Web Services: environments, usage and methodologies

CSCI 3171 Network Computing - Course Project

This report explains about what are Webservices, where they are currently used and what are they used for. Then, the text accounts for current market methodologies to address those needs and how they stack side by side, and where each technique performs better. It talks mainly about REST, SOAP and Thrift.

Igor Gomes dos Santos – B00617605 Winter / 2012-2013

Table of Contents

INTRODUCTION: WHAT ARE WEB SERVICES?

A web service is a communication method between different devices using as medium network connections, many times being through the World Wide Web. It is a concept used to describe how different systems are able to interact with each other, and how they should function in order to enable interoperability of operations between different applications and needs. Shortly, it's a machine-to-machine interaction system.

It should not be misunderstood as something similar to a network protocol. Rather, a web service is something more high level; it's not only a way to transfer information between two peers in a network, but a way to request an operation to be done remotely, instead of running local code. It may be requesting information about an object stored in another database that you don't have access to, or requiring some expensive operation to be done outside of your scope – that your machine was not meant to do, but still need that result to do its job.

There are two main concepts that will be discussed in this paper: architecture approaches and technologies available to fit those architectures. You will be presented with the types of usage and environments where web services are beneficial; then, how they can be implemented; later on, there's a comparison on the methodologies and technologies. In the conclusion there are some advices on which one best suits certain situations.

The official definition of "web service", as per the W3C Working Group Note from February 2004 (W3C Working Group 2004), is formally related to SOAP. However, nowadays the expression is widely used to refer to other types of machine-to-machine protocols, and it's this broader meaning that we are discussing in this paper.

WHAT ARE WEB SERVICES USED FOR?

There are some common computation and architectural problems that can be solved by using web services. Those include internal communication between different machines in the same network; separation of responsibilities into different machines for better decoupling; inclusion of externally provided services to improve overall application functionality; and something deeper than this, called *mashup*.

Communication between servers in the same infrastructure

In big applications, where you have different servers running different parts of your infrastructure, such as having a load balancer, several web servers, some database machines, cache servers and so on, it may be needed to enable communication between them. Not only to forward requests or ask for a database entry – those are already handled by their underlying technologies, but for example to be able to centralize logging information about every machine in one place. This is not something easy to be done, since simultaneous read/write synchronization is something very hard to be achieved.

One of the solutions found for this type of problem was developed by Facebook, and later open-sourced: it's called Scribe. It is a scalable log server architecture, designed as a peer-to-peer system (see Figure 1), where each machine is able to write their logs to another one, funneling the structure until some of them write all the logs into a Hadoop Database. There it's possible to analyze all the information by using *map/reduce* functions. The log data is streamed in real-time between the servers, and uses Apache Thrift (discussed in detail later) as communication medium between the peers. It's also currently used by Twitter and Zynga (one of the biggest mobile/online gaming company).

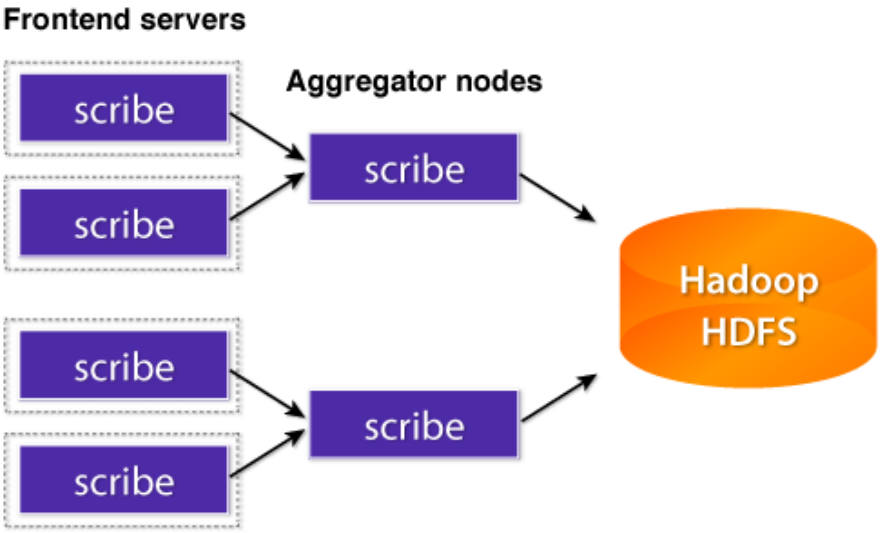


Figure 1: Scribe peer-to-peer architecture

Scribe is available for a number of languages, since it builds on top of Thrift (that's almost language agnostic). It's very fast as it's written in C++, but on the other hand it makes it harder to be extended and maintained. Currently one of the alternatives presented in place of Scribe is Fluentd, a similar application written in pure C with the topmost layer written in Ruby, making it much easier to be extended – the reason why they choose Ruby for the API level. It features a number of language plugins, such as Ruby, Python, PHP, Perl, Node.js and Java, but it's not as versatile as Scribe is. However, it does not depend on web services on its core, as the communication is done through pure TCP sockets and a binary serialization format for JSON (the native storage format) called MessagePack. This decision shows us a little bit the balance needed between versatility and efficiency.

Share of tasks and responsibilities between different machines

Similar to the last point, it's common for big systems to have different parts of the same software running in different machines. Sometimes, to maintain a decoupled architecture, web services are more interesting than vanilla sockets. Surely they can harm the performance, but on the other hand this type of technology can be useful when it's needed a more high-level component, that could be easily detached and substituted by another one without much changing of the internal code, or that could be reused by different sub-systems.

Use of different services to improve application possibilities

This is, by and large, the most common and easy to see type of use of web services nowadays, and it's what made the expression so widespread. As you can see in Figure 2, there has been an enormous growth of REST APIs in the web. We will talk about the differences later, as what matters now is simply how big they become in the later years. On one hand, the companies started seeing how interesting is opening some services they do to the public; on the other, developers and product managers started thinking about what's possible with those available services.

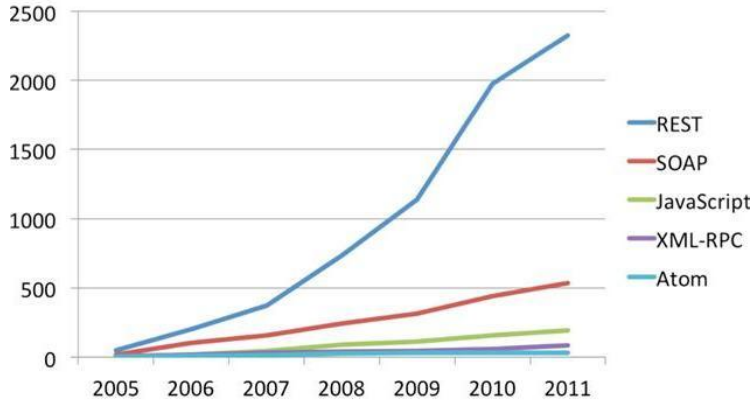


Figure 2: Growth of REST APIs versus common options since 2005

Currently, instead of creating your own maps system, you can simply add a few lines and call the Google Maps API, showing custom-made widgets, with many settings available to further customize the user experience. Instead of integrating tedious payment SDKs into your website, you can simply integrate with the Paypal API; in the work of some (one?) days you can fully offer your users the ability to pay for your services or products using the market's credit cards or even by wire transfer - without having to bother with security certificates, secured promises, and so on. You can run custom face recognition calls with no need to create big software just for use this feature, you can create telephony systems, or even ask your to-be-users to login on Facebook or Twitter instead of creating a new account, just for your website.

Talking about Twitter, because of the very website nature, they saw a big boom in applications that used their API after the launch – it was very convenient to have your phone showing you the latest tweets, and why not having a similar software on your computer instead of having to open the website every time you wanted to check if there's something new (and that's a frequent task for the average Twitter user)? After some time they even ended up by buying the most popular desktop application, instead of developing their own – TweetDeck was bought for $40 million in mid-2011. And they took some time developing their own mobile applications too – why bother if their users were already quite satisfied with third-party apps?

Mashups

Another possibility quite similar to the last topic is called a "mashup": a deeper version of API integration where your very application value resides on the usage of one or more web services. The idea here is not to simply use a service in your system – is to make existing data more useful for the end user.

Google Maps, for instance, is a type of mashup that you probably already use: it integrates information from Wikipedia, Panoramio, Flickr and even YouTube to make their maps more useful and

informative. DUIMap.org is a mashup of Google Maps using information about drunk driving accidents to provide useful information about this type of incident through the USA.

There's also many mashups regarding search, such as:

- Shopbot.ca: allows you to compare prices of the same product in multiple websites
- MP3Skull.com: enables the user to download a music from different sources
- DuckDuckGo.com: integrates many different services to provide the user with a powerful search engine that may give you information about currency conversion, Wikipedia articles, programing method documentation, and so on. They even allow developers to publish their own extensions to the search engine, through DuckDuckHack.com.

Another successful example of a mashup is Teambox.com, an integrated system to provide teams with organization of tasks and information around projects they are working on. It integrates Google Calendar for meetings and tasks deadlines; Dropbox and Box to store files locally; Google Docs attachments to tasks and notes; Gmail extension to turn messages into tasks; integration of tasks from coding systems such as GitHub and Pivotal Tracker; and so on. Everything was put together in a nice way that does not clutter the user interface, but instead adds value to the user, following the mashup idea.

WHAT ARE THE ENVIRONMENTS THEY CAN BE USEFUL?

As noticed before, there are two main environments where web services are useful: in enterprise settings and in the open web, each one having different behaviours and needs. Here we will be discussing about them, and making some assumptions so we can have two very different scenarios to develop our arguments later.

Usually, big companies use web services to merge different servers into an integrated environment, and frequently that depends on high performance technologies to save on computing resources. It's common to see that type of infrastructure being developed by one team, or someone guiding different teams, so they can have a more coupled development without losing much functionality. The need of high security communication protocols is frequent too, so the executives can rest safe their sensitive information is not being listened by someone else.

On the other hand, the average web company that depends on web services is small and fast-paced. Many times they follow Agile practices to develop more features in less time. The majority of companies are focused on one or two products, and the main focus is on making they work great for the end user. This way, their need of web services is rarely related to separate servers in their infrastructure, but instead to other services offered by third-parties upon which they can build their software. This also lead us to talk about the companies that are distributing those services, companies such as FourSquare, Facebook, Dropbox, Google, and many others that offers web services that can be consumed by several different systems, that usually have similar needs but different approaches on how to tackle those problems. Those systems can also be built on different languages, ranging from the most common such as Java and PHP to newer environments such as Ruby, Python, and JavaScript (client or server-side – available through the Node.js platform) and even some more exoteric like Scala, Grails or Erlang. And

those web services should be available in a way that could serve all those different environments and needs – and the best way to achieve that is via a decoupled architecture.

WHAT ARE THE METHODOLOGIES USED TO MEET WEB SERVICES USAGE?

We can see that there are two main mindsets in the web service world: that about decoupled architecture and open technologies, and the opposite side. Both serve to different purposes, and have thus developed very different environments for creation of web services.

Service Oriented Architecture - SOA

On the enterprise setting the most discussed architectural approach is SOA – Service Oriented Architecture. Shortly, it's a way to organize multiple systems on a way that they can depend on each other and share tasks and services between them. The main idea is having sub-systems doing specific tasks and bigger applications depending on that, instead of re-writing the same – or similar – algorithm again and again. In one way this idea is similar to the commonly referenced DRY philosophy (Don't Repeat Yourself), as it says that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system". Following SOA means a big system, made of different sub-systems, will always have only one place to define a shared function and also only one place to change it if needed. This makes the development more lean and straight-forward, further speeding up the process of releasing new software or features for the existing systems.

Common approaches to apply SOA relies on technologies such as Remote Procedure Call / Remote Method Invocation, standards for service discovery and other, more pragmatic technologies. The W3 Consortium has released a specification on Web Services that is directly related to SOA and its underlying technologies.

Representational State Transfer – REST

The other approach used nowadays to solve web services needs is tightly coupled with the underlying technologies. While SOA defines autonomous ways to discover services and how they should be called, REST is more straight-forward and clean. It depends on the HTTP protocol and its methods (commonly called *verbs*) to separate different types of operations – for example, destructive calls should call the web service using the DELETE method, while pure informative ones should use GET. The name come from the fact that each resource in the system should have a representative state at any given time – and thus, an URI (Universal Resource Identifier). We will be discussing how REST is implemented in more detail later.

The main difference from REST to SOA is that, while the latter relies on remote calls to methods and functions, and thus creates highly coupled systems, REST is free and more human-oriented, and thus is easier to be implemented in different environments and the systems that depends on REST services does not need to be implemented in a given way – the technologies supported by REST are open and pretty much any system connected to a network will have libraries to work with them (HTTP and XML or JSON, namely).

How web service technologies are used to meet those needs?

RPC / RMI

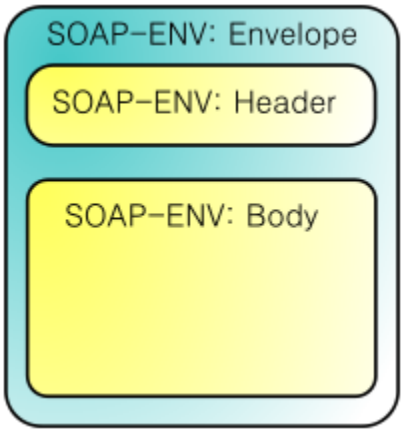
One of the first approaches to solve the separation of concerns between different systems was RPC. It stands for Remote Procedure Call, and on short it's a way to call a function that's defined outside of the local machine, through the network. It's implemented in many different languages and usually those implementations are incompatible with each other, as many of them have different methods of transport and packaging of information.

The local software have a stub of the server functions, knowing their names and signatures; this stub is called with the needed arguments, and it packs that information in a message that's sent through the network to the server specified by that stub. The server receives the request, processes and returns a message back, that's received by the stub and then used by the local application again. All this processing is, by default, blocking – meaning the local application will wait for the reply from the server, and the CPU will be freed for other processes to run, similar to how database access is usually done.

On the object oriented world, RPC is called RMI and there are little changes in the underlying architecture, but for the purposes of this paper, they will be referenced as synonyms.

Usually web services that have RPC/RMI as their basis depends on a methodology called service discovery, that's used for automation of various levels of the web service architecture – mainly the initial coding and binding of connections between machines, as this type of information can be served in the network and the clients may use it to generate bootstrap code.

SOAP



One of the derived web service methods from RPC architecture is called SOAP – usually referencing Simple Object Access Protocol, but after version 1.2 the acronym meaning was dropped. It was created in 1998 and its definition relies on several standards, usually called WS-*, that define different layers of SOAP operations, such as security, routing, discovery of services and so on. SOAP is the successor of XML-RPC, a technique of Remote Procedure Call based on the XML text format. It envelopes the original message with its header into a SOAP root element, and transfers this through the network. For sending that message SOAP can rely on different transport protocols, such as pure TCP, HTTP, SMTP or JMS, making it quite neutral about networking implementations and performance, fitting to the software needs. Being able to work over HTTP, it can also work on a similar fashion as standard RESTful services, that will be discussed later on this paper.

Although not a required part of the SOAP architecture, the standard packs two other acronyms – that have actual meaning: WSDL and UDDI, Web Service Descriptions Language and Universal Description Discovery and Integration. UDDI is a former service where public servers would work as brokers between clients and the end-point servers; the client would ask the UDDI server for someone

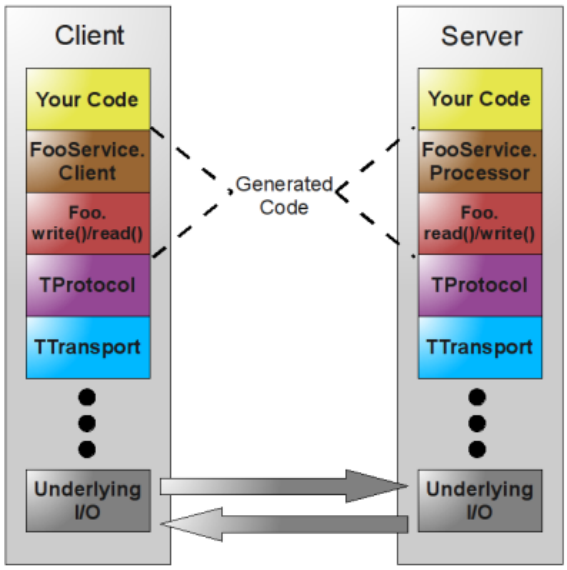
who could fulfill their need of a specific task through a specific protocol, and then the client would receive the information needed to establish a connection to the service provider. Then this provider would send the client a WSDL file, that describes the operations available and how they may be called, along with information on how the requests and responses should be formed.

UDDI was not widely adopted, and nowadays the public servers from IBM, Microsoft and SAP (the specification creators) were shut down, and the working group responsible for it has been closed too. This happened because the system ideal is quite different from how web services have evolved, and the environment had little to do with how web services have evolved. However, this is still used in close environments such as companies. On the other hand, WSDL plays a key role on how SOAP is used nowadays.

Thrift

Another method of serving RPC services is Thrift, a framework developed by Facebook to tackle their internal communication needs. In 2007 they decided to open source the system and have donated it to the Apache Foundation, and now it's formally known as Apache Thrift. It relies on a description language with a similar purpose of that from WSDL, but with a more directed usage, to improve development speed and language performance too.

Instead of relying on pure XML for service discovery and algorithm automation, the developer has to write (or receive from someone else, or find) a Thrift definition file that cleanly describes the structures and operations available through that server, with syntax similar to C/C++ (using *`structs`* and *`enums`*, for instance). Then, that developer will run the Thrift generator on top of the given file to generate all the files needed to build the client and the server. Thrift supports several languages, such as Java, PHP, Python, Ruby, JavaScript, C, C++, C#, Cocoa, Haskell, Erlang, Perl, OCaml and SmallTalk, to a varying degree. Those files are later required given the language conventions to startup a server – or create a connection from the client –, and then the developer can call their own business classes as needed by their environment, doing their operations and calling the Thrift operations as needed to send a request or response.



Thrift comes packed with many different options of transport layers and protocols, and the server developer can choose what fits the situation better. There are plain-text and binary protocols (JSON, XML, simple and compact binary), and socket, framed, memory and file transports. It's also possible to choose which type of server will be used, between threaded and non-threaded server, and blocking or non-blocking (this requiring the framed protocol to be used).

By "supporting several languages to a varying degree" I meant that not all protocols, transports and servers are available in all platforms. For example, in the client JavaScript implementations it's not

possible to create servers (due to language constraints), and the compact binary protocol is only available on Java, Python, Ruby and C++. The client and the server implementations may differ (being this the main purpose behind the Thrift architecture) as long as both use the same protocol and transport options.

REST

REST, while being an architecture style for distributed systems, is also considered a technology approach for web services since it's tightly coupled with open and relatively easy-implemented technologies such as HTTP, XML and JSON. The definition of this standard was created in 2000 by one of the lead authors of the HTTP specification.

The main concept behind REST is that each resource should have a unique identifier (an Universal Resource Identifier, URI), and it should be accessed through specific HTTP verbs, that would define what would happen to that resource. A REST server will have a root URI, that may contain the API version (such as *www.example.com/api/v1/*) and the resources will be attached to that URL, adding the collection name and then the resource identifier when needed. Namely:

- GET is used to retrieve information; it does not send anything in the request, except the resource's URI, and receives back the resource details. An example of a resource partial identifier would be */products/4596*. A GET request to a clean URI would retrieve all resources under that name (as would happen with */products*).
- POST is used to create new resources; it's usually request through the main resource name, and sends in the request's body the information needed to create the resource. To create a new product in our example, the client could send a POST request to */products*.
- PUT is used to change an existing resource; it sends the new information in the body, and the URI contains the specific identifier for that resource, for example, */products/4596*.
- DELETE is used to remove or hide resources. The client does not send any information in the body, but the URI is complete as it happens with PUT requests.
- OPTIONS is also used by some implementations to list the available operations, although this is not part of the REST standard.

Other definitions not related to REST but to HTTP itself are related to the used verbs: PUT and DELETE are idempotent methods, meaning that if they are called multiple times at once, they should have the same effect as one unique call; GET is a nullipotent method, meaning it should not alter resources in any way – should not have side effects beside its informative purpose.

As REST is not a protocol but rather an architecture, there's no official standard, but recommendations only. Services that implement correctly those recommendations are usually called RESTful services. Also, a collection does not need to implement all the verbs to be considered RESTful, as many times this is not even feasible through a business view – for example, the client can be able to create a money transfer using POST, but there's no sense on creating a DELETE operation for transfers – once they happen, they are permanent, and could only be undone with another transfer; in this example there's no sense on using PUT on a resource too, but GET could be used to retrieve information on past transfers.

How those technologies compare, given the environments?

Security

One of the biggest misconceptions between REST and SOAP are related to security. As SOAP is an entire protocol on its own, and relies on different protocols for transport and other tasks, its WS-* specifications explain how a fully secure communication can be achieved. On the other hand, REST can work easily through HTTPS, and this way it's as safe as SOAP + WS-Security, when talking about end-to-end encryption. This "big specification" on security created the myth that SOAP is more secure than REST, when actually they are equivalent on security measures. It's also possible to implement SSL on Thrift, being as safe as the other two options.

Atomicity

SOAP here clearly "wins the battle", through the WS-Atomic Transactions specification. HTTP, being by definition a Stateless protocol, and REST being an architecture designed to change from state to state, is not meant to guarantee atomic operations. This can surely be implemented through an abstract "Transaction" resource, but this operation is not standardized in any way nor supported by any part of the REST specification. The same applies to Thrift, as it does not sport a default transaction specification, but it could be easier to implement since it does not create a stateless communication.

Interoperability

Here REST has a clear advantage in front of the others: while being usually only implemented through HTTP using XML or (more commonly nowadays) JSON could hinder its ability to work through different environments, as those technologies are widely available in the most different platforms, this guarantees that REST will be able to be used in almost anywhere.

On the other hand, SOAP has a large number of different standards (each one with different versions), and different vendors implement different standards, and many times those implementations slightly differ, sometimes giving the developers issues when they're trying to implement a service using a different environment than the one in the server.

This issue also happens on Thrift, although it's somewhat lighter; as explained before, different languages may lack some protocol/transport implementations, and thus if the server is implemented with one of those not-widely-supported components, it will decrease the range of possible languages to implement the client side.

Conclusions

What would be better in a corporate environment with X details, or a common website, or need of high traffic of binary data, and so on.

REFERENCES

Apache Foundation. "Language Support." *Thrift Wiki.* January 24, 2012. (accessed April 05, 2013).

Mason, Ross. "InfoQ." *How REST replaced SOAP on the Web: What it means to you.* October 20, 2011. http://www.infoq.com/articles/rest-soap (accessed April 07, 2013).

MessagePack. "Powered by." *MessagePack Wiki.* May 25, 2012. http://wiki.msgpack.org/display/MSGPACK/PoweredBy (accessed April 06, 2013).

Programmable Web. *Mashups Directory.* n.d. http://www.programmableweb.com/mashups/directory (accessed April 06, 2013).

Prunicki, Andrew. "Introduction to Apache Thrift." *Object Computing, Inc. - Java News Brief.* June 2009. http://jnb.ociweb.com/jnb/jnbJun2009.html (accessed April 04, 2013).

TechCrunch. *Twitter buys TweetDeck for $40 Million.* May 23, 2011. http://techcrunch.com/2011/05/23/twitter-buys-tweetdeck-for-40-million/ (accessed April 07, 2013).

Treasure Data. "Enabling Facebook's Log Infrastructure with Fluentd." *Treasure Data Blog.* January 17, 2012. http://blog.treasure-data.com/post/16034997056/enabling-facebooks-log-infrastructure-with-fluentd (accessed April 06, 2013).

W3C Working Group. "Web Services Architecture." *World Wide Web Consortium.* February 11, 2004. http://www.w3.org/TR/ws-arch/ (accessed April 01, 2013).

Wikipedia. *Apache Thrift.* March 14, 2013. http://en.wikipedia.org/wiki/Apache_Thrift (accessed April 04, 2013).

—. *DRY (Don't Repeat Yourself).* April 02, 2013. http://en.wikipedia.org/wiki/Don%27t_Repeat_Yourself (accessed April 07, 2013).

—. *Remote Procedure Call.* April 03, 2013. http://en.wikipedia.org/wiki/Remote_procedure_call (accessed April 05, 2013).

—. *Representational State Transfer.* April 07, 2013. http://en.wikipedia.org/wiki/Representational_state_transfer (accessed April 07, 3013).

—. *Scribe (log server).* March 23, 2013. http://en.wikipedia.org/wiki/Scribe_(log_server) (accessed April 05, 2013).

—. *SOAP.* April 05, 2013. http://en.wikipedia.org/wiki/SOAP (accessed April 07, 2013).

—. *Universal Description Discovery and Integration.* March 30, 2013. http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration (accessed April 07, 2013).

—. *Web Services Description Language.* February 22, 2013. http://en.wikipedia.org/wiki/Web_Services_Description_Language (accessed April 07, 2013).