chosen to define *ordered* semantics for the triple space operations. As a consequence, emission and rendering of messages can be regarded as a single atomic operation. It can be proved that Linda systems with ordered semantics are Turing complete [17]. Table 2 shows the semantics of the calculus using labeled transitions:

The labeled transition relation $\rightarrow$ for triple space operations is the smallest one satisfying axioms (1) - (10). Rules (1) and (2) are classical rules for parallel composition and structural congruence of processes [7]. Rule (3) defines the spawning of a new process. Rules (4) through (7) define the main operations over triple spaces and how the reduction of those rules modify the triples inside the triple space they are applied to. Rules for the blocking versions of the operation have been omitted for the sake of brevity. Rules (8) and (9) define the semantics for the $notify$ operation. They show

(1)	$\frac{P \rightarrow P'}{P\|Q \rightarrow P'\|Q}$
(2)	$\frac{P \rightarrow P'}{Q \rightarrow Q'} if\, P \equiv Q \, and \, P' \equiv Q'$
(3)	$!P.Q \rightarrow Q \mid P$
(4)	$\frac{rd(\theta_i,p).P}{rd(\theta_i,p).P \xrightarrow{<p,\theta_i>} P}$
(5)	$\frac{in(\theta_i,p).P}{rd(\theta_i,p).P \xrightarrow{<p,\theta_i>} P, \theta_i = \theta_i - <p,\theta_i>}$
(6)	$\frac{out(\theta_i,v).P}{out(\theta_i,v).P \xrightarrow{\overline{v}} P, \theta_i = \theta_i \cup v}$
(7)	$\frac{swap(\theta_i,p,v).P}{swap(\theta_i,p,v).P \xrightarrow{<p,\theta_i>,\overline{v}} P, \theta_i = \theta_i - <p,\theta_i> \cup v}$
(8)	$\frac{out(\theta_i,v).Q}{notify(\theta_i,out,p).P \| out(\theta_i,v).Q \xrightarrow{\overline{v},<p,v>} P\|Q}$
(9)	$\frac{in(\theta_i,p).Q}{notify(\theta_i,in,q).P \| in(\theta_i,p).Q \xrightarrow{<p,v>,<q,<p,v>>} P\|Q}$
(10)	$\frac{if\, T\, P.Q}{if\, T\, P.Q \xrightarrow{\overline{0}} Q}, \frac{if\, T\, P.Q}{if\, T\, P.Q \xrightarrow{\overline{v}} P}$

Table 2: operational semantics for triple space operations

the relation with in and out operations and how the pattern used as an argument for the $notify$ operation is applied to the triples removed in an in operation and written in an out operation before the reduction of a $notify$ rule takes place. Finally, rule (10) defines the semantics for simple if branching construct defined in the calculus.

From this definition of operational semantics, remarkable conceptual similarity between a triple space and a HTTP resource can be observed. The syntax and semantics defined for the triple space operations could be enough basis for the description of RESTful architectures if key aspects of the triple space model are identified with different aspects of the HTTP protocol:

- A triple space can be identified with an HTTP resource.
- The identifier of the triple space can be identified with the URI associated to a HTTP resource.
- GET HTTP operations can be defined as rd operations over the triple space.
- POST HTTP operations can be defined as out operations over the triple space.
- PUT HTTP operations can be defined as $swap$ operations over the triple space.
- DELETE HTTP operations can be defined as in operations over the triple space.

$$
\begin{array}{rcl}
P & ::= & 0 \mid T \mid M \mid P|P \mid !P \mid if\ T\ ?\ P.P \mid x ::= T \mid \\
 & & new\ \mu\ in\ P \\
M & ::= & \overline{req(\mu)}[m,p,v] \mid [m,p,v]req(\mu) \mid \overline{resp(\mu)}[c,v] \mid \\
 & & [c,v]resp(\mu) \\
 & & \\
m & ::= & \{get, post, put, delete\} \\
c & ::= & \{200, 201, 404, 401\}
\end{array}
$$

Table 3: process communication syntax

3. SEMANTIC RESOURCES AND RUNTIME PROCESSES

The simple model for the description of HTTP resources introduced in the previous section presents some important shortcomings. In the original conception of the tuple space model, the shared space is supposed to be globally available and unique. On the contrary, a convenient way to model HTTP resources involves the use of as many triple spaces as resources are being modeled. As a consequence, these resources would not be static shared spaces, but they would be created after POST operations and destroyed with DELETE operations. The definition for POST and DELETE operations discussed in the previous section, where both HTTP methods were equated to *out* and *in* triple space operations, does not fit into triple space model semantics, since *out* and *in* operations can only manipulate triples but cannot create and destroy whole triple spaces. The formalism could be extended [15] introducing new operations over triple spaces as well as triples. With these extensions, triple spaces could be thought as processes in the calculus that can be spawned, receive messages from other processes and finish their execution.

Other important feature of the triple space model that have been defined is the ability to assign URIs to triple spaces. Processes can gain access to new triple spaces by reading triples containing URIs in their components. From this point of view, triple spaces can be regarded as mobile processes as described by the Pi-Calculus [7] [8]. In this kind of process calculi, processes coordinate with each other exchanging messages through named channels that can be exchanged inside the components of the messages being received or sent by processes.

The Pi-Calculus formalism is extremely useful for the description of web computation, since the concept of named channels being exchanged between processes closely resembles the exchange of web resources containing URIs between clients and servers identified by URIs.

The calculus previously introduced can be expanded to allow this new way of inter process communication as shown in table 3. The operations described are variants of the ones described in the polyadic Pi-Calculus [9]. Messages can be sent through named channels representing URIs (μ). Messages can be requests (req) or responses ($resp$) and processes can send or receive both kind of messages. Messages themselves comprise a method (m), a pattern (p) and a value (v), in the case of requests and a code (c) with a value for responses. Additionally, a construction introducing a new URI in a process and restricting its application ($new\ \mu\ in\ P$) is also defined.

$$
\begin{array}{ll}
(11) & \dfrac{P \xrightarrow{\overline{req(\mu)[m,p,v]}} P', Q \xrightarrow{[m,p,v]req(\mu)} Q'}{P|Q \rightarrow P'|Q'} \\
 & \\
(12) & \dfrac{P \xrightarrow{\overline{resp(\mu)[c,v]}} P', Q \xrightarrow{[c,v]resp(\mu)} Q'}{P|Q \rightarrow P'|Q'} \\
 & \\
(13) & \dfrac{\overline{req(\mu)[m,p,v]}.P}{\overline{req(\mu)[m,p,v]}.P \xrightarrow{*} [c,v]resp(\mu).Q} \\
 & \\
(14) & \dfrac{[m,p,v]req(\mu).P}{[m,p,v]req(\mu).P \xrightarrow{*} \overline{resp(\mu)[c,v]}.Q}
\end{array}
$$

Table 4: operational semantics for triple space operations

Operational semantics for these operations are defined in table 4. These rules also follow closely the operational semantics defined for the polyadic Pi-Calculus. Rules (11) and (12) are adaptations to the chosen syntax of the communication rule from the Pi-Calculus, showing the reaction between a process sending a message and a process receiving a message. Rules (13) and (14) impose restrictions in the order requests and responses can be exchanged. According to these axioms, if a process sends a request through an URI, it must expect a response in a certain number of reductions. In the same way, if a process receives a request through an URI it must send a response through the same URI in a certain number of reductions.

The syntax and semantics described make possible the description of HTTP resources as processes receiving messages through URIs from HTTP agent processes. These messages consist of a HTTP operation plus a triple pattern and/or semantic meta data. Resource processes can be spawned from other processes after receiving a POST request and can finished its execution after receiving a DELETE message. Responses returned from resource processes to agent processes can also contain URIs referencing other resources.

4. MODELING RESTFUL SEMANTIC SERVICES

In previous sections, two different formalisms for distributed computing have been introduced: the triple space computing model and mobile processes calculus. Both formalism offer different coordination mechanisms among processes performing a distributed computation: distributed shared memory the former and message passing the later. Both mechanisms can be regarded as variants of a more generic interprocess coordination mechanism [16]. Nevertheless, each model is specially well suited for the description of certain aspects of computations using RESTful semantic services. The triple space formalism offers an excellent description of semantic RESTful resources as static repositories of semantic data, using operations over the stored data as the main coordination mechanism among processes manipulating the stored triples. Conversely, process calculi are well suited for the description of the HTTP protocol as a message passing mechanism through named channels (URIs associated to IPs) among web agent processes and resource processes, as well as for the description of the dynamic aspects of REST-

$$
\begin{array}{rcl}
R_{REST}(\theta,\mu) & ::= & [m,v,p]req(\mu).\, if\ m = get\ ?\ R_{get}(\theta,\mu). \\
 & & if\ m = post\ ?\ R_{post}(\theta,\mu). \\
 & & if\ m = put\ ?\ R_{put}(\theta,\mu). \\
 & & if\ m = delete\ ?\ R_{delete}(\theta,\mu). \\
 & & \overline{resp(\mu)}[406,0].R_{REST}(\theta,\mu) \\
 & & \\
R_{get}(\theta,\mu) & ::= & x ::= rd(\theta,p).\overline{resp(\mu)}[200,x].R_{REST} \\
 & & \\
R_{post}(\theta,\mu) & ::= & \overline{new\ \nu\ in\ out}(\theta,<p,\nu>).!R(\theta,\nu). \\
 & & \overline{resp(\mu)}[201,<p,\nu>].R_{REST} \\
 & & \\
R_{put}(\theta,\mu) & ::= & swap(\theta,p,v).\overline{resp(\mu)}[200,v].R_{REST} \\
 & & \\
R_{delete}(\theta,\mu) & ::= & in(\theta,p_\mu).\overline{resp(\mu)}[200,0].0
\end{array}
$$

Table 5: parametric definition of a simple semantic RESTful resource

ful semantic resources such as their creation and destruction.

In this section, a description of semantic RESTful computation is defined by combining aspects of both formalisms and using the syntax and operational semantics introduced in previous sections. The main features of this model can be summarized as follows:

- Any computation is performed by distributed processes: agents and resources.
- Any process has a number of associated triple spaces that can be manipulated using triple space operations.
- A certain number of processes sharing the same triple spaces can be grouped in a *computational locus*: a server web application or a web browser application are examples of computational loci.
- Resource processes will have an associated named channel (URI) where they can receive messages from other processes.
- Agent processes do not have associated named channels but can send messages to resource processes through URIs stored in their triple spaces.
- Intra computational locus coordination is triple space based, inter computational loci coordination is message passing based
- Named channels (URI) can be exchanged via message passing and they can be stored as part of the triples inside a triple space
- Triple space handlers cannot be exchanged via message passing

Using this model, a semantic RESTful resource can be formalized as a process with an associated triple space and URI, processing remote requests according to REST semantics: $S_{REST}(\theta,\mu)$. Table 5 shows the formalization of a basic semantic RESTful resource.

This definition of a RESTful semantic web service describes a process with an associated triple space that receives messages through an assigned URI. If the message is a *get* HTTP request the process simply applies the triple pattern received to the associated triple space, and returns the matching triples. If the message is a *post* HTTP request, the process introduces a new URI and binds the subject of the triple pattern sent in the request to the new URI, generating as a result a new set of triples. These triples are written into the triple space and a new process for the resource to be created, associated to the same triple space and the new URI is spawned. The URI of the new resource process is returned in the response message for the HTTP POST operation. A *put* HTTP request containing a pattern and a set of new triples is transformed into a *swap* operation over the associated triple space, where the triples matching the received pattern are removed and replaced by the received triples. Finally, a *delete* operation means the end of the execution of the resource process as well as the deletion from the triple space of all the triples containing the resource associated URI (p_μ).

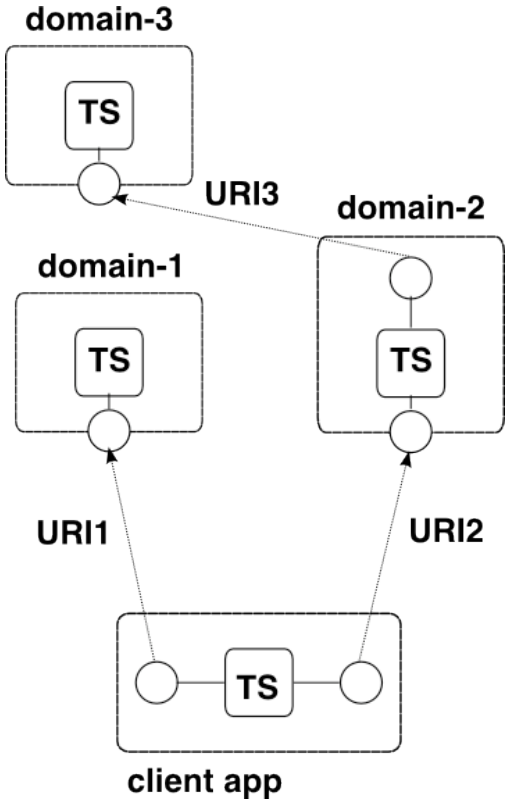

Figure 1: Sample semantic RESTful computation

Figure 1 shows a graphical representation of the proposed model displaying one client application and three different domains. Each domain hosts a semantic RESTful web service with an associated triple space and URI. The client application contains two agent processes that communicate with each other through the client application triple space. Agent processes can send messages to the resources at *domain 1* and *domain 2* through the URIs associated to each resource process. *Domain 2* also contains an agent. When

the resource at *domain 2* modifies the triple space, the agent at *domain 2* receives a $notify$ message and starts the communication with the resource at *domain 3*.

5. COORDINATION AND COMPOSITION OF RESTFUL RESOURCES

One of the main goals of the formalization of RESTful semantic services is the possibility of describing interaction patterns between resources and clients. Complex processes and workflows can be modeled as well defined and reusable descriptions, that can be automatically executed by software implementations of the calculus.

Different frameworks for web services orchestration and choreography have been proposed in the world of WS-* web services, like the W3C standard WS-CDL [18]. In this section, a simple example taken from WS-CDL literature [19], will be formalized using the calculus introduced in previous sections.

The problem discussed involves three parties: a *Buyer*, a *Seller* and a *Shipper* actors, performing a purchase transaction. The protocol is described in the following terms:

- *Buyer* asks *Seller* to offer a quote for a fixed good.
- *Seller* replies with a quote.
- *Buyer* accepts or rejects the quote.
- If buyer accepts, then *Seller* sends a confirmation to *Buyer*, then asks for delivery details to the *Shipper*.
- Finally, *Shipper* sends the delivery details to *Buyer*.

In order to describe this system using RESTful semantic web services, three different computational loci must be defined:

- The buyer client application, containing a Buyer HTTP agent process.
- The Seller web application, containing two resources: Products and PurchaseOrders.
- The Seller web application contains also a QuoteUpdater agent and a Shipper agent.
- The Buyer web application, containing one resource: ShipmentOrder.

Proper definition of the semantic meta data for the Product, PurchaseOrder and ShipmentOrder resources in some ontology definition language like OWL, as well as the precise content of the triple patterns, are not included for the sake of brevity. A full description of the computation is shown in figure 2

The Buyer HTTP agent is described in table 6.The definition of the process is parametric on the URI of the Product resource to buy (μ_p). The Buyer agent creates a new PurchaseOrder resource through the URI μ_{pos} in the Seller application using a POST request. The pattern sent in this request (p_{no}) includes the URI of the product to be purchased and a literal for the state of the order with value "created".

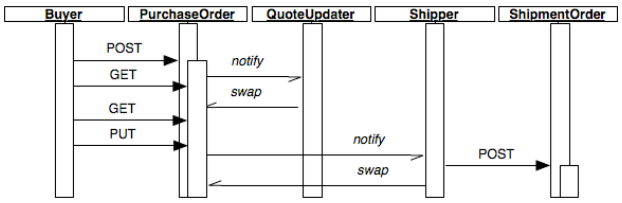

Figure 2: Sequence diagram for the modeled transaction

$$
\begin{array}{rcl}
Buyer(\mu_p) & ::= & CreateOrder.PollQuote \\
CreateOrder & ::= & \overline{req(\mu_{pos})}[post, p_{no}, 0]. \\
 & & [201, \mu_{po}]resp(\mu_{pos}) \\
PollQuote & ::= & \overline{req(\mu_{po})}[post, p_q, 0]. \\
 & & [200, q]resp(\mu_{pos}). \\
 & & if\ q\ ?\ ProcessOrder.PollQuote \\
ProcessOrder & ::= & \overline{req(\mu_{po})}[get, p_{mp}, 0]. \\
 & & [200, vq].resp(\mu po). \\
 & & if\ vq\ ?\ AcceptOrder.RejectOrder \\
AcceptOrder & ::= & \overline{req(\mu_{po})}[put, p_{mp}, accept]. \\
 & & [200, x]resp(\mu_{po}).0 \\
RejectOrder & ::= & \overline{req(\mu_{po})}[delete, 0, 0]. \\
 & & [200, x]resp(\mu_{po}).0
\end{array}
$$

Table 6: Buyer application processes

After the order is created and the URI for the new PurchaseOrder resource is returned μ_{po}, the Buyer process starts polling the newly created resource using GET requests containing a triple pattern that tries to retrieve the triples for the resource μ_{po} with an state of "quoted" (p_q). When one of these GET requests returns successfully with some triples, the Buyer agent issues a new GET request trying to retrieve the triples for the PurchaseOrder resource identified by μ_{po}, with price minor or equal to literal value min_price, using the pattern p_{mp}.

If this last GET request fails, the Buyer agent rejects the quote deleting the resource with a *delete* request. On the other hand, if some triples are returned, the Buyer process accepts the quote issuing a PUT request to the PurchaseOrder process, updating in this way the state of the resource to the value "accepted".

The Seller web application, described in table 7, contains two RESTful semantic resources Products ($Prods_{rest}$) and PurchaseOrders ($PurchOrds_{rest}$). Both of them have their associated triple spaces (θ_p, θ_{po}). These triple spaces are also manipulated by two agent processes QuoteUpdater ($QUpd$) and Shipper ($Ship$). QuoteUpdater gets a notification after each write operation of triples matching state "created" (p_{no}). Then, it goes through an internal transition τ that generates a price for the product in the PurchaseOrder and,

$$
\begin{aligned}
Prods &::= R_{rest}(\mu_{ps}, \theta_p) \\
PurchOrds &::= R_{rest}(\mu_{pos}, \theta_{po}) \\
QUpd &::= notify(\theta_{po}, write, p_{no}).pr ::= \tau. \\
&\quad swap(\theta_{po}, p_{no}, << p_{no}, quoted, > pr >) \\
&\quad .QUpd \\
Ship &::= \overline{notify}(\theta_{po}, write, p_{ok}). \\
&\quad \overline{req}(\mu_s)[post, p_{pok}, 0]. \\
&\quad [201, so]resp(\mu_s) \\
&\quad swap(\theta_{po}, p_{pok}, so).Ship
\end{aligned}
$$

Table 7: Seller web application

$$
Shipms ::= R_{rest}(\mu_s, \theta_s)
$$

Table 8: Shipment web application

finally, updates its triples with the price and a state of "quoted".

The Shipper agent gets notified when a new set of triples for a resource with state "accepted" (p_{ok}) is written in the PurchaseOrders triple space. After receiving this notification, it issues a POST request to the ShipmentOrders resource and updates the state of the PurchaseOrders resource with the state of "shipped" and a reference to the newly created ShipmentOrders resource.

Finally, The Shipment web application just contains a REST resource for the ShipmentOrders resources ($Shipms$). The formalization of the Shipment web application is shown in table 8.

6. CONCLUSIONS AND FUTURE WORK

RESTful semantic web services have the potential to enable a new generation of distributed applications, retaining the scalable and successful architecture of today's web, and adding the powerful data description and interoperability capacities of semantic data.

With this paper we have tried to provide a precise definition of a certain theoretical model for this kind of computation, combining two well known formalisms: tuple space computing and process calculi.

We have found that both formalisms are specially well suited for describing different aspects of RESTful semantic computation: triple space computing as a way of describing computations taking place inside computational loci, like web applications, that will be exposed to external processes as RESTful resources, and process calculus for the description of the exchange of HTTP messages between HTTP agents across different computational loci.

In our conception, RESTful semantic services are just processes receiving HTTP messages through a well known URI and manipulating an associated triple space according to the messages received and REST semantics.

As an example of how this calculus can be used for describing actual computations, a basic example from the literature on web services orchestration has been formalized in terms of a set of RESTful semantic web services and agents.

Further work on the calculus should deal with different aspects not presented in this article.

A type theory for the calculus must be developed. Ontology languages like OWL, rooted in the theoretical background of description logics, make possible the introduction of type systems dealing with well typed patterns and values associated to URIs and RESTful resources.

In its current form, the calculus describes computations as different sets of equations for each party taking part in the interaction. A global calculus for the description of the computation [19] must be defined, taking into account the inherent duality of the message passing operations introduced in the present formalism.

Furthermore, the message passing portion of the calculus lacks some features present in the triple space portion, like the presence of blocking operations, or a notification mechanism. Research must be taken in the different ways of extending REST semantics to allow this kind of coordination primitives in HTTP agents, communicating with each other through a shared RESTful resource.

A current implementation of the introduced calculus, built on top of technologies like Erlang OTP, RabbitMQ AMQP queue broker and the Open Sesame triple repository is currently being developed. In this implementation extensions of HTTP operations with blocking semantics, notifications and a subscription mechanism based in the websockets [20] standard have also being tested as part of our ongoing research on RESTful semantic web services.

7. REFERENCES

[1] Fensel D. et alt. (2006) Enabling Semantic Web Services: The Web Service Modeling Ontology Springer-Verlang.

[2] Fielding R. (2000) Architectural Styles and the Design of Network-based Software Architectures University of California, Irvine

[3] Kopecky, Vitvar & Fensel. (2009) hRESTS and MicroWSMO CMS WG Working Draft

[4] A. P. Sheth & K. Gomadam & J. Lathem. (2007) SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups IEEE Internet Computing, pages 91-94

[5] Fensel. (2004) D. Triple-space computing: Semantic Web Services based on persistent publication of information IFIP Internation Conf. on Intelligence in Communication Systems, pages 43-53

[6] Adida & Birbeck. (2008) RDFa in XHTML: Syntax and Processing W3C Recommendation

[7] R. Milner (1999) Communicating and Mobile Systems: the Pi-Calculus Cambridge University Press

[8] R. Milner & J. Parrow & D. Walker (1989) A calculus for mobile processes

University of Edinburgh
[9] R. Milner (1991) The Polyadic Pi-Calculus: a Tutorial
Logic and Algebra of Specification
[10] P. Hayes. (2004) RDF Semantics
W3C Recommendation
[11] D. Gelernter. (1985) Generative communication in
Linda
ACM Transactions on Programming Languages and
Systems, vol 7, pages 80-12
[12] N. Busi & R. Gorrieri & G. Zavattaro. (2000) Process
Calculi for Coordination: from Linda to JavaSpaces
Proc. of AMAST, LNCS 1816, pages 198-212
[13] A. Neves & E. Pelison & M. Correia & J. Da Silva.
(2008) DepSpace: A Byzantine fault-tolerant
coordination service
Proceedings of the 3rd ACM SIGOPS/EuroSys
European Conference on Computer Systems -
EuroSys, pages163-176
[14] A. Seaborne & E. Prud'homeaux. (2008) SPARQL
Query Language for RDF
W3C Recommendation
[15] E. Simperl & R. Krummenacher R. & L. Nixon. (2007)
A Coordination Model for Triplespace Computing
9th International Conference on Coordination Models
and Languages, pages 1-8
[16] D. Gorla (2006) On the relative expressive power of
asynchronous communication primitives
Proceedings of 9th International Conference on
Foundations of Software Science and Computation
Structures, vol 3921, pages 47-62
[17] N. Busi & R. Gorrieri & G. Zavattar. (2000) On the
Expresiveness of Linda Coordination Primitives
Information and Computation
[18] Kavantzas & Burdett & Ritzinger & Fletcher & Lafon.
(2005) Web Services Choreography Description
Language Version 1.0
W3C Candidate Recommendation
[19] M. Carbone & K. Honda N. Yoshida & R. Milner &
G. Brown & S. Ross-talbot. (2006) A theoretical basis
of communication-centred concurrent programming
[20] I. Hickson. (2010) The Web Sockets API
W3C Editor's Draft

A RESTful Messaging System for Asynchronous Distributed Processing

Ian Jacobi and Alexey Radul
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{pipian,axch}@mit.edu

ABSTRACT

Traditionally, distributed computing problems have been solved by partitioning data into chunks able to be handled by commodity hardware. Such partitioning is not possible in cases where there are a high number of dependencies or high dimensionality, as in reasoning and expert systems. This renders such problems less tractable for distributed systems. By partitioning the algorithm, rather than the data, we can achieve a more general application of distributed computing.

Partitioning the algorithm in a reasonable manner may require tighter communication between members of the network, even though many networks can only be assumed to be weakly-connected. We believe that a decentralized implementation of propagator networks may resolve the problem. By placing several constraints on the merging of data in these distributed propagator networks, we can easily synchronize information and obtain eventual convergence without serializing operations within the network.

We present a RESTful messaging mechanism for distributing propagator networks, using mechanisms that result in eventual convergence of knowledge in a weakly-connected network. By enforcing RESTful design constraints on the messaging mechanism, we can reduce bandwidth usage and obtain greater scalability in heterogeneous networks.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications, Distributed databases; D.1.3 [**Concurrent Programming**]: Distributed programming; H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Design, Reliability

Keywords

propagator networks, REST, distributed computing, consistency, synchronization

WS-REST 2010, April 26, 2010; Raleigh, NC, USA

1. INTRODUCTION

As cheap commodity computers have become more powerful, there has been a corresponding explosion in the number of distributed computing efforts making use of large networks of commodity hardware to perform complex computation. Heteogeneous distributed systems such as SETI@home [2], the related BOINC platform [1], and Google's MapReduce implementation [7], have been used to solve many otherwise intractible problems in recent years.

Many of these efforts have relied on partitioning the domain of a problem into smaller, more tractable chunks. Individual hosts then may perform identical processing on each data partition to obtain partial results that are later merged to produce a final result. Unfortunately, data partitioning in this manner may not be feasible for all problems. Problems that feature a high degree of dimensionality and have a high number of data dependencies in the final solution are well known to be difficult to partition. [4, 15]

Expert systems and reasoning engines, for example, may depend on a sizeable number of logical statements to calculate a meaningful answer to a query. Although reasoning with just a few simple pattern-matching rules has been done successfully with distributed algorithms [20], more complex rule systems may have a larger number of dependencies in a rule which may not fit into a single data partition.

One solution to such problems in reasoning is splitting the algorithm into several "rounds" of reasoning over different rules, but this merely trades space for time. Another possible solution may not require such a trade off. In the Rete algorithm [11, 9], a well-known algorithm for forward-chaining reasoning, a network of nodes is used to separate pattern-matching filters from a merge function over the outputs of several filters and other merge functions. Rather than dividing the problem by partitioning data across multiple nodes, the Rete algorithm suggests that we divide the algorithm itself.

One issue arises when applying this solution more generally: traditional distributed computing efforts like MapReduce and BOINC assume a weakly-connected network of nodes that only check in with a central server when a given subtask is complete. Reducing communication in this way strongly divides the domain so that tasks that rely on continuously updating inputs, such as reasoning, are infeasible. A loosely-connected decentralized model could prove to be more flexible than centralized architectures, but would be dependent on more frequent communications. We believe that a distributed variant of the data propagation model of computation [17] may provide such an architecture.

Unfortunately, data propagation as generically described makes no constraints as to how strongly connected the network of propagators is. Many of the examples Radul presents assume the existence of a tightly-linked computational platform. This assumption may not hold in many distributed computing scenarios that rely on a weakly-connected network. By utilizing several key constraints on propagator networks, we can safely eliminate this assumption; instead, we need only assume reliable broadcast and convergence of knowledge in a weakly-connected network, which is known to be possible. [3, 8]

Such a significant change to the implementation of distributed systems also requires careful reconsideration of the underlying technologies used by the systems. While traditional centralized models have tended to rely on simple client-server protocols, a decentralized, data-driven model like data propagation requires a different architecture. Given that propagator networks gradually refine data in distinct "cells", we believe that treating such cells as resources in a system using Representational State Transfer (REST) may give a combination of flexibility and simplicity of design that may not be possible with other implementations. Hence, we have constructed a "RESTful" implementation of a distributed propagation system, named DProp[1].

In this paper, we first introduce the concept of propagator networks in Section 2. Section 3 gives a brief overview of the REST architecture, while Section 4 outlines the design choices we made to construct our RESTful implementation of propagator networks. We then discuss mechanisms that reduce the incidence of race conditions (Section 6) as well as possible security mechanisms within our system (Section 7). Finally, we review related work in this field.

2. DATA PROPAGATION

The data propagation model of computation [17, 16] is a concurrent message-passing programming paradigm. Data propagation operates over a network of stateless computational "propagators" connected by a number of stateful "cells" that may be used for both input and output.

Although superficially similar to traditional memory, cells normally store a single partial value that may be refined, but not overwritten. The initial state of a cell is an "empty" partial value, in which no assumptions may be made as to the cell's contents. Propagators connected to the cell may forward an update to it to add to the partial value stored within.

Upon receiving an update to its contents from a neighboring propagator, a cell will apply an appropriate merge operation to unify the existing partial value with the new information in the update. These merge operations must not only be designed with consideration of the data-types being merged, but also the intended contents of the cell, so that a set intersection operation is not applied where a set union is needed.

For example, if we assume we are constructing a reasoner using propagator networks, a cell might wish to contain a set of facts about animals. Upon receiving a new set of facts about dogs via an update, the cell may merge it into its own set by taking the union of the new dog facts with those already known and set that as the new contents of the cell.

[1]DProp is currently available in a Mercurial repository at http://dig.csail.mit.edu/hg/dprop/.

While a naïve implementation might merge facts using set operations, we may use any arbitrarily interesting function for the merge operation. If a cell properly supported reasoning about its contents, we could eliminate previously known facts if they are subsumed by a new one. For example, explicit statements about two dogs, Rex and Fido, both having four legs might be able to be eliminated if a new statement stating that "all dogs have four legs" is learned. By allowing arbitrary merge operations, we can obtain greater power and expressivity in the propagator paradigm.

In order to ensure predictable computation in propagator networks, all merge operations must have four key properties:

1. **Idempotence:** The number of times a particular update is merged into a cell has no impact on the results of the merge. Knowing a fact and being told that fact again in an update doesn't mean we know anything new.

2. **Monotonicity:** Once a particular update is merged into a cell, no update should be able to directly unmerge that update. While the contents of a previous update may be marked as contradictory or disproven, it should not be possible to simply undo the update. In short, things once learned by a cell are not forgotten.

3. **Commutativity:** The order of merge operations in time is irrelevant. If we are told that "the cat is brown" and "the dog is black", we eventually know both facts regardless of order.

4. **Associativity:** Updating a cell with two separate update messages should have the same result as updating the same cell with a message that is the sensible "merge" result of the two. For example, if we are told that "Fido is at least 3 years old" and that "Fido is at least 5 years old", our knowledge should be the same as if we had just been told "Fido is at least 5 years old."

Each of these properties has influenced the structure of our system, and we will refer to them throughout the paper, summarizing their effects at the end of the paper.

These constraints are not as restricting as they may seem. Many functions that do not adhere to these constraints may be retrofitted to work in a propagator system. Much work in the design of distributed databases and internet protocols has attempted to satisfy demands for serializability of operations with the unreliability of the Internet, which may result in data being received out of order or not at all. By doing so, non-communtative operations may be performed despite the implicit commutativity of of packet ordering in the Internet [6, 5, 19, 18].

Once a cell's content has changed, any propagators that directly depend on the cell wake up and perform further processing, as in Figure 1. These propagators may send updates to additional cells, prompting yet more processing. For example, suppose we have a pattern-matching propagator attached to a cell containing facts about animals. This pattern-matcher will look at facts in that cell and then copy those of the form "x is brown" to another cell containing knowledge about brown animals. When some facts about dogs are merged into the first cell, the pattern-matcher will fire and any new knowledge about dogs that are brown will

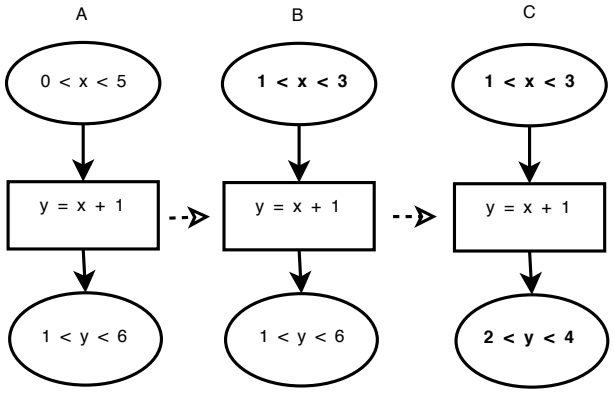

Figure 1: A propagator system at work with time increasing from A to C. Note that the updated constraints on x in the top cell in B cause the $y = x + 1$ propagator to wake up and update the constraints in the bottom cell in C.

be sent as an update to the "brown animals" cell. It will then merge that knowledge with the knowledge it already holds, may cause other propagators to fire, and so on.

Propagator networks may be cyclic, leading to complex loops and tail-recursive computation. As both single propagators and groups of propagators are separated by cells on their boundaries, we may also be able to treat a group of propagators as a single abstract propagator in its own right. In this way, we can actually instantiate them as needed, allowing for recursive processing.

The separation of computation from data in propagator networks explicitly modularizes computation. No time ordering is explicitly defined over the execution of propagators other than implicit dependencies caused by the order of cell changes. This makes propagator networks inherently concurrent and suggests a suitability to distributed systems.

3. THE REST ARCHITECTURE

Representational State Transfer, or REST, is an architectural style rooted in the concept of hypertext and the HTTP protocol.[10] The REST architecture aims to model resources as entities strictly capable of being created, read, updated, and deleted, also known as the CRUD operations. These operations are to be carried out through the transfer of *representations* of a resource. More complex operations may be modelled in a RESTful system in terms of these more fundamental resource operations. By restricting the number of operations that may be performed within the system, REST ultimately reduces API complexity.

Five primary constraints serve as the basis of a RESTful architecture, including:

1. a *client-server model* that separates data storage on a server from local display or handling of content

2. a *stateless design* such that any communication contains all information necessary for processing

3. *cacheability* of content to reduce network usage

4. *uniformity of interface* which reduces complexity of client implementations

5. and a *layered architecture* that helps to modularize network structure and treat it independently from the application structure.

Applications with a RESTful design are commonly implemented using HTTP, and ideally align the CRUD operations of creation, retrieval, updating, and deletion with the HTTP methods of PUT, GET, POST, and DELETE respectively.[2] For example, the content of a RESTful resource delivered with HTTP may be updated simply by submitting a POST request containing the information needed to update the resource.

By relying on the common HTTP protocol, RESTful applications are not only able to rely on existing software libraries, but also make themselves useful within the context of the World Wide Web. This allows applications to refer to external resources through the use of the HTTP URIs that identify them.

We choose to implement propagators using a RESTful architecture for a number of reasons:

1. Propagation maps nicely to a RESTful model. The stateless constraint on RESTful applications mirrors the inherent statelessness of the propagators themselves and the atomicity and associativity of cell updates make them particularly well-suited for implementation in a stateless architecture.

2. As cells are the only objects that contain state within propagator networks, it seems appropriate to model them as a class of resources in a RESTful architecture. This reduces the complexity of the implementation. This is only made more appropriate by the fact that only two primary operations, reading and updating, happen to cells.

3. The uniformity-of-interface constraint ensures that distributing propagators with RESTful techniques will be an easily expandable system. While our implementation, DProp, relies on Python, one may envision a a propagator network consisting of some nodes running Python on Linux, others using JavaScript in a web browser, and yet others using the .NET framework in Windows. By adhering to a RESTful architectural style, we ensure that such heterogenous systems are more easily constructed.

Together, these facts make REST a suitable architectural model for distributed propagation.

4. A RESTFUL PROPAGATION MODEL

As we intend to distribute the propagator model using RESTful techniques, we must first decide what the nodes in the physical network represent. We choose to treat hosts on the physical network as containers of both propagators *and* the cells that these propagators use to compute. In order to link hosts and properly distribute computation, we may simply use a simple network communication algorithm as a propagator designed to synchronize the contents of cells on

[2]Many variations exist, including those that account for web-browsers that may not support the PUT and DELETE methods, and several caused by confusion about the precise meaning of the PUT and POST methods.

multiple hosts. This may be done by simply forwarding any new updates observed by one cell to all other cells.

Two primary mechanisms for achieving such synchronization of cells seem to be likely candidate architectures for any such system:

1. We may choose to use a *client-server architecture*. In such a model, one host acts as the canonical server of a cell, in charge of performing merges and maintaining the canonical representation of the cell. All other hosts wishing to use the data stored in the cell act as clients, sending update messages to the server for its consideration. These hosts must remotely fetch data from the canonical representation following a successful merge.

   While this model simplifies the cost of maintaining the network with a simple star topology, use of a single canonical server means that this model is prone to failure of a single node (the server) causing a halt to all computation.

2. We may choose to use a *peer-to-peer architecture*. In this model, every host interested in a cell acts as a server for its own copy of that cell, and a client to other copies. Provided that there is a reliable method of both registering interest in a cell and forwarding updates to all hosts, any update to a cell should eventually be synchronized across all hosts.

   This model is more difficult to maintain, as it may have more complex network topologies, with a fully-connected network being the most efficient, but is not prone to complete failure caused by the failure of any one node.

   Choosing a peer-to-peer architecture *need not* imply that we must abandon the client-server constraint specified by REST. Instead, we may consider the peer-to-peer model to be an overlay model for how updates are propagated to remote cells. Similar to how web applications may invoke server-side methods that act as clients to other servers, we may employ a server and client acting in concert to manage a local copy of a cell to achieve the desired peer-to-peer propagation of updates.

While both models have their strengths and weaknesses, we believe that the gains to be found through decentralization, such as an increased tolerance of arbitrary network topologies and increased redundancy of connections, outweigh the costs of ensuring that synchronization will eventually occur in all hosts in a timely manner. We thus chose to implement DProp as a RESTful peer-to-peer-like system for cell synchronization.

4.1 Resource Representations

In constructing a RESTful system, we should note the kinds of objects modeled in our system as resources. Though propagators are generally stateless and need not be represented as resources, we do need to model the cells and the peers that have an interest in them.

The peer-to-peer model of synchronization introduces several complications to what would otherwise be an intuitive modelling of cells in a RESTful architecture. As mentioned previously, RESTful architectures assume the constraint of a client-server model. As a result, a resource must be identified uniquely only within the context of the server, rather than globally. HTTP somewhat alleviates this problem by providing a URI based not only on the local identifier but also the identifier of the server. Taken together, these two components may identify a resource globally.

This is ideal for our purposes, as we treat cells as being contained on individual hosts, rather than spanning them. This allows us to more easily identify distinct cells. A complication arises when we wish to associate the cell with a specific "synchronization propagator," which *does* span hosts.

If we were to simply implement this naming scheme, we would lack a way to easily identify which particular group of cells a cell is synchronizing with, short of listing those cells in the group. This means we would lack a method to describe relationships between different groups of cells and may unnecessarily complicate the process of manually analyzing a distributed propagator network.

We resolve this by giving each synchronization propagator a universally unique identifier (UUID)[14] so that distinct cells connected by such a propagator may be referred to with a single identifier. We include the UUID in the URIs of the copies of the cell, so we may identify the synchronization group that each representation belongs to.

In order to more easily add peers to a network, we must also represent each peer of a cell as a resource, or part of a resource. These resources must be able to be created or modified by new peers. Thus, each cell copy has a collection of "Peers" containing the URIs of other peers of the cell. This collection may then be modified as needed.

Each peer would only keep track of the peers it is interested in, so any of a number of arbitrary network topologies of the peers is possible. Although the propagator network needs only to be eventually connected, certain topologies are more preferrable to others: an increase in the number of links in the network will increase timeliness of update messages due to a reduced network diameter; it will similarly increase failure tolerance due to replication of updates sent on those links.

Our implementation is designed to maintain a clique. By assuming existence of a clique, we may eliminate the additional complexity of routing update messages and still ensure that the number of messages needed to propagate an update across the network is $O(n)$, the number of nodes. Even if a clique is not maintained, our protocol will still ensure convergence, although timeliness is not guaranteed for nodes farther than distance 1 from the updated node.

4.2 Initializing a Cell

Creating a brand new cell and making it globally available is rather simple. After creating the local storage for the cell and associating the corresponding merge operation with it, the host need only mint an UUID for the new synchronization propagator for the cell and assign it a URI. Local updates may happen immediately following creation of the local cell, while the URI of the cell as a resource must be made available before others may synchronize.

Joining an existing network of cells is less trivial. First, one must locate a copy of the remote cell to initialize from. We will assume that this has already been done and a URI has already been obtained. We make this assumption as we believe the process of peer discovery to be independent of the problems of initialization and synchronization.

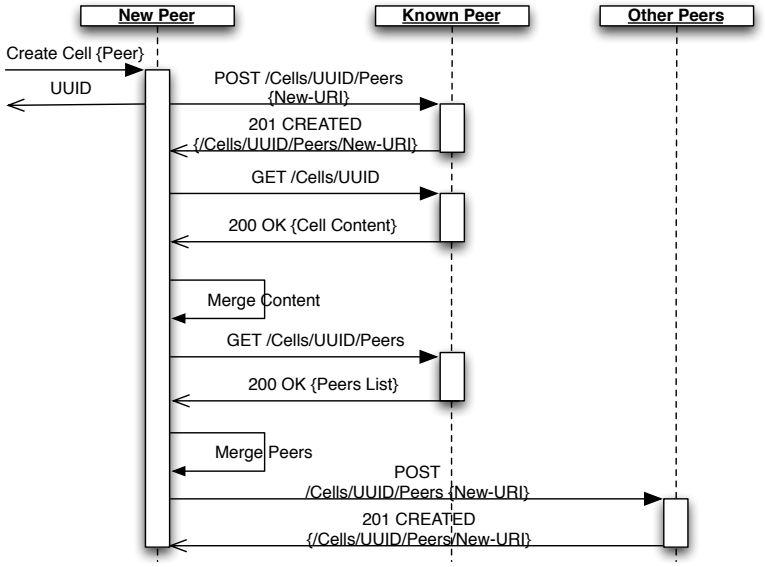

Figure 2: Connecting a new cell to an existing group of synchronized cells. Note that all actions are driven by the connecting peer.

Basic initialization follows a simple algorithm, depicted in Figure 2. A propagator wishing to connect to a remote cell will first create the cell as above, but assign it the UUID of the remote cell. Once the URI of the cell has been created, the cell submits a POST request to the collection of "Peers" of the cell held by the remote host. This POST request will contain the URI of the new peer's copy of the cell, and serves to add the peer to the network of peers "listening" to the cell. The remote peer will now be able to forward any updates to the new peer.

The copy of the cell is then synchronized with the remote peer by performing a GET request to the URI of the remote host's cell. The response will contain the contents of the cell on the remote host, and may be merged into the local copy of the cell. This may cause propagators local to the new peer to fire. Further GET requests are made to initialize the contents of the "Peers" collection from that known by the remote host. This completes the copy of the cell.

After the local peer copies the collection of peers, it submits a POST request to each peer in the newly updated "Peers" collection. This way, the new peer becomes visible to all other peers in the network, much as it did to the initial remote peer. These POST requests will result in the creation of a new, larger clique of synchronizing cells.

Note that we do not require any locking mechanism during the initialization of the cell. The associativity and idempotency constraints on the merge operation ensure that the merge of data from the remote peer will only increase the amount of knowledge known by the cell. The firing of local propagators following the merge also ensures that any change of the cell caused by the update will still generate meaningful computation once something is known by the cell.

4.3 Updating a Cell

As a propagator processes, it may send an update to another cell. It will do so by sending an update on the local host, which will then merge the new fact before causing any connected propagators to fire. In a distributed propagator network, the custom merge operation is actually used to complete a generic stub which additionally sends that fact as a POST request to all cell copies known in the local peer's "Peers" collection, as in Figure 3. In this way, the local peer that made the update also informs other peers about the new fact it has learned and lets them handle merging it.

After receiving a POST request representing the update, a cell will first confirm that the POST request came from a known peer in its "Peers" collection, so that facts cannot be tampered with by someone who is not trusted. If the update does come from a known peer, it will merge the newly received update message using the merge operation and, unlike local merging, will not do any further notification of nodes on the network. Finally, it will wake up any local propagators to perform any computation, just as if the update had been generated locally.

We assume that the merge algorithms on each peer are identical. This allows us to only transmit the updates across the network, rather than the fully merged data. Furthermore, the commutativity constraint on the merge operation implies that the order in which messages are received is irrelevant to the contents of the cell. This eliminates the need for synchronized timestamps across the entire network. Finally, the clique embodied by the interested peers ensures that the number of update messages sent is $O(n)$, the number of peers in the clique. This reduces communication within the system as well as the diameter of the network for iterative algorithms.

5. FAILURE RECOVERY

Although this basic system is functional at first glance, there are a number of potential points of failure in the system. The most obvious points of failure are links in the physical network. While use of HTTP over TCP makes the

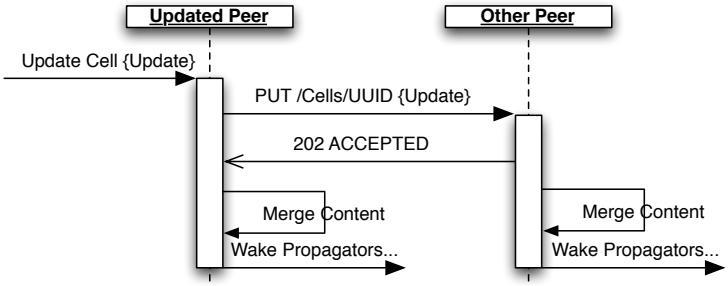

Figure 3: Process of propagating an update to a cell. Note that all actions are driven by the updating peer.

system somewhat tolerant of communication errors, routing irregularities and inconsistent uptime of connections must still be accounted for to ensure eventual convergence of the cell contents.

Recovery from such errors relies on the associativity and idempotency constraints on the merge operation. These allow for contents of cells to be merged without having seen the same update messages. Each peer interested in a cell will occasionally perform a full synchronization of its contents with all peers it is aware of. This synchronization consists of a number of GET methods applied to the remote peers' cells and "Peers" collections. The results of these GET requests may then be merged into the existing cell.

Synchronization in this way allows the local peer to determine connectivity to its set of peers. It also allows it to properly correct any possible loss of synchronization that may have been caused by messages that did not manage to reach the peer, without needing to resort to caching a history of all update messages on each node.

This operation is likely to be quite wasteful of bandwidth, as an identical resource representation may be obtained from each peer. The RESTful constraint of cacheability proves useful in reducing such waste. HTTP provides several headers that assist with client-side caching. Each HTTP resource may be served with an ETag, which uniquely identifies a particular state of a resource. When a resource changes, its ETag should change as well to uniquely identify the new state.

Requests made to an HTTP server may provide the If-None-Match header with a previously cached ETag value of the cached resource. If the ETag of the resource matches the If-None-Match header, the server may respond with a simple "304 Not Modified" message and no content. This indicates that the content in the client's cache remains the most recent version of the resource. We may thus use the ETag and If-None-Match header in our GET requests to ensure that data is only transferred when there is an inconsistency, rather than on every GET.

Should a link fail and a peer of the network lose connectivity, the loss of update packets between the peer and other members of the clique does not mean that processing is forced to halt. Assuming connectivity is achieved again, the above synchronization operation will suffice to converge the system. Furthermore, failure of a peer may simply be treated as if all links to the node have failed and no further processing is done. Reconnecting to the network will resolve any global inconsistencies that arise.

6. REDUCING RACE CONDITIONS

While race conditions are a natural byproduct of a concurrent system, our implementation of distributed propagation has several features that help to reduce the number of race conditions possible. The idempotency and commutativity constraints eliminate the harm from double updates and out-of-order updates, while several other race conditions are ameliorated by other means.

Most remaining race conditions are resolved through the synchronization mechanism. For example, suppose that a remote peer has not yet merged a new peer into its "Peers" collection. Should it receive a POST message from the new peer, it will drop the update as being from an unrecognized host. However, if the new peer has been registered with at least one other peer, the synchronization mechanism will ensure that the remote peer that dropped the update will not only eventually be made aware of the new peer, but also of the update that was dropped.

Despite the power of synchronization, there still exist several instances where inconsistencies may arise. For example, a remote peer may be down when the local peer attempts to connect, or the desired cell may not exist on that remote peer. In this case, the local peer is never able to properly join the network in the first place, and may be inconsistent with the network from the start. The addition of a mechanism that allows connecting to a cell through alternate peers, or simply retrying such connections may allow the local peer to connect to the cell anyway, after which the synchronization mechanism will resolve any inconsistencies between the disconnected cell and the other cells.

7. SECURITY

A practical distributed propagator system must be able to ensure the security of data within it. While a cell containing animal facts may unworthy of security, a similar cell could be used to share classified information which a government has a vested interest in keeping secure.

As our implementation uses HTTP as the substrate for operation, there are several possibilities for securing data. We may choose to secure the protocol through the use of SSL or TLS; we may also choose to secure the data by encrypting the contents of the cells themselves. We believe that a combination of the two approaches is necessary to achieve sufficient security.

Encryption of the protocol with SSL or TLS will defend communications against man-in-the-middle attacks and provide a mechanism for non-interactive authentication. How-

Constraint	Benefit
Idempotence	(with associativity) removes need for locking on cell initialization
	(with associativity) permits synchronization procedure to simply exchange knowns
Associativity	(with idempotence) removes need for locking on cell initialization
	(with idempotence) permits synchronization procedure to simply exchange knowns
Commutativity	removes need for global timestamps
	removes timeliness constraint on communications
Monotonicity	removes need to account for deletion of information
	allows for computation of results using partial knowledge

Table 1: Benefits of the four constraints on the merge operation within this synchronization system

ever, SSL/TLS itself is insufficient as multiple distinct propagators may use the same port on the server to host their cells. As SSL/TLS certificates are presented before the cell itself is requested, it is difficult to confirm that a particular cell is actually "maintained" by any particular user on the shared instance, such as in the following scenario:

Assume that Alice wishes to connect to an intermittently available cell operated by Bob. Bob's cell is hosted on a DProp peer that he shares with Eve. If Eve was aware of the times when Bob's cell was unavailable, she could create a cell with the same URI and wait for updates from Alice. Alice would be unable to distinguish whether the cell had been created by Bob or by Eve, as the only mechanism for authenticating the DProp peer is a server-wide certificate. This means that the same identifying certificate is provided to Alice upon connecting to the DProp peer, even if Bob and Eve were to have separate client certificates for when they sent updates to Alice.

There are a number of ways to resolve this problem. We may enforce a permanent reservation system for cell identifiers, or instead have a secondary identifier used to contain information about the cell's owner. The most foolproof method would be to encrypt the updates themselves. By encrypting them, Alice and Bob could ensure that only the propagator with the correct key would be able to decrypt updates, despite the limited level of security granularity offered by SSL/TLS.

While this double encryption requires additional overhead, we cannot simply do away with SSL/TLS either. Without SSL/TLS, the identity of the cells contacted could not be encrypted without abandoning HTTP. By adding SSL/TLS as an additional security layer below HTTP, we can ensure that this meta-data is not subject to man-in-the-middle attacks. As a result, we must perform double encryption if we are to maintain compatibility with the HTTP standard.

8. RELATED WORK

The work presented in this paper bears strong similarities to work done in the field of database replication. Basic database replication approaches would be sufficient for a cell synchronization system if we assumed that cells are nothing more than simple databases with data rows that correspond to the updates received by the cell. However, such techniques tend to adhere to constraints that are unnecessary for propagator networks, such as serializability, used in weighted voting [12], and the non-monotonicity of row deletions.

Update propagation mechanisms for database replication, such as those developed for the Grapevine system [5] are similar to the mechanisms described here. Like the synchronization mechanism above, Grapevine seeks eventual convergence of knowledge in the network, rather than guaranteeing immediate convergence. Unlike Grapevine, however, we integrate the database more tightly with the messaging system on a single host, rather than implicitly allowing for their separation. Furthermore, our system has an explicit mechanism for resolving inconsistencies as part of the synchronization protocol.

The Bayou system [19] is in some ways more similar to our system than Grapevine. Unlike Grapevine, it guarantees convergence on a weakly-connected network. Nevertheless, its guarantee of eventual serializability of updates leads to the construction of a centralized system of "soft writes" which our system neither requires nor implements. Furthermore, Bayou again concerns itself with the problem of non-monotonicity caused by deletions which is not required for cell synchronization.

Singhal [18] provides an algorithm that is similar to the synchronization protocol described here, allowing for the synchronization of replicated databases through update distribution. However, like in Bayou, Singhal also adheres to a serializability constraint that is unneeded for propagators. Furthermore, Singhal does not assume a weakly-connected network as DProp does.

Decentralization of RESTful practices has been proposed in Khare and Taylor's ARRESTED architecture [13]. While its principles provide an important basis for our implementation, the ARRESTED architecture ultimately implements several features that are unnecessary for a propagator network, such as estimation, locking, and routing. As propagator networks are constrained to be monotonic, there is no need to guarantee receipt of an update or to estimate a cell's content.

9. CONTRIBUTIONS & FUTURE WORK

We have implemented and demonstrated a RESTful data propagation system that permits useful distributed computation with a weakly-connected network. In doing so, we avoid constructing a centralized model that is subject to point failures. It is our belief that a system such as this will prove to be a viable computational platform that provides a greater flexibility than that offered by traditional client-server architectures for distributed computing like that of BOINC or other similar grid architectures.

We have also identified four constraints of the merge operation used in propagator networks, idempotence, monotonicity, commutativity, and associativity. These constraints give us a number of advantages, described in Table 1, that allow us to simplify our system and provide greater redundancy and flexibility than existing approaches to distributed databases.

Although DProp has been tested in small propagator networks to demonstrate the feasibility of RESTful distributed propagation, we have not yet tested it in larger networks. We also have not yet fully implemented the security protocol described in this paper and tested it within a working propagator network. In order to test the system with a large-scale application, we currently intend to implement an application that manages information sharing with provenance using the DProp framework.

We believe a more complete analysis of the proposed security mechanisms is warranted following implementation of the security component in DProp, as we have only performed a cursory inspection of possible security concerns with the proposed system at this time. This analysis may require migration away from a strict HTTP implementation to avoid the double encryption issue mentioned above, as well as any other issues that arise with the use of SSL for this protocol. We are also unclear on the overhead that an HTTP client-server model incurs for this implementation, although initial tests of DProp have not exhibited significant issues with overhead when propagating small updates.

10. ACKNOWLEDGEMENTS

We would like to thank Lalana Kagal, Gerry Sussman, Hal Abelson and other members of the Decentralized Information Group at MIT for their advice and criticism throughout the entire process of refining the ideas discussed here and putting them down on paper. We would also like to acknowledge that this work was supported in part by the National Science Foundation under NSF Cybertrust Grant award number 04281 and by IARPA under Grant FA8750-07-2-0031.

11. REFERENCES

[1] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[3] B. Awerbuch and S. Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 278–281. ACM, 1984.

[4] R. E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, Princeton, NJ, USA, 1961.

[5] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[6] V. G. Cerf and R. E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, May 1974.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*. USENIX Association, 2004.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM, 1987.

[9] R. B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, January 1995.

[10] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] C. L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, February 1979.

[12] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–162. ACM, 1979.

[13] R. Khare and R. N. Taylor. Extending the representational state transfer (REST) architectural style for distributed systems. In *Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.

[14] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN namespace, July 2005.

[15] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: A review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

[16] A. Radul. *Propagation Networks: A Flexible and Expressive Substrate for Computation*. PhD thesis, Massachusetts Institute of Technology, 2009.

[17] A. Radul and G. J. Sussman. The art of the propagator. Technical Report MIT-CSAIL-TR-2009-002, MIT Computer Science and Artificial Intelligence Laboratory, January 2009.

[18] M. Singhal. Update transport: A new technique for update synchronization in replicated database systems. *IEEE Transactions on Software Engineering*, 16(12):1325–1336, December 1990.

[19] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–182, Copper Mountain, CO, USA, 1995. ACM.

[20] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using MapReduce. In *Proceedings of the ISWC '09*, volume 5823 of *LNCS*. Springer, 2009.

Developing a ReSTful Mixed Reality Web Service Platform

Petri Selonen, Petros Belimpasakis, Yu You
Nokia Research Center
P.O. Box 1000
FI-33721 Tampere, Finland
+358 718 039 052
{petri.selonen, petros.belimpasakis, yu.you}@nokia.com

ABSTRACT
This paper discusses the development of a ReSTful Web Service platform for serving Mixed Reality content at Nokia Research Center. The paper gives an overview of the Mixed Reality domain, the requirements for the platform and its implementation. We further outline a method for developing resource oriented Web services, beginning with high-level requirements, formalizing them as UML models and refining them to a ReSTful Web service specification. The approach is demonstrated with detailed examples of designing one particular Web service subset for Mixed Reality annotations.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design – *methodologies, representation*. D.2.11 [**Software Architectures**]: Service-oriented architecture. D.2.12 [**Interoperability**]: Interface definition languages

General Terms
Design, Documentation, Experimentation

Keywords
ReST, Mixed Reality, UML, Web Engineering

1. INTRODUCTION
Mixed Reality (MR) refers to the fusion of the real and virtual worlds for creating environments where physical and digital objects co-exist, encompassing both Augmented Reality and Augmented Virtuality. It has been a research topic for over a decade at Nokia Research Center.

Up to date, typical MR research prototypes have either been stand-alone applications without linkage to centralized content repositories, or in rare cases, had proprietary backends with closed Application Programming Interfaces [5]. A few selected trials have integrated with third party media content repositories like Flickr, but such repositories do not offer built-in support for Mixed Reality content. The MR applications have effectively been built as one-off applications in silos with little sharing of content. One exception to the rule is Layar[1] which allows 3rd party content providers to serve content as layers through a HTTP API that mobile AR browser then renders on top of a view-through camera. Another similar service is Junaio[2], which allows adding new points of interest to 3rd party channels. Both services allow adding new content but do not offer APIs for others to build on.

To avoid having to re-invent the backend for every new service, we started developing a Web Service platform for serving Mixed Reality related content for MR applications and solutions. The platform, hence referred to as MRS-WS, has geo-spatial relations in the core of its design, meaning that special attention is placed on searching and storing content with geographical location and spatial arrangement information. The system not only stores standard multimedia content with metadata, but also other content like 3D models, point clouds, street-view panoramas, building models, points of interest (POIs), terrain morphology and road networks.

In addition to serving basic MR content, the platform also serves advanced differentiating features like unique geo-data extracted and post-processed from Navteq[3] raw content and data extracted from user generated content like pointclouds. We apply advanced computer vision algorithms to detect buildings visible in particular panoramic images and eventually enabling linking digital content with physical objects *in situ*.

ReST [2] and Resource Oriented Architecture was chosen as the architecture style for the platform because we essentially serve content: the value of the service is in storing, retrieving and managing interlinked content. However, so far there does not exist a commonly agreed, systematic and well-defined way to proceed from a set of requirements to an implemented ReSTful Web service.

Based on the above requirements and our previous research on developing ReSTful Web Services, we utilized a systematic but light-weight method for proceeding from new requirements to an implemented ReST API, and integrating this with the existing system. Further, to be able to communicate with the different stakeholders, we utilize UML models to capture the system requirements.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

[1] http://layar.com/

[2] http://www.junaio.com

[3] *NAVTEQ* (http://www.navteq.com/) creates the digital maps and map content that power navigation and location-based services solutions around the world

In what follows, we discuss the development of the MRS-WS, outlining its development mode, the modeling aspects and some of its design details.

2. DEVELOPING MRS-WS REST APIS

The MRS-WS platform is build within the context of a larger NRC Mixed Reality Solutions program. Figure 1 shows a concept of a mobile MR client currently under development on top of the platform. The client fetches panoramas, building outlines and points of interest based on a location, and visualizes annotations and related comments belonging to the user overlaid on top of a panoramic image or a see-through camera view. The application enables the users to annotate particular buildings and other landmarks through touch and share these annotations with other users.

Figure 1. Example MR client concept

The research program entails some of the challenges from both research and product development environments: need to serve dedicated first-priority clients, high internal visibility, innovation and prototype driven mode, requirement for short time-to-release cycles, short sprint iterations, multi-disciplinary, multi-site and multi-continent teams spanning from Tampere and Helsinki to Palo Alto and Chicago, using subcontractors, having several stakeholders and finally operating in a constantly changing operating environment. At the same time, because of the research innovation mode, both the available content like post-processed assets from Navteq data and the client requirements keep on changing.

In particular, such a program mode implies a need for a very pragmatic approach for architecting the service platform. When client requests come in, they must be fulfilled. Engineering for serendipity is a virtue, but MR domain does not have commonly agreed content types to begin with. Instead, we tried to make sure that the architecture would not prevent enhancing interoperability support in the future.

The platform should support both mobile and desktop clients, it should be secure and scalable, it should manage not only MR enabled content but also non-MR related elements like social connections, context, maps, and other such information, and it should provide a unified access to all MR clients to all this content. We also need to support 3rd party developers and rich mash-ups to enable open innovation and ecosystem building.

Some of the additional benefits expected from choosing ReST include decoupling the clients from the service to support both high priority program clients and 3rd party clients in the future, uniform interface to enable evolution of the platform to support new content types over time and aligning with Nokia Services business unit reference stack. We have also previously explored using ReST with mobile clients, and the lessons learned from these exercises were taken into use in the platform building exercise.

The MR domain exhibits *a priori* resource oriented characteristics. However, proceeding from requirements to an implemented ReSTful Web service is still not a trivial exercise. Probably the best known formulation of designing ReSTful Web services has been presented by Richardson and Ruby which can be summarized as follows ([6], pp. 109 and 148):

- figure out the data set, split it to resources and interconnecting links, and assign URIs as unique identifiers to the resources;

- select the suitable uniform operations and response codes; and

- define the representations for each resource.

The rest is for the service architect to figure out herself. The above formulation or the original nine steps are too abstract to be followed as a method. The upcoming ReSTful Web Services Cookbook [1] is a promising effort to describe many of the aspects related to building a ReSTful Web Service. Its recipes, however, are not presented as patterns as such that could be organized into a pattern language and into a process. Further, to be able to communicate with the different stakeholders, there is a need for some commonly understood modeling notation.

In our previous work (Laitkorpi et al [3][4]) we have explored how to devise a model driven process for developing ReSTful service APIs. In short, the proposed process proceeds from a set of service requirements to a service specification and ultimately an implementation. The work explored in particular the paradigm shift from a functional specification, expressed in terms of arbitrary actions, to a resource-oriented, descriptive state information content, and how to support this process with model transformations. We took the learnings of this previous work to construct a more lightweight and agile process, suited for the Mixed Reality program and its a priori content oriented domain.

Once the requirements for the new service (i.e. MR content subset) had been collected from program customers through meetings, email exchange and other informal communication, they were written down as an initial service specification. This specification was then formalized into an Information Model, a simple structure model expressed as Unified Modeling Language (UML) class diagrams. By following a set of simple rules, this model can be further transformed into a Resource Model and finally into an implementable specification, either using e.g. Web Application Description Language (WADL) or some other description language, or as in our case, by mapping the entities to our existing Java classes based on the Java EE–Hibernate–Restlet[4] technology stack.

3. MRS-WS DOMAIN MODEL

Figure 2 shows a high-level domain model describing some of the main concepts of the MRS-WS system together with their attributes and relationships. The concepts can be described as follows:

[4] http://www.hibernate.org; http://www.restlet.org

- Location. Supports geo-spatial arrangement of content required by Mixed Reality applications. Central element type that MR enabled content types can utilize. In addition to basic (x,y,z) coordinates, also supports spatial arrangement (roll, bearing, pitch), required for representing an object in the 3D space .
- Content. Describes content that can be managed by a user. Content element can have attributes (name-value pairs), textual tags, comments, and Access Control List items.
- User. A MRS user that can own and manage MRS-WS content.
- Annotation. Content type that can be used to annotate buildings and general locations with textual description and link to other content like photos.
- 3D Model. Geometrical objects in 3D space like pointclouds and volumetric models describing eg. three dimensional building outlines.
- Photo. The inevitable photo content type. Supports thumbnails in addition to the full sized media.

In addition to the user generated content above, there are also read-only content types that are pre-created based on Navteq data:

- Panorama. 360 degrees street view images taken on a location. With advanced post-processing, panoramas link to buildings and POIs visible in the picture.
- Building. Information on buildings and their outlines and 3D model.
- POI. Points of Interest with information on their category (conveniently ranging from wineries to hotels to border crossing) and address.
- Terrain. Map tiles containing terrain information, that describes the earth's morphology.

In what follows, we give a detailed example of how one particular service subset can be developed, before summarizing the whole MRS-WS API.

4. EXAMPLE: ANNOTATIONS

To illustrate the design process from requirements to an implemented Web service, we take a deeper look into one particular Web service subset for Annotations.

An Annotation is a Mixed Reality annotation that can comprise a title and a textual description, as well as links to other content elements like photos. Essentially an Annotation is a way of the users to annotate physical entities with digital information and share these annotations to groups of friends or to the general public at will.

In addition, Annotations can include physical address and a category, and this way allow the public to describe new places in addition to pre-defined POIs. As is apparent from Figure 2, Annotations and POIs share similar information, including the postal address and a category. This redundancy is one symptom of aggregating content from different data sources and is a possible subject for future reconciliation.

4.1 Annotation Requirements

Requirements for the Annotations were provided by a program client team after a few rounds of iterations. Some of the structural requirements were already implied by Figure 2, namely that an Annotation

R1. has support for a location and spatial arrangement in 3D space, for user generated comments and for an owning user;

R2. has properties title, a description, creation and modification timestamps, street address and view count;

R3. can be searched by location in a bounded box;

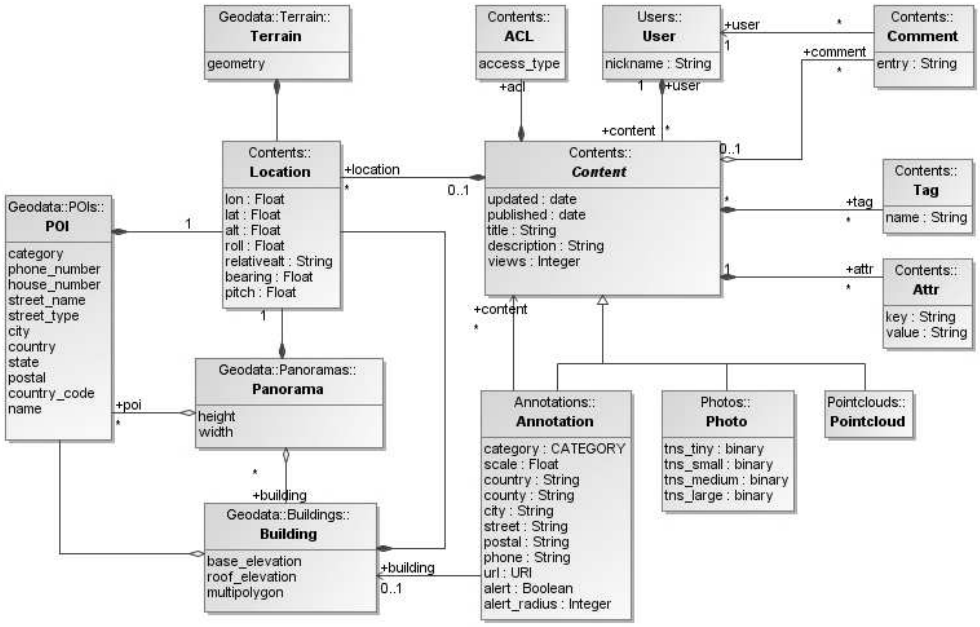

Figure 2. Partial MRS-WS domain model

R4. can be searched by location within a given radius;

R5. can be searched based on links to a particular building;

R6. can be searched based on selected categories only;

R7. can be searched based on most viewed elements;

R8. can be set to have been viewed by the client;

R9. can be attached content (eg. multimedia, binary, pointcloud) ; and

R10. can be arranged spatially based on a normal vector.

4.2 Annotation Information Model

Some of the above features were already present in the domain model shown in Figure 2. A refined information model fragment for Annotations is shown in Figure 3.

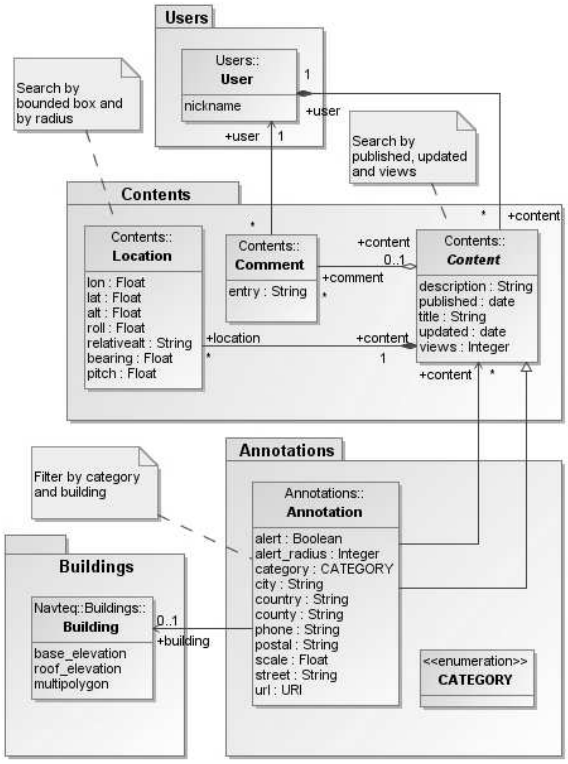

Figure 3. Annotation information model

In the figure an Annotation

- is a concrete instance of a Content element, thus inheriting attributes like updated, published, title, description and views, and relationships to Location, User, Comment, Tag and Attr;
- has a set of its own attributes (category, scale, address information, alert and alert radius);
- has a reference to (at most one) Building; and
- has references to zero or more other Content elements (with a constraint of not including other Annotations omitted).

The search criteria is shown as informal notes attached to the respective elements, marking them to be properly stereotyped later for more formal treatment.

4.3 Annotation Resource Model

A resource model re-organizes the elements of a domain model to addressable entities that can be more easily mapped to resources of a ReSTful Web service while still being readable and compact. The Resource Model concept is adapted from Laitkorpi et al [4]. Figure 4 shows a partial Resource Model profile.

The concepts of the Information Model become resources; depending on their relationship multiplicities, they become either Items or Containers containing Items. Composition relationships form resource–subresource hierarchies that are reflected by URI paths, while normal and aggregate associations become hypermedia references and collections of references between resources. Attributes are used to generate resource representations. Candidates for using standard MIME types are singled out when possible.

Attributes with special semantics are marked with respective property stereotypes, like «id» for attributes defining the identifier of the resource, or «appendable» defining that an individual attribute can be appended a value (using POST).

More specifically, each concept (ie. classifier) in the domain model becomes an Item that represents a basic addressable unit of information in the system that can by default be created, retrieved, modified and removed. Containers are used for managing collections of elements allowing their creation and retrieval. Each search or filtering feature defined in a note is represented by a projection is a view to a container, either representing a query or a pre-defined filtering action.

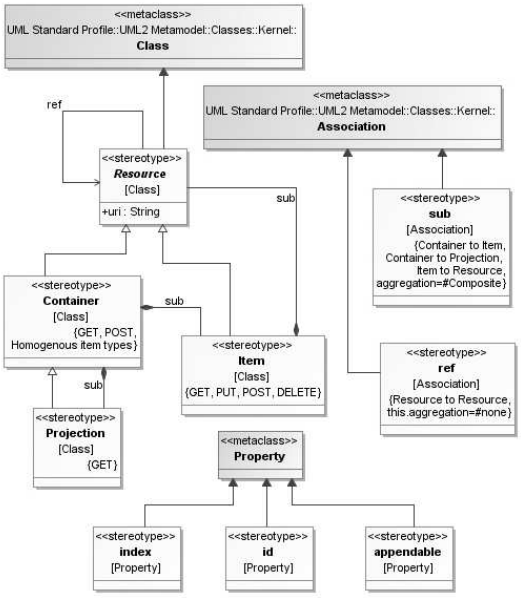

Figure 4. Resource Model profile (non-normative)

The underlying assumption is that the composition hierarchy of the domain model, represented by a UML structure model, naturally defines the primary decomposition of the resource hierarchy. The alternative paths through non-composite containers

will represent other hypermedia references to resources. The subresource composition associations and reference associations are marked with «sub» and «ref» stereotypes, respectively.

4.3.1 Annotation resource and representation

Figure 5 shows an example fragment of Annotation Resource Model. It shows how concrete Annotation element is divided into an Annotations container containing Annotation resources, and similarly Building element divided into Buildings container containing Building items. An Annotation can refer to a set of Content elements. Note that the concept of User owning Content in Figure 3 is omitted for brevity. In practice, this will cause explicit user ownership scoping in the URIs (ie. **/users/{user.nickname}**).

The resources implied by Figure 5 are as follows (starting with whatever path prefix MRS-WS system implies):

/content

/content/annotations

/content/annotations/{annotation.id}

/buildings

/buildings/{building.id}

The generalization relationship between Annotation and Content, the attributes and the subresource relationships are interpreted as cues for resource representation.

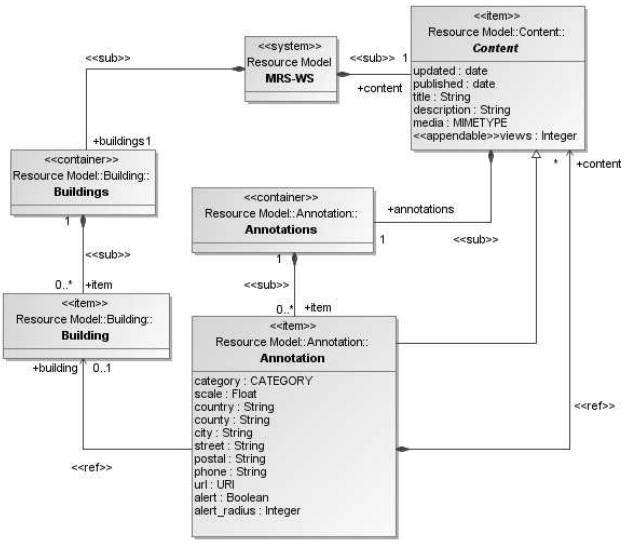

Figure 5. Annotation Resource Model fragment

The resource model and applied content types should be mapped to standard MIME types whenever possible. By default, a new content type is created with a template using the resource name. The content description is based on the resource attributes and attributes inherited from generalized resources. Example of Annotation representation is given in Figure 6.

```
<annotation href="...">
  <id>4195042682</id>
  <updated>2009-12-18 04:01:13.0</updated>
  <published>2009-12-18 04:01:02.0</published>
  <title>For rent!</title>
  <description>I'm renting a 60m2 apartment here.
        Call me if interested.</description>
  <media />
  <views>47</views>
  <category>1</views>
  <scale>1.0</views>
  <country>Finland</views>
  ...
  <alert>false</ alert >
  <alert_radius>0</ alert_radius >
</annotation>
```

Figure 6. Example content type for Annotation (application/vnd.research.nokia.annotation)

Each resource must have an attribute with a unique value that is used for creating the resource URI. Such attributes are marked with «id» stereotype. If none such attribute exists, we automatically create a field "id" for this purpose.

4.3.2 Annotation subresources

In addition to the attributes, Figure 5 also shows that an Annotation has subresources: it links to at most one Building, and several Content elements. Figure 7 shows another resource model showing content subresources. The implied resources are:

/content/comments

/content/comments/{comment.id}

/content/locations

/content/locations/{location.id}

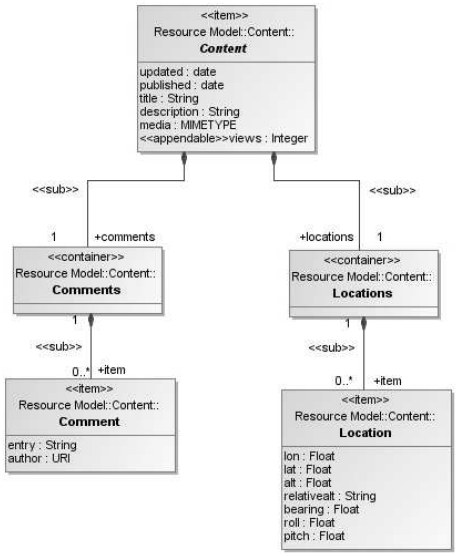

Figure 7. Resource Model for Content subresources

The subresources and referenced resources can be included to resource representation through inlining. To inline Annotation resources Location and Building, we use the following HTTP header:

x-mrs-deco: inline(locations,building)

An example of resulting representation is given in Figure 8.

```
<annotation href="...">
  <id>4195042682</id>
  ...
```

```
<locations href="...">
  <location href="...">
    <lat>61.44921</lat>
    <lon>23.86039</lon>
     ...
    <pitch></pitch>
  </location>
</locations>
<building href="...">....</building>
...
</annotation>
```

Figure 8. Example of inlined content for Annotation

4.3.3 Annotation projections

Figure 9 shows another annotation resource model fragment, describing searches. For each search feature described in an information model, a new container projection is generated together with references to the attributes and subresources used as projection criteria. Projections return references to items contained by a container. They do not have subresources themselves.

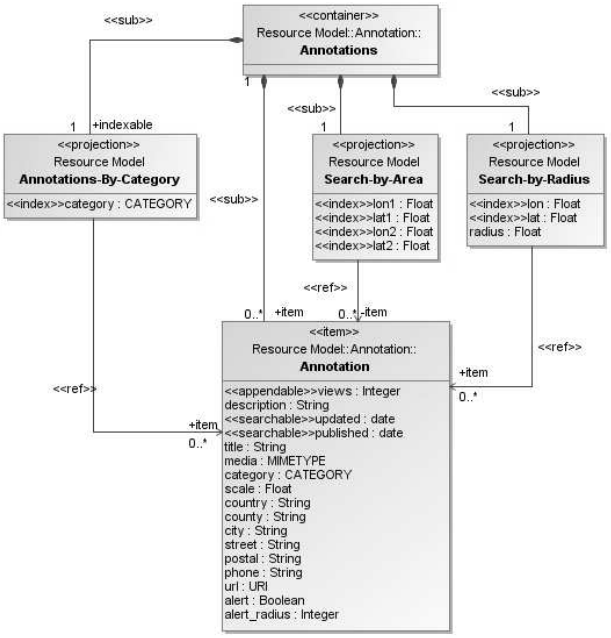

Figure 9. Annotations Resource Model Fragment for searches

The three projection container resources implied by Figure 9 are:

/content/annotations?

category={annotation.category}

/content/annotations?

lon1={location.lon}&lat1={location.lat}&

lon2={location.lon}&lat2={location.lat}

/content/annotations?

lon={location.lon}&

lat={location.lat}&radius={Float}

Some common and often used projections can be promoted to pre-defined filters. This is especially the case with projections that do not require user to specify parameters. This way the resources become addressable and in the case of public data, cacheable. Examples of such projections are:

/content/annotations;filtered/recent

/content/annotations;filtered/most_viewed

The former returns the annotations sorted based on the content.updated attribute, while the latter returns the annotations sorted with the most content.views attribute.

4.4 Requirements Revisited: Using the Service

To ensure that the implemented service fullfils the requirements, we give examples for each specified feature. The basic functionality, ie. creating, retrieving, modifying and deleting Annotations are handled in the obvious way.

R1. Yes. See Location and Comments subresources, and User reference (omitted) above.

R2. Yes. See representation example above.

R3. **GET /content/annotations?**
lon1=7.45&lat1=46.955&lon2=7.452&lat2=46.956

R4. **GET /content/annotations?**
lat=61.4467&lon=23.8575&radius=0.5

R5. **GET /content/annotations?building=usa_chi123**

R6. **GET /content/annotations?category=1,3**

R7. **GET /content/annotations;most_viewed**

R8. **POST /content/annotations/123/views**
3

R9. **POST /content/annotations/123/contenturis**

```
<contenturis>
<contenturi href="..." />
</contenturis>
```

R10. **GET /content/annotations/123**

```
<annotation href="" >...
  <location href="">
    <bearing>...</bearing>
    <roll>...</roll>
    <pitch>...</pitch>
  </location>
</annotation>
```

Client handles based on the bearing, roll and pitch attributes of associated Location.

4.5 Implementation notes

To realize the ReSTful Web service implied by the Resource Models, we map them to implementation level concepts. We have implemented most of the basic functionality related to the Resource Model on top of Restlet with the concepts of Resources, Containers and Items implemented as Java classes—ContainerResource and ContainerView, ItemResource and ItemView—and specialize them for each new service. Figure 10 gives a simplified view on how the Annotation service is implemented. There obviously is complex underlying machinery for implementing all the necessary infrastructure related to eg. security, user management, scalability, database access, optimizing searches, aggregating content and so forth. However, our experiences suggest that the mapping from new domain

models and APIs to implementation can be rather straightforward once the required framework is up and running.

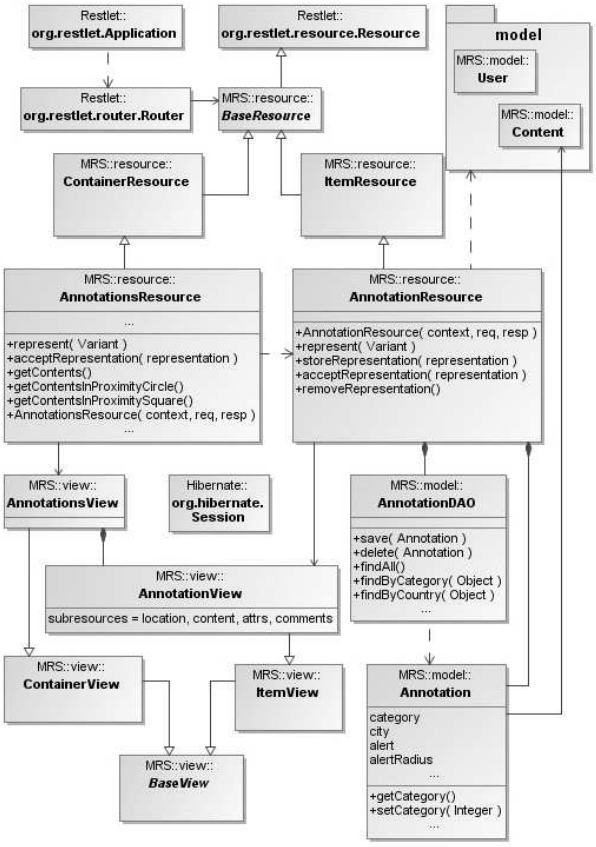

Figure 10. Annotation service implementation

5. MRS-WS ARCHITECTURE

Figure 11 shows a high-level architecture summary of the MRS-WS system. Resources from the following five main sources are offered through one unified ReST API:

- Geodata and other post-processed content from Navteq;
- Mixed Reality content like 3D objects, as well as other multimedia content and related metadata served on our own content repository;
- Multimedia content like photos and video aggregated from external content repositories together with possible metadata;
- Identity and social for user authentication; and
- Social networks like user's friends and groups.

A summary of the current MRS-WS service is given in Figure 12. By adhering to the ReSTful architecture style and uniform interface, the core of the service specification can be captured by a resource model.

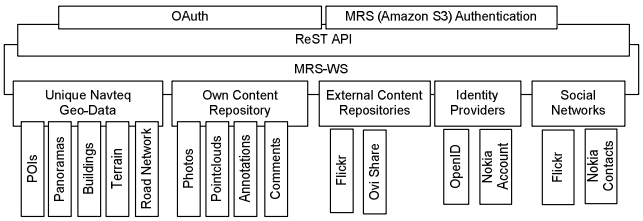

Figure 11. MRS-WS architecture

6. CONCLUDING REMARKS

This paper gave a brief overview of developing a ReSTful Mixed Reality Web Service platform at Nokia Research Center. We introduced the MR problem domain and identified a need for having a lightweight method to support systematic development of ReSTful Web Services from requirements to implemented system.

According to our experiences, most ReSTful APIs look deceivingly simple while the simplicity does not make developing them particularly easy. The situation is not helped by most of the publically available APIs claiming to be ReSTful effectively being of the RPC {resource}.function() variety, like Digg[5], Flickr[6] or Yahoo![7]. These services are obviously very useful but not very Resource Oriented nor ReSTful. The few exceptions like Amazon S3[8] or Google Search are good but overly simple in their structure, or more exemplary like the map service of [6] or the RESTify DayTrader[9].

There are several important topics beyond the scope of this paper: formalizing the approach and building supporting tools, evaluating the quality and not just the resource oriented characteristics of the produced APIs, providing support for both the ReST idioms we have identified together with the Allamaraju and Amudsen recipes [1], transforming arbitrary functional requirements into explicit resource manipulations, supporting synchronization of the domain model with resource models and database schemas, and aggregating content and other data from both Nokia systems and 3rd party services. Similarly, the specifics of content control, inlining, verbosity, pagination, compression and access control, and support for mobile clients have been omitted.

Nevertheless, we believe that the ReSTifying approach applied during the development is one step towards coining software engineering practices for ROA based Web Services development in real-life development environment. Based on our experiences, we believe the approach can support service architects' communication with the service clients, and perhaps more importantly, with software engineers not familiar with ReST and Resource Oriented Architecture. The presented approach supports bringing in new features and APIs to the system in an unintrusive way in a rapid pace while supporting traceability of requirements and actual implementation.

---

[5] http://digg.com/api/docs/overview

[6] http://www.flickr.com/services/api/

[7] http://upcoming.yahoo.com/services/api/

[8] http://docs.amazonwebservices.com/AmazonS3/latest/API/

[9] http://bitworking.org/news/201/RESTify-DayTrader

7. ACKNOWLEDGMENTS

The authors would like to thank Arto Nikupaavola, Markku Laitkorpi, Brenda Castro, the NRC Mixed Reality Solutions program and the former NRC Service Software Platform team for their valuable contribution.

8. REFERENCES

[1] Allamaraju, S., and Amudsen, M. 2010 RESTful Web Services Cookbook. O'Reilly.

[2] Fielding, R. T. 2000 Architectural Styles and the Design of Network-based Software Architectures. Doctoral Thesis. University of California, Irvine.

[3] Laitkorpi, M., Selonen, P., Siikarla, M., and Systä, T. 2008 Transformations Have to be Developed, ReST Assured. In Proceedings of International Conference on Model Transformation (Zürich, Switzerland, July 1 - 2, 2008). ICMT2008. ETH.

[4] Laitkorpi, M., Selonen, P., and Systä, T. 2009. ICWS. Towards a Model Driven Process for Designing ReSTful Web Services. In Proceedings of the International Conference on Web Services (Los Angeles, CA, USA, July 06 - 10, 2009). ICWS'09. IEEE Computer Society.

[5] Luo, X. 2009 From Augmented Reality to Augmented Computing: A Look at Cloud-Mobile Convergence. In Proceedings of the International Symposium on Ubiquitous Virtual Reality (Gwangju, South Korea, July, 2009). ISUVR'09.

[6] Richardson, L., and Ruby, S. 2007 ReSTful Web Services. O'Reilly.

The basic URI structure:	<ownership>/<resource>;<projection>.<format>?<query_string>
Proximity geosearching:	<container>?lon={}&lat={}&radius={}
Bounded box geosearching:	<container>?lon1={}&lat1={}&lon2={}&lat2={}
Time based searches:	<container>?updated_since={}&updated_until={} or <container>?taken_since={}&taken_until={}
Property based filtering:	<container>?{resource.parametername} ={}
Inlining of subresources:	x-mrs-deco: inline({resourcename},{resourcename})
Setting verbosity of resources:	x-mrs-api: verbosity({id|summary|full})
Pagination:	x-mrs-api: pagination({true|false})

Resource URI /mrs/rest	Description	Operations	Default status codes	Inline elements
/panos	Panoramas	GET	200, 401	
/panos/{pano.id}	Specific panorama item	GET	200, 401, 404	buildings, pois, locations
/buildings	Buildings	GET	200, 401	
/buildings/{building.id}	Building foot print outlines	GET	200, 401, 404	obj, pois, locations
/terrain	Terrain tiles	GET	200, 401	
/terrain/{terrain.id}	Individual terrain tile	GET	200, 401, 404	obj, locations
/pois	Points of Interest	GET	200, 401	
/pois/{poi.id}	Individual POI	GET	200, 401, 404	-
/users/{user.nickname}/content/	All user content	GET	200, 401, 404	pointclouds, annotations, photos
/users/{user.nickname}/content/pointclouds	Pointclouds	GET, POST	200, 201, 400, 401	
/users/{user.nickname}/content/pointclouds/{pointcloud.id}	3D mesh	GET, PUT, DELETE	200, 400,401, 404	media, locations, comments, tags
/users/{user.nickname}/content/annotations	Annotations	GET, POST	200, 201, 400, 401	
/users/{user.nickname}/content/annotations/{annotation.id}	Annotation	GET, PUT, DELETE	200, 400,401, 404	locations, comments, tags, building
/users/{user.nickname}/content/photos	Photos	GET, POST	200, 201, 400, 401	
/users/{user.nickname}/content/photos/{photo.id}	Photo	GET, PUT, DELETE	200, 400,401, 404	media, locations, comments, tags
/users/{user.nickname}/comments/	Comments	GET, POST	200, 201, 400, 401	
/users/{user.nickname}/comments/{comment.id}	Individual comment	GET, PUT, DELETE	200, 400, 401, 404	-
[parent resource]/tags/	Tags	GET	200, 401, 404	
[parent resource]/tags/{tag.name}	Individual tag	(GET), PUT, DELETE	200, 400, 401	-

Figure 12. MRS-WS API overview

A RESTful Architecture for Adaptive and Multi-device Application Sharing

Vlad Stirbu
Nokia Research Center
vlad.stirbu@nokia.com

ABSTRACT

In this paper we introduce a practical approach to share the user interface of MVC compatible interactive applications with remote devices that have the ability to adapt the user interface to their specific look and feel. We present the system architecture and the methodology to model the user interface as a set of RESTful resources. The remote user interface and the application state are synchronized using an Web-based event-driven system.

Categories and Subject Descriptors

D.2.2 [**Software**]: Design Tools and Techniques—*User interfaces*; D.2.10 [**Software**]: Design—*Methodologies*; D.2.11 [**Software**]: Software Architectures—*Patterns*

General Terms

Design, Experimentation

Keywords

REST, event driven system, user interface adaptation

1. INTRODUCTION

Sharing the screen of an application provides an easy way to remotely access and control applications from other devices. The increased capabilities of mobile devices have challenged the role traditionally held in this context by desktop and laptop computers. People expect to have access to applications running on remote devices at anytime, from anywhere and using any device.

Existing approaches of sharing the application's user interface are based on transferring between the devices the content of frame buffer or the drawing commands. They were developed when the desktop was dominant metaphor for interacting with computers. Mobile devices are characterized by much smaller screens and limited input capabilities. The new interaction metaphors for mobile devices are highly optimized for this characteristics. For example, interacting with an application designed for desktop computers from a mobile phone having keypad and joystick makes the application unusable. Therefore, when sharing an application user interface is it highly desirable to adapt it to the rendering device look and feel.

This paper outlines a practical approach for enabling adaptive and multi-device application sharing. The elements of an application user interface are modeled as RESTful resources enabling rendering devices to acquire platform specific representations compatible with the rendering device look and feel. The user interface and the application state are synchronized using an event-based system. We introduce the methodology for modeling the user interface elements using resources, describe the prototype implementation and discuss our experiences.

The paper is structured as follows. Section 2 provides an overview of existing work related to adaptive and multi device application sharing. Section 3 describes the remote MVC architecture and the methodology for modeling the user interface as RESTful resources. Section 4 presents the prototype implementation and experimental observations. Concluding remarks are provided in Section 5.

2. RELATED WORK

In this section we describes existing technologies and methodologies that are relevant for interactive network based applications that export adaptive and multi device user interfaces.

2.1 Remote user interface protocols

The process of exporting a user interface to another device can be performed at different abstraction levels: frame buffer, graphics device interface, or widget descriptions. With the remote frame buffer approach, used by Virtual Network Computing (VNC) [11], a server component captures the content of the local frame buffer, transfers it to rendering device, which copies it to the frame buffer of the remote device. Another approach, used by X Window[1] system, is to transfer to the remote device the drawing commands for the graphics device interface (GDI). The remote device receives and executes the commands locally. The widget descriptions approach is used primarily by rich internet applications, developed with Adobe Flash[2] and Microsoft Silverlight[3] toolkits, or HTML 5. A generic mechanism for exporting widget descriptions was initiated by Widex Working Group[4] in IETF but that effort is currently stalled.

2.2 Adaptive and multi-device user interfaces

The proliferation of mobile devices with rich user interface capabilities created a demand to develop applications that have the

[1]http://www.x.org
[2]http://www.adobe.com/products/flash
[3]http://silverlight.net
[4]http://tools.ietf.org/wg/widex

WS-REST 2010, April 26, 2010; Raleigh, NC, USA

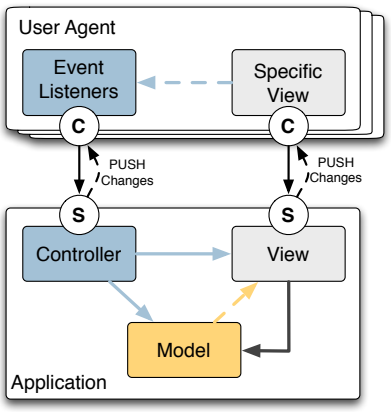

Figure 1: Conceptual view of the Remote Model-View-Controller architecture

ability to use the native features of the devices. Adapting the user interface to these capabilities improves the usability of the particular application and the overall user experience. UIML [6] defines a canonical meta-language that can describe any user interface in a manner that is device-independent and user interface metaphor independent. The language defines rules that allow translating the user interface to specific representations for traditional desktop, web or mobile devices. SUPPLE [4] provides a framework for automatically adapting the user interface to the device at hand. The approach considers the adaptation as an optimization process and can be customized to user usage patterns.

2.3 The web architecture

Representational State Transfer (REST) [2] architectural style proposes the design of network based applications in such a manner that their functionality is implemented as a set of resources. Each resource can be referred using an Universal Resource Identifier (URI). Clients can interact with the resource using a fixed set of verbs (e.g. the `GET`, `PUT`, `POST`, `DELETE` methods of HTTP protocol), and exchange well defined representations. A severe limitation of REST is that interactions follow the request response pattern which is always initiated by the clients. These limitations are addressed by ARRESTED [7], a set of extensions that provides support for distributed and decentralized systems.

The Web is the archetypal REST application. Thanks to the loose coupling principles that lay in its REST foundation, the modern web evolved from the early hypermedia system, in which web pages were the primary representation of the resources, into the largest information system in which is quite common for services to expose their internal data using HTTP RESTful APIs.

3. ARCHITECTURE

Our proposal provides an architecture that enables interactive applications to export their user interfaces on different devices a process in the user interface. We define a Remote MVC design model that follows the REST architectural style.

3.1 Remote MVC architectural pattern

The Model-View-Controller architectural pattern (MVC) [8] divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model. The pattern specifies that the structure of the scene-graph representing the user interface changes when the application logic (e.g. the model) updates the data that needs to be presented to the user, or when the user provides input, each such interaction being converted by the window manager into a specific event that is handled by the controller.

In our environment, the application sharing experience is provided cooperatively by two separate applications. The original application, located on the hosting device, provides the functionality of the model and the controller, while the user agent, located on the rendering device, provides the functionality of the view. In this distributed environment our original interactive application becomes a network based system in which the two applications provide users the same functionality.

To accommodate this situation, we extended the classical MVC pattern with an event based mechanism that provides a level of consistency close to the case when the model, the view and the controller reside on the same physical device. Therefore, the Remote MVC system (R-MVC) provides the capabilities to propagate, from one device to another, the changes induced on the view by the model, and the user interactions are delivered to the controller as remote events, see Figure 1.

3.2 A RESTful user interface data model

3.2.1 View related resources

A typical user interface is represented using a scene-graph data structure. Each node in this data structure represents an element of the user interface, e.g. a widget in graphical user interfaces, and each edge represents a parent-child relationship. Often, a node may have several children but only one parent.

The state of the user interface of an application is determined by the structure of a scene-graph and by the properties of each node. We model the data structure that contains the view using a set of resources. The *root* of the user interface is always modeled as the resource identified by `/ui`. By convention, we identify each resource corresponding to a user interface element using the following URI template:

```
http://example.com/ui/{uiElement}
```

Although the URI scheme that identifies the user interface resources is flat, a typical platform specific representation of the user interface root contains all information that allows a client to reconstruct the scene-graph structure for the specific target platform. In case a platform specific representation does not have native support for describing the hierarchy of the user interface, we can overcome this limitation using use the XML linking language (Xlink) [1] defined mechanism to encode the relationships between the resources.

`GET` method allows a user agent to acquire a platform specific specific representation of the user interface. The user agent informs the abstract view about its platform capabilities in the request using the HTTP content negotiation mechanism (e.g. the `Accept` header):

```
GET /ui HTTP/1.1
Accept: {mimetype specific to the platform}
```

The response contains the platform specific representation of the user interface, together with the information that will enable the user agent to be notified when the resource state changes. The URI where the notifications are available is provided using the `Link` [10] header, and its relation `rel` to the current document is *monitor* [12]:

```
HTTP/1.1 200 OK
Content-Type: {mimetype specific to the platform}
Link: <{notificationsURI}>; rel="monitor"
<!-- platform specific ui representation -->
```

If notifications URI is the same as the request URI, the user agent should understand that the `/ui` resource is compliant with asynchronous REST (A+REST) [7]. Alternatively, the resource may provide the notifications via an external XMPP resource.

`POST` method allows user agents to update the values of the properties specific to the target user interface elements. Typically the argument of the call is a dictionary containing key-value pairs:

```
POST /ui/{uiElement} HTTP/1.1
{"key": value, }
```

3.2.2 Controller related resources

A typical user using an interactive application generates a large number of events. Among them only a small amount are relevant for the application. Although all events detected by the local window manager are passed to the application, the event handlers treat only the relevant ones, the rest being ignored.

For our environment, it is not practical to transfer all events from the user interface rendering device to the application host device to handle there only a few. To address this situation, we create the Remote Event Hub (REH), a new resource `/reh` that provides the functionality which allows the user agents to find out which events are relevant for the controller. Each event handler defined in the controller has a correspondent REH sub-resource identified using the following URI template:

```
http://example.com/reh/on_{uiElement}_{eventName}
```

`GET` method allows a user agent to find which events emitted on the user interface rendering device are relevant for Controller. The user agent informs the controller about its platform capabilities in the request using the HTTP content negotiation mechanism (e.g. the `Accept` header):

```
GET /reh HTTP/1.1
Accept: {mimetype specific to the platform}
```

The response includes a list of sub-resources corresponding to the events relevant to the Controller, including information about the target user interface element and the event name:

```
HTTP/1.1 200 OK
[{"uiElement": {uiElement},
  "eventName": {eventName}},]
```

this contains the necessary information that allows the user agent to create local listeners for the events the Controller is interest in, corresponding to remote event handlers.

`PUT` method allows a user agent to inform the controller that a relevant event was emitted on the user agent:

```
PUT /reh/on_{uiElement}_{eventName} HTTP/1.1
{"key": value, }
```

The Controller is notified immediately when the message is received and the appropriate event handler is invoked with the provided parameters.

3.2.3 User interface manifest

The manifest is a resource that describes to a user agent, in a machine readable format, the collection of resources associated with the user interface, the structure of these resources and how the user agent can interact with them.

For example, the following Web Application Description Language (WADL) [5] snippet contains the description of the web interface exposed by an adaptive and multi-device application. The application has two resources, corresponding to the view and the controller related resource collections. Additionally, the application advertises the supported user interface description languages by listing their mime types as possible representations:

```
<application>
 <resources base="http://example.com/">

  <resource path="ui/" type="#ui">
   <method name="GET">
    <response status="200">
     <representation
      mediaType="{uiDescriptionMIMEType}" />
    </response>
   </method>
  </resource>

  <resource path="reh/" type="#reh" />

 </resources>
</application>
```

Further, the `ui` and `reh` resource types, including the sub-resource structure are described in a simplified form in an additional WADL snippet:

```
<application>

 <resource_type id="ui">
  <resource path="{uiElement}" />
 </resource_type>

 <resource_type id="reh">
  <resource path="on_{uiElement}_{eventName}" />
 </resource_type>

</application>
```

This conveys enough information that allows a user agent to identify the resources exposed by the applications, how to access them and if it is able to render the user interface exposed by the application.

3.3 Orchestration

The effect of having the user interface, rendered by the user agent, synchronized with the application logic is achieved by interacting in a collaborative way with the view and controller resources on the application host. This process is orchestrated during the initialization phase which allows the user agent to acquire the needed information, see Figure 2.

Initially, the user agent requests the user interface manifest allowing it to discover the resources exposed by the application. Then, the user agent requests the platform specific representation of the user interface. Together with the platform specific representation, the response form the application host server contains information on where the user agent can monitor the state of the view related resources. The user agent receiving this message creates the user interface and starts polling the notification resource for user interface related updates. Next, the user agent requests the target user interface elements and the specific events the controller is interested in. The server response contains also the location of the notification resource where the state of the controller related resources can be monitored.

Once the initialization is performed, the rendered user interfaces is consistent with the application state. Further changes are delivered by the application host server or by the user agent interacting

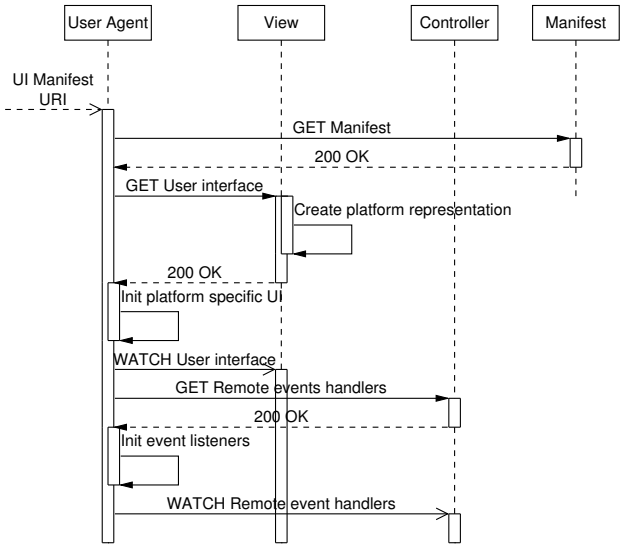

Figure 2: Interaction pattern during initialization

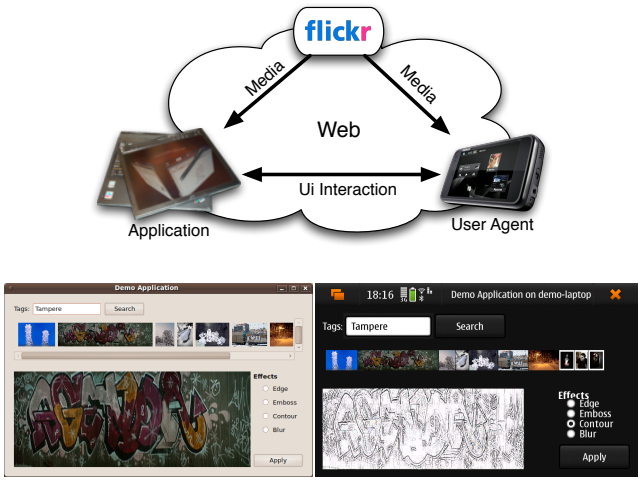

Figure 3: The test environment (above), the user interface rendered on the laptop (left), and the user interface rendered on the mobile device (right)

with the appropriate resources. Errors encountered in this interaction may lead to inconsistency between the user interface rendered on the user agent and the model. In such an event, the user agent should trigger the re-initialization for acquiring the latest state of the user interface on the server using the `GET` request on the view resource.

4. IMPLEMENTATION EXPERIENCE

4.1 Prototype implementation and test environment

We implemented the basic functionality as an extension of Qt[5] application and user interface framework. The prototype is developed in Python[6], with PyQt[7] providing the bindings to Qt libraries. We leverage the ability of the Qt framework and the developer tools to create applications that are compliant with the MVC design pattern as well as the cross platform capabilities which allow us to easily deploy the applications on multiple devices. The server functionality is provided as a library that can be embedded in any Python Qt application to make it compatible with our system, and the user agent functionality is fully contained in a standalone Python Qt application. The user interface manifest is discovered using bonjour/zeroconf[8].

We developed a sample application with the intention of experimenting and stress testing our prototype implementation. The application allows the user to search photos stored on an internet photo sharing service such as Flickr[9], then select one form the resulting set and apply locally an image processing effect. The user interface is updated by the model (e.g. when displaying the results from the photo sharing server), and by the user (e.g. when selecting the image and the effect). This allows to test the behavior of the system under both model initiated changes and user interaction initiated changes. Additionally, we can observe the behavior of the system under changes in the structure of the user interface as each search creates new widgets, and the corresponding resources.

The test environment contains a laptop running Ubuntu 9.10 and a Nokia N900 mobile device, see Figure 3. The test application runs on the laptop, which plays the role of the application host server while the mobile device plays the role of the user agent. The user interface is rendered on the laptop and mobile device using the local platform specific styles, e.g. Clearlooks for the laptop and Hildon for the N900. The general layout of the two representations of the user interface is similar (e.g. the structure and widgets) but there are differences with some of the properties of the widgets. We can observe that the user interface on the laptop is optimized for interaction using pointing device and keyboard, while the user interface on the mobile is optimized for finger interaction.

4.2 Observations

Modeling user interface using resources according to the REST architectural style is a conceptual exercise. It is as important to determine that this architecture is implementable and that it is possible to create applications with user interfaces that can adapt on multiple rendering devices. Our Remote-MVC pattern requires an event based infrastructure to keep the components on different devices synchronized. Implementing this functionality using RESTful resources provided the desired results although at network level it constrains us to a request response interaction pattern.

Our test use case involve only one user interacting with the application from one remote device. However, the architecture does not impose limitations on the number of user agents on remote devices, nor on the number of users involved. In case of multiuser scenarios there is the possibility that two or more user agents attempt to update the same resource. This situation can be addressed using a locking mechanism such REST with delegation (REST+D) [7], or a floor control protocol.

The user agent needs the user interface manifest to be able to render the user interface. In our test environment the devices hosting the application and the user agent were in the same local network and using zeroconf as the service discovery mechanism was appropriate. However, the architecture does not impose the use of any particular service discovery mechanism and other protocols

[5]http://qt.nokia.com
[6]http://www.python.org
[7]http://www.riverbankcomputing.co.uk/software/pyqt
[8]http://www.zeroconf.org
[9]http://www.flickr.com

that enable the user agent to find the manifest URI can be used. Alternatively, the user agent can acquire the manifest identifier as a result of real world interaction, such as by reading an RFID tag or taking a picture of a 2D barcode.

Applications running on constrained devices or applications that have a large number of user agents with active connections may consider that is more efficient to outsource the resource change notification to an external resource, such as PubSubHubBub [3] or publish/subscribe enabled XMPP [13, 9]. Such external resources provide functionality that is similar to the `WATCH` verb proposed in ARRESTED.

The native user interfaces of mobile devices especially are highly optimized for their hardware characteristics. Using this assets together with web technologies provides a powerful combination that allows the user interfaces of remote applications to be presented to the users using the same look and feel of the local applications. For many such devices it is not possible to achieve this level of integration using mainstream hypertext based user interface technologies available in modern web browsers.

5. CONCLUSIONS

This paper outlines a practical approach for enabling adaptive and multi-device application sharing. The elements of an application user interface are modeled as RESTful resources enabling rendering devices to acquire platform specific representations compatible with the rendering device look and feel. The user interface and the application state are synchronized using an event-based system. Furthermore, the prototype implementation and experiences with developing applications revealed that the architecture is both feasible and effective.

6. ACKNOWLEDGEMENTS

The author would like to thank Petri Selonen, Tapani Leppanen, Juha Savolainen and Rod Walsh from Nokia Research Center, and the reviewers for their comments and suggestions that improved this document.

7. REFERENCES

[1] S. DeRose, D. Orchard, and E. Maler. XML linking language (XLink) version 1.0. W3C recommendation, W3C, June 2001. http://www.w3.org/TR/2001/REC-xlink-20010627/.

[2] R. Fielding and R. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, Jan 2002.

[3] B. Fitzpatrick, B. Slatkin, and M. Atkins. PubSubHubbub Core. Working draft, Feb. 2010. http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html.

[4] K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 93–100, New York, NY, USA, 2004. ACM.

[5] M. Hadley. Web application description language. W3C member submission, W3C, Aug. 2009. http://www.w3.org/Submission/2009/SUBM-wadl-20090831/.

[6] J. Helms, R. Schaefer, K. Luyten, J. Vanderdonckt, J. Vermeulen, and M. Abrams. User interface markup language (UIML) version 4.0. Committee draft, OASIS, Jan. 2008. http://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf.

[7] R. Khare and R. N. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437, Washington, DC, USA, 2004. IEEE Computer Society.

[8] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[9] P. Millard, P. Saint-Andre, and R. Meijer. XEP-0060: Publish-subscribe. Standards track, XSF, Jan. 2010. http://xmpp.org/extensions/xep-0060.html.

[10] M. Nottingham. Web Linking. Internet draft, Jan. 2010. http://tools.ietf.org/id/draft-nottingham-http-link-header-07.txt.

[11] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33 –38, Jan/Feb 1998.

[12] A. B. Roach. A SIP Event Package for Subscribing to Changes to an HTTP Resource. Internet draft, Dec. 2009. http://tools.ietf.org/id/draft-roach-sip-http-subscribe-06.txt.

[13] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920, Oct. 2004.

Author Index

Belimpasakis, Petros 54

Engelke, Charles 27

Fernandez, Federico 31
Fitzgerald, Craig 27

Garrote Hernández, Antonio 39

Hadley, Marc 8
Hausenblas, Michael 23

Jacobi, Ian 46

Kelly, Mike 23

Marinos, Alexandros 1
Moreno García, María N. 39

Navon, Jaime 31

Parastatidis, Savas 16
Pautasso, Cesare 1
Pericas-Geertsen, Santiago 10

Radul, Alexey 46
Richardson, Leonard 4
Robinson, Ian 16

Sandoz, Paul 10
Selonen, Petri 54
Silveira, Guilherme 16
Stirbu, Vlad 62

Webber, Jim 16
Wilde, Erik 1

You, Yu 54