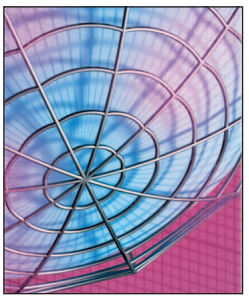

# A Difficult Abstraction

**Danny Ayers** • *Talis*

In this column, I'll describe some of what the developers of generalized Web systems have to deal with, assuming they take the specifications at face value. I believe the developer community is generally overloaded with what they see, leading to workarounds and simplifications that encourage the development of systems that are inconsistent with the underlying specifications. Although there's been an upsurge in the development of specification-friendly "RESTful" systems (that is, treating HTTP as an interface rather than just a transport), barriers still exist to taking the Web further into a generalized *Web of data* and developing significantly more useful services.

I suggest that one of the biggest barriers is the apparent complexity of Semantic Web systems. Syntax issues are popular talking points, but instead let's consider the Web aspect from the bottom up. I theorize that it's merely the abstractions we're using that are too complex. There might be aspects that we can't (or shouldn't) hide, but there's nothing to be gained from unnecessarily stretching human conceptual capabilities. In the next column I'll offer a possible solution (agents!).

## Those Web Versions Again

In previous columns, I've made plenty of reference to the Web 2.0 label, which brings with it the implication that there was a Web 1.0 and will be a Web 3.0 (although arbitrary and potentially misleading, these labels are useful for drawing lines in the sand).

Web 1.0 was concerned with HTTP and the HTML format, with distributed documents bound together through hypertext links. Web 2.0 brought improved user experience through client-side scripting and introduced a wider range of content-delivery formats. Asynchronous calls to the HTTP server from the browser — Asynchronous JavaScript and XML (Ajax) — have enabled more fluid interactions, and machine-readable data has made browser-based information merging viable, producing novel combinatorial applications known as *mashups*. Web 3.0 has (among other things) the promise of powerful services and sophisticated data integration, not least through Semantic Web technologies, to fully generalize from a Web of documents to a Web of data.

Clearly, one reason for the Web's viral success has been its relative simplicity. Clients (browsers) and servers are available as low-cost commodity software, deployable on a wide range of operating systems. In Web 1.0, the budding site developer produced material by copying and pasting HTML code, with most browsers' View Source capability offering insight into existing documents. In Web 2.0, the growth of content management systems (CMSs) has meant that publishing on the Web requires virtually no technical expertise. Anyone with basic desktop application skills can produce quality online publications, and an awful lot of people are doing just that in the form of blogs. Figure 1 shows a typical system architecture. If we are to extend to a Web of data, we need to figure out how best to go beyond simple data viewers and look at how systems can integrate and process that data with minimal human interaction.

## Services and Data

Two closely related approaches reduce the human work necessary to achieve a particular end and make the Web more capable. The first is to build machine-oriented services that let computers sort things out among themselves with less human intervention. The other approach is to increase the amount of useful data on the Web. Of course, for data to be really useful requires some kind of services. Conversely, for services to do useful work they need data on which to operate. Another consideration is that making data useful in the global

 1089-7801/07/$25.00 © 2007 IEEE 

environment requires conventions on how the data is expressed so that disparate, independent systems can all make sense of it. The Web is built upon HTTP, which can support services. The typical interaction of participants is a simple request–response between browser and host, but it's reasonable to say that every site is already implementing a service by delivering information according to HTTP.

In recent years, Semantic Web technologies have brought Web-friendly data languages to the table, so you might imagine that there would be a clear way to improve the Web through HTTP and Semantic Web-based HTTP services.

## Simple Is as Simple Does

What makes the Web the Web is its links: documents contain appropriately encoded references to other documents, and Web browsers know how to interpret these references and let users navigate the Web hyperspace. The markup for HTML links is simple, based around uniform resource identifiers (URIs), and the notion of linkage is relatively straightforward to other languages (see "Evolving the Link," *IEEE Internet Computing*, 2007, vol. 11, no. 3, pp. 96, 94–95).

At the protocol level, things get rather more complicated with URIs, identity resources, conceptual entities, and so on. When clicking on a link, users want to see in their browsers a representation of the resource, typically a HTML document. But a single resource might have multiple representations of different media types — text, HTML, image formats, and so on. When a URI is dereference in a browser, part of the client-server messaging is concerned with content negotiation. The browser informs the Web site host of the kind of material it would like to receive, and in return receives the representation that the server determines is the best match. The protocol level's complexity is hidden from the browser, largely because the browser has preferred media types (with HTML at the top of the list). For the Web publisher, complexity tends to be hidden through conventions such as the "magic" interpretation of particular media types' filename extensions (an `example.jpg` file, for example, will be delivered over HTTP with media type `image/jpeg`). Sometimes such conventions do offer conceptual simplifications without distorting the reality of the protocol. Although there's no definitive specification for the commonly used term "URL" when used to describe URIs that might be dereferenced over HTTP, there's no real loss of fidelity. But in general, the interfaces that a large proportion of Web developers program against seem to bear only passing resemblance to what's actually contained in the relevant specifications — they program against what their tools offer them.

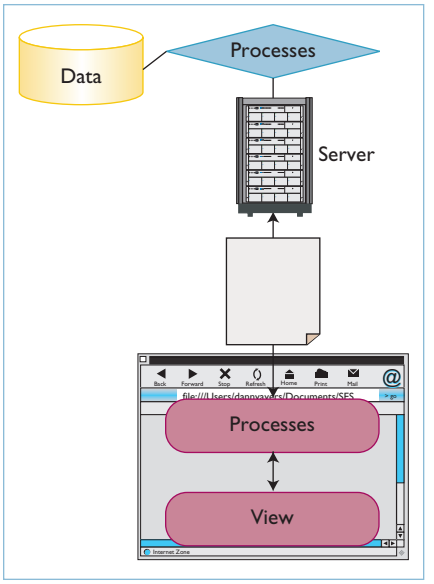


*Figure 1. Typical Web 1.0 content management system. Content (with its associated metadata) is available to processes managed on a server, which exposes documents to clients. The user interacts with the system through a view of those documents, as presented by a browser.*

## Using More of the Web?

Programmers (like most humans) commonly use abstractions of complex systems to provide simpler sets of symbols, letting them deal with phenomena without having to delve into unnecessary detail. The Web's core conceptual model is that of resources and their representations. This seems suitably simple and appears to be a reasonable fit for the typical operations carried out by HTTP servers, browsers, and intermediaries such as caches. However, even without dropping down to the level of messages on the wire, this model's simplicity is illusory. Figure 2 shows a resource and its set of representations of different media types.

Yet, a resource's individual representations aren't necessarily complete. There's no expectation that a photograph of a horse will contain all the information found in other representations — the horse's pedigree, for example. It's up to the publisher to decide whether that is a valid representation. What's more, although we can consider the association between a resource and its identifier permanent — "Cool URIs don't change," according to Tim Berners-Lee (www.w3.org/Provider/Style/URI) — individual representations almost always change over time. *IEEE Internet Computing's* homepage URI is http://computer.org/internet (among others), but the HTML representation's content varies from issue to issue.

If we look at how Semantic Web technologies express data on the Web, the plot thickens further. As Figure 3 shows, one or more of a resource's representations might describe a Resource Description Framework (RDF) graph. The media-type representation space is relatively well constrained: typically, documents will be RDF/XML or Turtle (although a Gleaning Resource Descriptions from Dialects of Languages-aware agent will also deal with arbitrary XML formats). Within the graph represented by the document,

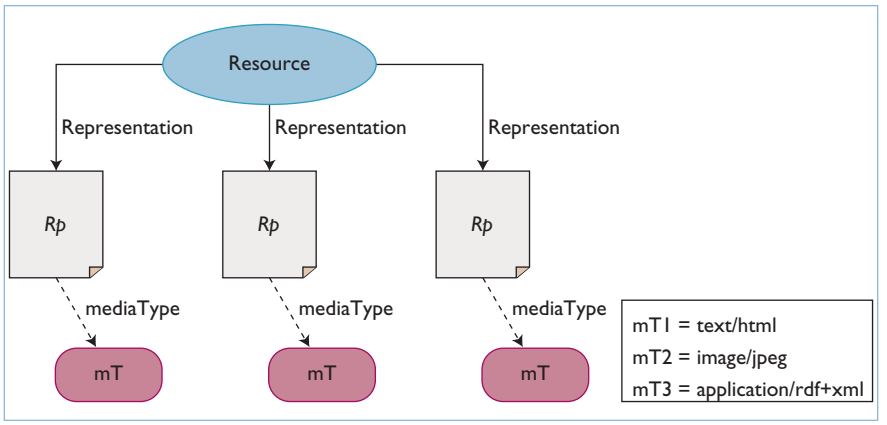


*Figure 2. A resource and its representations. The resource is a conceptual entity (identified by an uniform resource identifier), which is reflected on the computer by one or more concrete representations, series of bytes associated with media types which correspond to standard formats.*

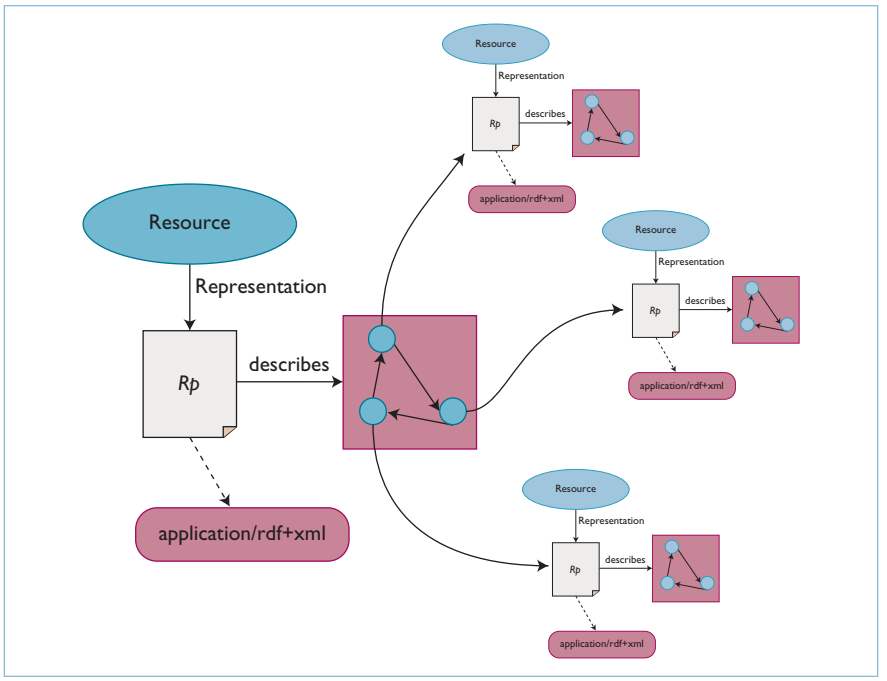


*Figure 3. RDF graphs on the Web. Here, the conceptual resource has a concrete representation in a standard RDF format (media type). This representation describes a graph model. The model's nodes and arcs might, in turn, correspond to other resources with RDF-interpretable representations.*

arcs have a fairly natural conceptual correspondence to links in HTML documents. However, a large proportion of nodes and arcs will have URIs beginning with "http." This offers the document's consumer the opportunity to follow its nose using the HTTP protocol and get representations of the identified resources to increase its knowledge. The resources identified in a document might well have their own graph representations, which inherit all Web documents' characteristics, including not being "complete" representations of the resource, varying over time, and so on.

## And the REST...

When talking about this abstraction of the Web, I've really been talking only in terms of the HTTP `GET` method, used by a client to ask a server for a resource's representation. Even the humble browser supports the `POST` method, and the HTTP specification includes other methods that might modify the identified resource: `PUT` and `DELETE`.

Other ways also exist to abstract messages and data over the Web. One example is the Web services framework, which began with SOAP as a protocol for wrapping data for delivery over HTTP. The best-known approach is to encode remote procedure calls. One way we can avoid complexity is to use only a small amount of the facilities that HTTP has to offer — typically, the `GET` and `POST` methods — with an additional protocol layer passing the responsibility for data description inside the message bodies.

However, if we're talking about maximizing interoperability with existing Web systems and leveraging the techniques that have scaled to Web proportions, we should use HTTP as designed. This involves working directly with the HTTP interface down at the transport layer, rather than addressing system-specific interfaces through opaque wrappers for method calls or data tunneled through messages.

Without wishing to disparage WS-* approaches, it seems self-evident that a Web developer should look to "raw" HTTP for communications and play nicely with the existing Web architecture unless there are very good reasons not to. The W3C Technical Architecture Group's publication, "Architecture of the World Wide Web, Volume 1" (www.w3.org/TR/webarch), is required reading for anyone remotely involved in Web systems.

Although the abstraction provided by the specifications and experts in the field might be more consistent than the abstraction assumed by many PHP

hackers and the like, it doesn't exactly provide an intuitive mental model for the programmer who'd rather code than read articles like this.

## Mashups

But wait, hasn't Web 2.0 solved the problem of service complexity on the Web with the mashup? Hardly. Although several useful and impressive mashups exist (see www.programmableweb.com), they're one-off combinations of their source data (see Figure 4). That data itself is usually expressed in domain-specific formats and translated to a local representation for display, with little opportunity for doing anything useful with the integrated data. Systems such as Yahoo Pipes (http://pipes.yahoo.com) hint at the potential for interwired services on the Web, but they're constrained to manipulating content and content-oriented metadata (RSS, for example) rather than having a generic data language such as RDF to play with. It's certainly impressive to see what people have been able to do in the browser, but the underlying flaw common to most browser-based Web 2.0 systems is that they take the same abstraction found server-side and echo it within the browser level. Server-side mashups rarely improve on this.

I hope I've made a reasonable case that, beyond the simplest system, "correct" Web development is *difficult*, and have suggested a likely cause is the level of abstraction with which the specifications express it. How can we avoid the burden of these complex abstractions? There isn't space here to go into detail — in my next column, I'll fill some of that in — but I believe a good approach might be to fairly radically abstract away many of the components we're familiar with, such as the client and server. Additionally, Semantic Web technologies make it possible for software systems to communicate with each other, so that if we can find abstractions which don't get in the way, the end result can be much more than a simple document view. Where I'm going with this is all the way back to the old artificial intelligence notion of intelligent agents, which has generally been either forgotten or maltreated when applied to Web systems. I believe this is feasible because most of the necessary components are now implemented in a form that treats them as commodities that can be joined with near-invisible glue.

**Danny Ayers** works for Talis as a development community liaison for their Semantic Web platform (http://talis.com/platform). His weblog is at dannyayers.com. Contact him at danny.ayers.ieee@gmail.com.

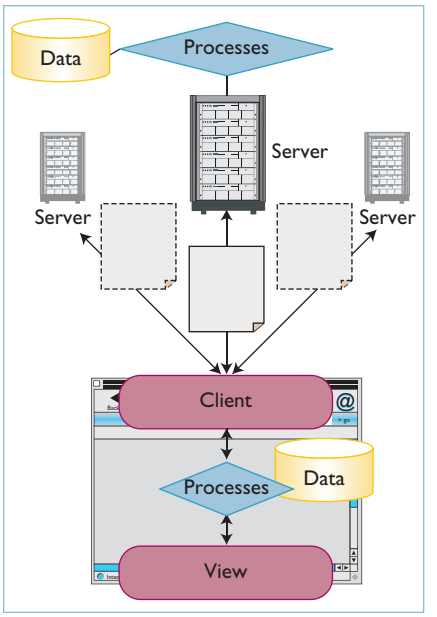


*Figure 4. Mashup architecture — more of the same. A typical mashup will take documents from multiple source servers, each with their own specialized processes and data storage, and combine their contained data locally using purpose-built processes and data containers.*