Cyberinfrastructure and Web 2.0

Marlon E. PIERCE, Geoffrey FOX[a, b, c], Huapeng YUAN[a, b], and Yu DENG[a,b]
[a] *Community Grids Laboratory, Indiana University*
[b] *Department of Computer Science, Indiana University*
School of Informatics, Indiana University

Abstract. We review the emergence of a diverse collection of modern Internet-scale programming approaches, collectively known as Web 2.0, and compare these to the goals of cyberinfrastructure and e-Science. e-Science has had success following the Enterprise development model, which emphasizes sophisticated XML formats, WSDL and SOAP-based Web Services, complex server-side programming tools and models, and qualities of service such as security, reliability, and addressing. Unfortunately, these approaches have limits on deployment and sustainability, as the standards and implementations are difficult to adopt and require developers and support staff with a high degree of specialized expertise. In contrast, Web 2.0 applications have demonstrated that simple approaches such as (mostly) stateless HTTP-based services operating on URLs, simple XML network message formats, and easy to use, high level network programming interfaces can be combined to make very powerful applications. Moreover, these network applications have the very important advantage of enabling "do it yourself" Web application development, which favors general programming knowledge over expertise in specific tools. We may conservatively forecast that the Web 2.0 approach will supplement existing cyberinfrastructure to enable broader outreach. Potentially, however, this approach may transform e-Science endeavors, enabling domain scientists to participate more directly as co-developers of cyberinfrastructure rather than serving merely as customers.

Keywords. Cyberinfrastructure, Web 2.0, distributed computing, network computing

Introduction

Cyberinfrastructure 1 and e-Science 1 are conventionally presented in terms of Grid technologies 2 3 that support remote access to computational science resources (such as supercomputers), distributed data management, networked instruments and similar technologies. Web Services are a key technology for realizing this vision 4 5. In contrast to these heavyweight approaches, however, many important innovations in network programming are emerging outside the (by now) traditional Web Services framework and are collectively known as Web 2.0 6. As we discuss in this chapter, these developments need to be tracked and incorporated into the e-Science vision. This chapter reviews some of the core Web 2.0 concepts by considering their impact on e-Science activities. For a related perspective on these issues, see 7.

Web 2.0 is not a specific set of technologies and is best characterized as a movement towards network programming for everyone. The blurred distinction between a Web application and a Web tool (such as Google's My Maps and Yahoo's Pipes applications) means that even non-programmers can make sophisticated custom Web applications. In contrast to Web 2.0, the Enterprise-centric view of Web technologies holds that Web development is the domain of highly trained programmers working in very sophisticated development environments with complicated service interface and message specifications. Enterprise technologies are characterized by various Web software vendor efforts from Microsoft, Oracle, IBM, Sun, HP, and others. Grid computing is closely aligned (through the Open Grid Forum 8, for example) with Enterprise computing. The numerous Web Service specifications (collectively known as WS-*) have formed the basis for much of Grid computing since 2001.

We compare Web 2.0 with Enterprise-style cyberinfrastructure applications in the Table 1. This table will also serve as a definition for our usage of the phrases "Enterprise Grids" and "Web 2.0" throughout this chapter.

Table 1. A comparison of Enterprise Grid and Web 2.0 approaches.

Network Programming Concept	Enterprise-Style Grids	Web 2.0 Applications
Network Services: programming interfaces and quality of service specifications.	Web Service APIs are expressed in WSDL. WS-* OASIS specifications for security, reliability, addressing, transactions, policy, etc are used. For a summary of these specifications, see 5.	Representational State Transfer (REST) services 9 are used: HTTP GET, PUT, POST and DELETE operations on URL resources are the universal API. Security is provided by SSL, HTTP Authentication and Authorization. HTTP error messages convey error conditions. No additional quality of service is provided.
Network State	WSRF 4 is used to model stateful resources.	Services are stateless (idempotent).
Network messaging	SOAP is used to convey XML message payloads. SOAP header extensions provide quality of service	Simple, more easily parseable XML formats such as RSS and Atom are exchanged. Like SOAP, Atom and RSS can include payloads of XML, HTML, etc.
Science Portals, Start Pages, and other service consumers	Science Portals are based on server-side standards such as JSR 168 portlets. Portal components are exchanged using WSRP. Science portals typically provide capabilities such as secure access to supercomputing resources.	Services are integrated using client-side applications and mash-ups 10. Self-contained JavaScript "gadgets" snippets can be embedded in Web pages. Web browser *Start Pages* aggregate RSS/Atom feeds, gadgets and widgets.
Service composition and aggregation	Workflow specifications orchestrate services and address business concerns such as support for database transactions. Scientific workflows 11 seek to capture science use cases. Workflows are typically specified with XML languages.	Mash-ups and mash-up building tools combine services into novel applications. Mash-ups are typically developed using scripting languages such as JavaScript.
Online communities	Virtual Organizations 12 are based on shared authentication and authorization systems. VOs work best for enabling cooperation between large, preexisting institutions (i.e. the NSF TeraGrid 13 or the NSF/DOE Open Sciences Grid).	Online communities are populated by volunteers and are self-policing. Work best as ephemeral, overlapping collections of individuals with shared interests.

As summarized in Table 1, there is a parallel between conventional cyberinfrastructure (Web Services, science portals, virtual organizations), and Web 2.0 (REST services, rich internet applications, online communities). Conventional cyberinfrastructure in these cases can be conceptually adapted to use Web 2.0 style approaches where appropriate. For example, not all computational services need to be associated with strong Grid-style authentication, and by loosening some of these requirements, the scientific community potentially will enable many outreach opportunities and (one would hope) more do-it-yourself Web-based computational science.

Combining Web 2.0 with conventional cyberinfrastructure also promises to enable virtual scientific communities. Grids typically are presented in terms of "Virtual Organizations", and the TeraGrid and Open Science Grid are two prominent examples. The Open Science Grid in particular is composed of a number of more-or-less dynamic virtual organizations. Grid-style virtual organizations tend, however, to focus on partnerships of real organizations that desire to share relatively scarce and valuable commodities such as computing time and access to mass storage at government-funded facilities. While this is necessary and has important consequences on security (such as the requirement for strong authentication), it is a limited (and Enterprise-centric) view of a virtual community. Web 2.0 community applications (such as Facebook, MySpaces, and other applications too numerous

and ephemeral to mention 14) have demonstrated that online communities can form and gain millions of members. Flickr and YouTube are well-known, more specialized social web sites dedicated to photo and movie sharing.

In the following sections, we review several of the technical underpinnings of Web 2.0 from the e-Science perspective. We organize these into network messaging and services; rich internet applications and user interfaces; tagging and social bookmarking; microformats for extending XHTML; a survey of Web 2.0 development tools; and a finally a comparison of Enterprise science portal development to Web 2.0-style Start Pages.

1. Network Services and Messaging in Web 2.0

1.1. Simple Message Formats

Unlike the SOAP-based Web Service specifications, many Web 2.0 applications rely upon much simpler news feed formats such as RSS 15 and (to a lesser degree) ATOM 16. These formats are suitable for simple parsing with JavaScript XML parsers. Like SOAP, both RSS and Atom carry content payloads. SOAP payloads are XML, which are drawn from another (i.e. non-SOAP) XML namespace or else encoding following conventions such as the remote procedure call convention. These encodings are specified in the Web Service Description Language (WSDL) interface, which a client can inspect in order to determine how to generate an appropriate SOAP message for the service. This can be done manually, but it is typical for a developer to use tools to hide the complications of constructing SOAP.

News feeds on the other hand convey content that can be XML but also can be text, HTML, binary files, and so forth. This is because the content is not strictly intended for machine processing, as in the case of SOAP, but also for display in readers and browsers. Interestingly, SOAP and WSDL are compatible with strongly typed programming languages (such as Java and the C/C++/C# languages), while the simpler feed formats match well with loosely typed scripting languages (JavaScript, PHP). Although there is not a strict separation (JavaScript libraries for SOAP exist, for example), it indicates the spirit of the two messaging approaches.

Atom and RSS formats are often combined with REST invocation patterns described in the next section. We give a simple Atom example in Section 1.3.

1.2. REST-Style Web Services

Representational State Transfer (REST) is a style of Web Service development that is based on manipulating URL-identified resources with simple HTTP operations (most commonly GET as well as PUT, POST, and DELETE). That is, REST services have a universal API, and effectively all the processing is done on the XML message associated with the subject URL. REST also explicitly avoids service statefulness that occurs in more closely coupled distributed object systems. Strict REST services are idempotent: identical requests will always give identical responses.

Less dogmatically viewed, the real advantage of REST is that it provides simple programming interfaces to remote services, even though the services themselves may be quite complicated. Such issues as transaction integrity and security are not exposed in the REST interface, although they will probably be part of the interior implementation details of the service. In contrast to WSDL and SOAP, REST services do not provide service interfaces to strongly typed messages. This makes them well suited for scripting language-based clients, and they are consequently are well suited for Web developers to combine into new services and custom interfaces (called mash-ups). For a list of mash-up programming interfaces, see the “APIs” section of the Programmable Web 10. At the time of writing, more than 430 APIs are available for mashup building.

This stands in contrast with much of the Web Service based development of cyberinfrastructure. Arguably, this has been in part derived from the Enterprise-centric view that, for example, one must expose the state transitions of complicated resources on a Grid. While this may be true in some cases, it has also led to extremely complicated

services and (in practice) fragile interdependencies on Web Service specifications. This also fosters the requirement for highly trained Grid specialists to implement and sustain the entire system.

The consequence of this is a division of labor in traditional Grid systems between the computer scientists and domain scientists. The computer science team members collect requirements and build systems while the domain scientists are customers to these end-to-end applications. In contrast, the REST approach combined with simple message formats potentially makes it possible for domain scientists, with general programming experience but perhaps not the specialized experience needed to build Enterprise-style services and clients, to be co-developers and not simply customers of services. That is, the domain scientists can take and use REST services to building blocks for their own custom applications and mash-up user interfaces.

1.3. An Atom Feed Example

News feeds are commonly associated with human-authored content, and the use of syndication formats distinguishes Web logs ("blogs") from simple HTML Web diaries. Feeds are associated with REST style applications since we only need standard HTTP operations (PUT and GET) to publish and retrieve feed information. In addition to human-authored content, it is also desirable to publish machine-generated data using single syndication format. This allows the feeds to be displayed in a wide range of clients (including mobile devices) with no additional development. In this section, we consider a very simple case study.

The Common Instrument Middleware Architecture (CIMA) project 17 builds Web Services and Web browser science portals for accessing both archival and real time data and metadata generated during experiments. CIMA is a general architecture, but its primary application is to manage data and metadata generated by crystallography experiments. Besides the actual data generated by the lab, it is also desirable to monitor lab conditions such as the temperature of system components. This information is collected by CIMA services.

To implement CIMA feeds, we chose the Atom 1.0 format over RSS because Atom complies more closely with XML standards than RSS (it has a defining XML schema, for example). It also allows message payload types to be included (HTML, plain text, XML, Base64 encoded binary images, and so forth). Atom is supported by most major news reader clients. We chose the Atomsphere Java package (http://www.colorfulsoftware.com/projects/atomsphere) for implementation.

The CIMA bay temperature service is a streaming service that pushes data to a CIMA client receiver. Since news readers use HTTP GET (a "pull" mechanism), we do not encode the actual data stream in Atom as it is emitted. Instead, we create a streaming (Java) client that intercepts the stream and converts it to the Atom format. The resulting Atom feed is then returned to the news feed client using HTTP GET.

A sample Atom feed of CIMA bay temperatures is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>-5a3d6d28:1127c3baa91:-8000</id>
  <updated>2007-05-11T15:34:50.16-05:00</updated>
  <title type="text">Bay1Temp Atom Feed</title>
  <author>
    <name>Yu(Carol) Deng</name>
  </author>
  <entry>
    <id>-5a3d6d28:1127c3baa91:-7fff</id>
    <updated>2007-05-11T13:42:04.182-05:00</updated>
    <title type="text">SensorName, TimeStamp, DoubleData</title>
    <content type="html">Bay1Temp  2007-05-11 19:34:08Z
   25.5&lt;br&gt;Bay1Temp  2007-05-11 19:34:29Z  25.5
   &lt;br&gt;Bay1Temp  2007-05-11 19:34:49Z  25.5
    </content>
  </entry>
```

```
</feed>
```

News readers vary in their tolerance of improperly formatted Atom, so it is useful to use one of the several online feed validators such as http://feedvalidator.org/, which we recommend, to check feeds for compliance. The feed listing above is shown in two different clients in Figure 1.

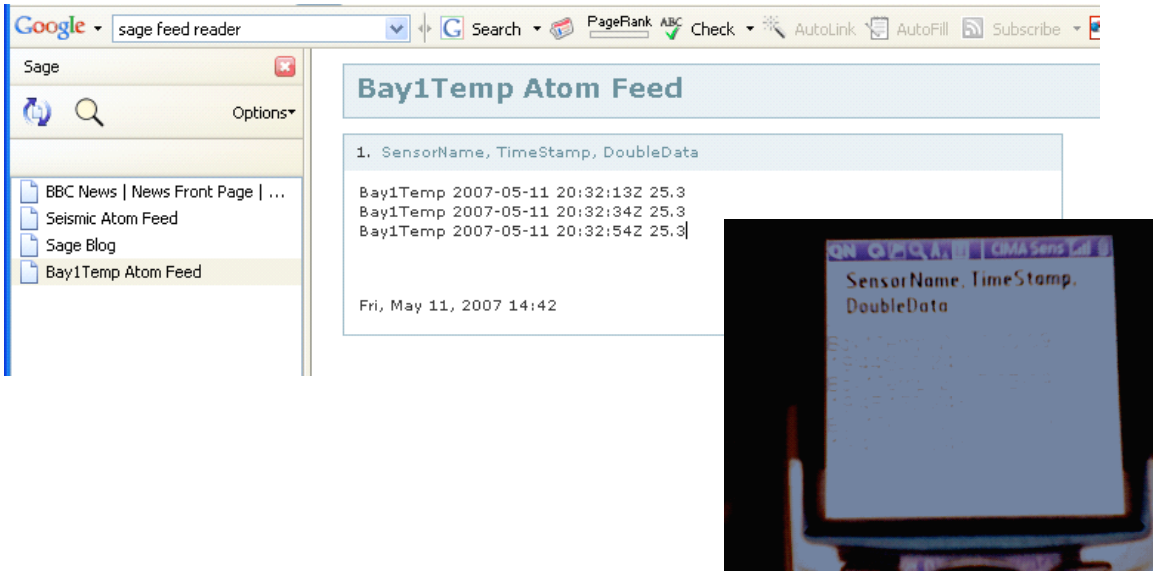

Figure 1. The Bay1 Temperature Atom feed using the Sage reader (upper left). The inset photograph (lower right) is of the same feed loaded in a cell phone's Web browser (photo courtesy of Rick McMullen).

1.4. REST for Online Computing Services: Storage and Computation

Interestingly, while computational science Grids have traditionally focused on providing access to computing, the more general online community has arguably found online storage services to be more useful. These services provide disk space for sharable online content (music and videos, for example). Amazon's Simple Storage System (S3) is a prominent example that has both SOAP/WSDL and REST style interfaces and a very simple shared secret key security model. S3 is a for-fee service, but many smaller startups provide storage services for free, if one is willing to take the risk and tolerate advertisements. Online storage systems may be used by individuals, but they also may be used by start-up social networking services as a backend storage system for shared binary content.

It is useful to consider the S3 security model through an example, which contrasts with the more complicated Public Key Infrastructure and GSSAPI based GSI model that dominates Grid systems and predates their transition to Web Services. To use S3, you must first create an account with Amazon using a credit card. As part of the registration process, you will be given a shared secret key (Amazon will have an identical copy) that can be used to digitally sign your communications with the service 26. This secret key is associated with a public identity. In communications with S3 services, the client will send the public identity of the key along with messages digitally signed by the secret key using one-way hashing (see 26 for overviews of various cryptography techniques). Amazon will use the public ID to look up its copy of the secret key and confirm the signature by reproducing the message hash. In the REST version of this service, all remote operations are performed by sending an HTTP command: PUT the file, GET the file, or DELETE the file. Clients write the contents directly to the remote resource using standard HTTP transfer mechanisms.

Security is handled by a custom HTTP request property called "Authorization". This is a string placed in the HTTP request stream that has the form

"AWS "+"[your-access-key-id]"+":"+"[signed-canonical-string]"

The canonical string is a sum of all the "interesting" Amazon custom headers that are required for a particular communication. This is then signed by the client program using the client's secret key. Signing is performed using standard libraries, such as Java's MessageDigest class or equivalents in PHP and Ruby. Amazon also has a copy of this secret key and can verify the authenticity of the request by checking the digest value.

Your uploaded files will be associated with the URL

https://s3.amazonaws.com/[your-access-key-id]-[bucket-name]/

That is, if your key's public ID is "ABC123DEF456" and you create a bucket (a simple organizational folder) called "my-bucket", and you create a file object called "test-file-key", then your files will be in the URL

https://s3.amazonaws.com/ABC123DEF456-my-bucket/test-file-key

By default, your file will be private, so even if you know the bucket and key name, you won't be able to retrieve the file without also including a signed request. This URL will look something like this:

https://s3.amazonaws.com/ABC123DEF456-test-bucket/testkey?Signature=xxxxx&AWSAccessKeyId=ABC123DEF456

Also note that this URL can be reconstructed entirely on the client side without any communication to the serve or maintenance of a security context. All the information needed is the name of the bucket, the object key, and access to the secret key. Note even though the URL looks somewhat random, it is not, and no communication or negotiation is required between the client and Amazon S3 to create this.

Note also that this URL is in no way tied to the client that has the secret key. One could send it in email or post it to a blog and allow anyone to download the contents. It is possible to randomly guess this URL, but guessing one file URL would not help in guessing another, since the Signature field is a message digest hash of the file name and other parameters. That is, a file with a very similar name would have a very different hash value.

You can also set access controls on your files. Amazon does this with another custom HTTP header, x-amz-acl. To make the file publicly readable, you put the magic string "public-read" as the value of this header field. If your file is public (that is, can be read anonymously), then the URL

https://s3.amazonaws.com/ABC123DEF456-my-bucket/test-file-key-public

is sufficient for retrieval.

In addition to the S3 storage service, Amazon also offers an object lesson in how to build a virtual computing resource: its Elastic Compute Cloud (EC2) allows users, for a small fee, to directly access Amazon computing resources. Such applications are well suited for pleasingly parallel applications. It is worth noting also that Amazon has deliberately made using its services as simple as possible. EC2 resources can be accessed via command line tools and ssh (in traditional Linux fashion). While we cannot comment on the viability of these as commercial endeavors, they are certainly worth using, examining, and mining for ideas. Finally, in closing, we note the Web 2.0 model at work: the Amazon S3 and EC2 service implementations are undoubtedly quite sophisticated. However, none of this sophistication is exposed in the REST programming interface.

2. Rich Internet Applications

REST services emphasize simplicity of invocation patterns and programming interface design. Sophisticated client interfaces are at the other end of the Web 2.0 spectrum. Adobe Flash plugins have shown for a number of years that the browser can overcome the limits of the HTTP Request/Response cycle to provide a more desktop-like experience for Web applications. More recently, the standardization of JavaScripts's XmlHttpRequest object (originally developed by Microsoft for Internet Explorer only but now supported by most major browsers) has enabled non-proprietary rich Web clients to proliferate. The core concept of rich user interfaces is that the Web browser can make calls back to the Web server (or Web servers) to request additional information without the user's direct request. Instead, the call-backs are driven by JavaScript events generated by the user's normal interactions with the browser user interface. This enables Web-based user interfaces to much more closely resemble desktop applications from the user's point of view.

As has been pointed out in 19, this has an important implication for Web interfaces in general and on science gateways in particular. Following the terminology of Cooper 20, traditional Web browser applications, even very sophisticated ones, are still only "transitory" applications and not "sovereign" applications such as word processor and integrated development environment (IDE) tools. Transitory applications are intended only for use for short periods of time. Web mail and Web calendar applications (and all of electronic commerce) are examples. But these are not suitable for day-long, continuous usage. In contrast, one commonly uses a Word Processor, an Integrated Development Environment tool like Eclipse, or other desktop tools for hours at a time. Rich internet applications for collaborative text editing, code development, spreadsheets, and so on are beginning to emerge and demonstrate that properly developed Web applications can indeed be sovereign.

We paraphrase the discussion of 19 here because it is an important criticism of most science gateways. Gateways have long been developed to be sovereign applications for setting up and running computational science experiments, but their success has been limited in this domain for exactly the reasons pointed out: slow response times severely limit the interactivity of the user interface, making science portals very frustrating and limited to use as sovereign applications. Not surprisingly, most successful gateway applications have transitory interfaces: queue monitoring, job tracking, machine and network load monitoring, community data access, and so forth. The lessons and techniques of Rich Internet Applications can potentially have a dramatic impact on the next generation of gateway interfaces, allowing them to finally become sovereign applications. We now examine some of these techniques.

2.1. Ajax and JSON

Asynchronous JavaScript and XML (AJAX) is the combination of a set of pre-existing technologies that can be used to build the interactive Web client interfaces described above. AJAX's key idea is that JavaScript's XmlHttpRequest methods can be used to make calls back to the server on the user's behalf. These calls return XML messages, which can be parsed by JavaScript parsers. This latter restriction encourages the XML messages that are returned by the server to be small and uncomplicated (shallow trees suitable for stream parsing, for example). This avoids putting a computational burden on the browser and keeps the response time short to provide a higher level of interactivity. A side effect is that the XML messages, because they are simpler than is common in Web services, tend to be more human-comprehensible. RSS and Atom feeds are excellent candidates for such parsing, and streaming or pull-style parsers are much more useful than the more powerful but memory intensive and computationally demanding DOM parsers.

JavaScript Object Notation (JSON) is an alternative to XML for encoding data structures using JavaScript in over-the-wire messages. JSON objects do not contain executable JavaScript code, just JavaScript-encoded data structures. One would still need to develop a client side application with JavaScirpt and HTML to manipulate the data. XmlHttpRequest can be used to fetch JSON objects instead of XML, or JavaScript can be used to dynamically alter the HTML page to add an additional `<script>` tag. The `<script>`'s `src` attribute (normally used to download additional JavaScript libraries) can instead download JSON objects. This latter technique (or trick) provides a way to circumvent XmlHttpRequest security sandbox conventions and is known as cross-domain JSON. Instead of parsing the XML, the JSON object is directly cast to a client side JavaScript variable, after which it can be manipulated as any other JavaScript object. JSON provides a useful way to encode related data (similar to JavaBean objects and C structs) using JavaScript as the encoding format.

The primary advantage of JSON over XML is that it uses full-fledged scripting language data structures to represent data. This has the obvious advantage if one wants to pass data structures such arrays over the wire, since these are not part of the standard XML Schema. Web Services typically use SOAP XML array encoding format conventions, but these are very verbose compared to JSON arrays. The disadvantage of JSON is obviously that non-JavaScript JSON consumers and producers must parse or translate the JSON data, although numerous such tools are available. Prominent uses of JSON include del.icio.us, Yahoo Web Services, and Google Maps.

2.2. An Example: Google Maps

The well-known Google Maps application serves as an illustration of many of the principles of rich Internet applications. The Google Maps API provides a high level, object oriented JavaScript library for building interactive maps. These libraries include convenient XML parsers, object representations of maps and overlays, access to additional services such as geo-location services, and other useful utilities. Map images are returned to the user's browser as tiles fetched from Google Map servers. Unlike older Web map applications, this map fetching is done without direct action by the user (i.e. pushing an "Update Map" button). Rather, new map tiles are returned based on the user's normal interactions with the system: panning to new areas creates events that result in new map downloads. Google Maps thus illustrates both REST style interfaces (the maps are retrieved using HTTP GET) and rich user interfaces (AJAX/JSON style JavaScript libraries that encapsulate server callbacks). The API also supports simple message formats, as developers can load and parse their own XML data for display on the maps. For a more thorough analysis of the server side of Google maps and how to build a custom tile server, see 22.

3. Tagging, Shared Bookmarking, and Folksonomies

A particularly interesting Web 2.0 development has been driven in large part by the scientific community. Connotea and CiteULike are shared bookmarking Web applications that allow users to bookmark scientific journal URLs and describe these resources with metadata. With open archives projects funded and supported by scientific funding agencies, we expect many scientific journal fields to be dramatically opened for network-based text mining and scavenging. Google Scholar and Microsoft Live Scholar already apply online search techniques for finding and ranking online publications (using Google's PageRank approach, for example, to rate the quality of publications). A large number of scholarly databases (such as the NIH's PubMed) are also available and provide service interfaces as well as web-based user interfaces.

Shared bookmarks are commonly described with simple key words called tags. Before discussing tagging in detail, we first note the related field of automatic, domain-specific mining of online literature is a very promising scientific application for Web 2.0-style applications. For example, in collaboration with the Murray-Rust group at Cambridge, we have investigated the parallelized mining of chemical abstracts using the OSCAR tool 23 to identify and convert chemical structure names in text files into SMILE identification strings. SMILE strings concisely express the two dimensional structure of the small molecule in question, and it is thereafter possible to use this to drive structure-based calculations. One may also envision useful information that may be obtained from such mining. For instance, one may query to find all other research groups looking at chemical compounds of interest.

3.1. A Bookmarking Case Study: Del.icio.us

Del.icio.us is an example of a general purpose tagging and bookmarking service. Del.icio.us serves only as a place to store and annotate useful and interesting online web resources (URLs). Note as always that a "Web resource" can be easily extended to include not just URLs but any digital entity that can be described with a URI. That is, the digital entity does not have to be directly retrievable.

As with many Web 2.0 applications, del.icio.us comes with many different interfaces that support a wide range of interactions with the service.

- A public Web browser interface (which probably accounts for the vast majority of its usage);
- An exportable web snippet (in JavaScript) that can be embedded in other web pages such as blogs, showing personal tag and link collections ("rolls") and user information;
- Exportable RSS feeds of personal bookmarks (via http://del.icio.us/rss/your_user_name); and
- A public programming interface (described below).

Obviously, the important thing is the flexible ways for creating, accessing, and sharing information. Web 2.0 style applications are easily embeddable in arbitrary Web sites since they use the browser rather than the server as

the integration point. This makes them much more flexible and simpler than Enterprise-style applications such as Java portlets and WSRP (described below). We observe that this should have a profound impact on science gateway portals.

Just as del.icio.us's simplicity for creating and delivering Web content is in stark contrast to the very heavyweight Enterprise approach, its simple approach to programming interfaces is a challenge to the complications of Web services. The service interface is briefly summarized below.

- Update: returns the update time for a user's tags.
- Tags: get and rename methods return a user's tags and rename them, respectively.
- Posts: used to get, add, or delete tag postings to del.icio.us.
- Bundles: provides programmatic access to collections ("bundles") of tags.

These service interfaces are REST-like: the API is actually a set of rules for constructing HTTP GET URLs. These URLs emit XML messages or JSON objects (whichever the developer prefers) as return values. The XML returned by these services is particularly simple and suitable for parsing by parsers available in JavaScript, PHP, and so on. Thus it is easy to use del.icio.us as an online service for storing and retrieving tags and key words specific to ones preferences. Provided one can agree with the Internet community at large on the tags (or keywords) used, it is easy build applications that monitor del.icio.us for new resources that may be of interest and take an appropriate action. It is also easy to see how one may use such systems to build cliques, or communities, of users based solely on their shared interests.

As we have seen, the del.icio.us programming interface provides a mechanism for manipulating user or community-defined tags and tag collections. Tags are used to annotate and organize bookmarks, and they effectively define a very simple and unstructured but powerful keyword naming system. These are sometimes referred to as "folksonomies", as the keywords used to describe a particular resource are supplied by the user community. Reusing popular tags is encouraged (it makes it easy for others to find your bookmarks), effectively winnowing out redundant tags. Related tags can be collected into "bundles", but the relationship between the tags in a bundle is not explicitly defined. That is, there is no RDF-like graph, much less the logical relationships of the Semantic Web's OWL 2425.

While this is open to abuse (a resource may give itself popular but inaccurate tags to lure web traffic), the system also uses (effectively) community policing: sites with a particular keyword are ranked by the number of users who have recently added the site to their personnel collection.

4. Microformats

Microformats are an approach to making XHTML markups that separate styling and presentation from the data's meaning. Like AJAX, microformats are not a new technology but are instead the application of existing technologies in an innovative way. The key technical concept is simply to combine XHTML and tags to descriptively mark up content in ways that can be understood by humans and processed by computers. Although sometimes presented as an alternative to the Semantic Web 2425, microformats are arguably closer to providing a mechanism for implementing the famous model-view-controller design pattern within the Web browser.

While XHTML and are intended (when combined with Cascading Style Sheets) to allow custom markup definitions, the key microformats insights are

- User-defined tags can just as easily encode semantic meaning as custom formatting instructions, and
- The power of the approach comes when large communities adopt the same and conventions for common objects, such as descriptors of people, organizations, calendars, etc.

Thus the primary agenda for microformat proponents is to define small but useful markup standards (or conventions) and lobby for their widespread adoption. Prominent microformat examples include hCard and hCalendar, which are XHTML encoding variants of the vCard and iCalendar IETF specifications, respectively. The standards are simple to encode because they consist of just name-value pairs with no semantic relationships

explicitly defined. Dublin Core conventions for publication metadata would similarly be a good candidate for microformatting.

Microformats have another advantage that will probably be exploited in next-generation Web browsers: the browser can identify common or standard microformats and associate them with plugins and preferred external applications. For example, an hCalendar markup fragment may be exported to the user's preferred calendar tool.

Science portals seem to be a particularly good match for microformats for encoding scientific metadata. Obviously encoding large binary data sets using microformats is out of scope, but more realistically portals often deal with data and metadata presentation and manipulation and so would benefit from shared microformats and associated JavaScript libraries. For example, an earthquake fault can easily be encoded in as a set of name/value property values. We propose a hypothetical microformat for this below:

```
<div class="earthquake.fault">
      <div class="faultName">Northridge</div>
      <div class="latStart></div>
      <div class="lonStart></div>
      <div class="latEnd"></div>
      <div class="lonEnd"></div>
      <div class="strike"></div>
      <div class="dip"></div>
  </div>
```

This particular format, which would be embedded in an (X)HTML page retrieved by a Web browser could be associated with numerous rendering options by the page developer, hopefully chosen from pre-existing, reusable libraries and style sheets. For example, the above may be rendered (with the right helper) as either a Google Map display or a Web form for collecting user input. The user may want to alter the strike value, for example, as part of the process for setting up a finite element simulation.

5. Web 2.0 Application Development Tools

Web 2.0 applications can be built using any available tools and programming languages that have been used for Web and network programming. However, some tools have gained more popularity than others. Many of the lightweight, mash-up style Web applications are built using the so-called Linux-Apache-MySQL-PHP (LAMP) approach. MediaWiki, which powers Wikipedia, is a prominent example of a LAMP application. Ruby on Rails is another development environment that has attracted some attention for its transparent support for AJAX and built-in Object-Relational Mapping (ORM) database support. Enterprise development environments such as Java and .NET technologies are compatible with the overall Web 2.0 technologies as well. Note also that all of the above technologies are for server-side programming and generate the dynamic HTML and JavaScript needed for rich user interfaces.

Numerous tools (Ruby on Rails, Direct Web Remoting (DWR), Google Web Toolkit (GWT), and others) provide higher level libraries for generating the otherwise fragile JavaScript code needed for AJAX applications. Arguably, the primary language of Web 2.0 is really JavaScript, which has undergone a surprising resurgence, supported in large part by its use by Google and by Yahoo's YUI libraries.

Google's GWT provides an interesting example of a Web 2.0 development tool, as developers create Web applications using a Java Swing-like set of user interface classes that are (when deployed) used to generate HTML and JavaScript. The implications are interesting, as this runs counter to the conventional Model-View-Controller (or Model-2) Web application development strategies of most Enterprise frameworks. Java Server Faces (JSF) serves is perhaps the ultimate existing example of MVC web development: the user interface and code logic portions are almost completely decoupled. The assumption is that teams of developers will include both Web design specialists (who will develop the look and feel of the application using JSF XML tags) and programming specialists, who will write all of the backing Java code for interacting with databases, Web services, and other form actions.

GWT supports a very different assumption: the Web application will be designed and developed almost entirely by closely interacting groups of programmers. All HTML, links for style sheets, and JavaScript will be generated from compiled code. There is no imposed separation of responsibility among team members with different areas of specialization. Although to our knowledge no one has attempted to build a science gateway using GWT, this is arguably a more appropriate model for implementers of cyberinfrastructure.

6. From Science Portals to Widgets, Gadgets, and Start Pages

Science portals have been major constituents of cyberinfrastructure since its beginnings. Portals are extensively discussed elsewhere, and our work in the field of Grid portals is summarized in 27. Figure 2 shows a screen shot of our QuakeSim science portal to support earthquake science. QuakeSim exemplifies many of the characteristics of traditional science portals 28.

However, after examining Web 2.0 approaches, we believe that the Enterprise-based technologies used to build science portals are insufficient. In particular, Web 2.0 applications possess the following characteristics that are missing in traditional science portals:

1. Client-side integration of components from multiple service providers.
2. Multiple views of the same Web application (see for example the previous del.icio.us discussion and also Figure 3).
3. Simple programming interfaces that encourage "do it yourself" science mash-ups.
4. Widgets/gadgets that allow portal capabilities to be exported to other Web sites and start pages.

As we have mentioned earlier, many Web 2.0 web applications include embeddable JavaScript "gadgets" that enable portions of the site content to be embedded in other web pages. For example, Flickr, del.icio.us, and Connotea all have small JavaScript code snippets that users can embed in other pages.

The implications of this on science portals should be considered. Science portals tend to use the server, rather than the browser client, as the content integration point. Users may select from content provided by the server, but to add a new application or capability to the portal server requires that an entire new service be deployed. For our discussion here, we note that all the portal applications are deployed on the same server (although they are typically Web service clients to remote services). In contrast, widgets and gadgets (Figure 3) are JavaScript snippets that are embedded in the HTML, making the browser the integration point.

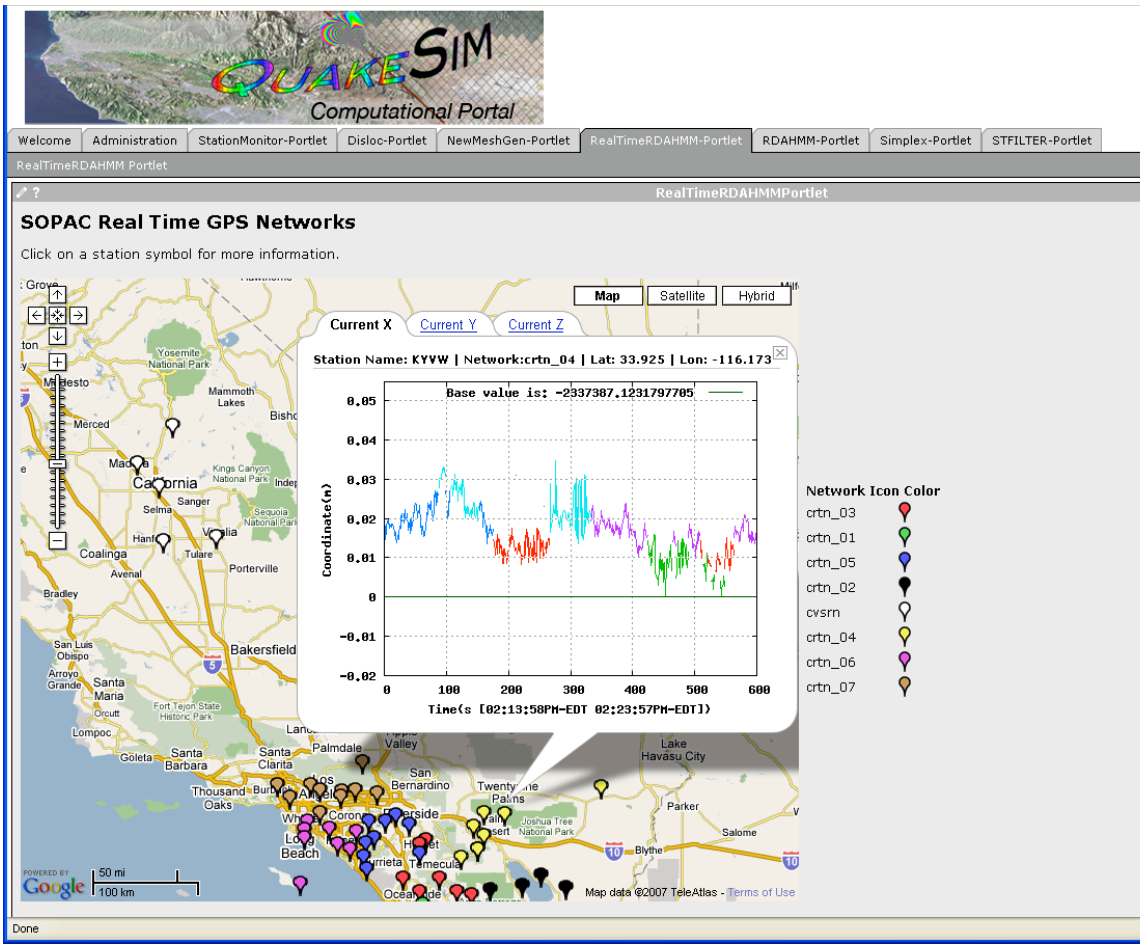

Figure 2. The QuakeSim portal screen shot shows results for the real-time analysis of Global Positioning System position data. This application is a portlet and corresponds to a web application on the portal server. Other portlets available through this portal ("StationMonitor-Portlet", "Disloc-Portlet", etc) are listed as tabs across the top beneath the logo.

Figure 3. Two del.icio.us gadget examples (inset) embedded in a blog. The gadget shows the user's latest bookmarks (top of inset) and tag cloud (bottom of inset). The blog itself generates news feeds (using Atom in this case) that can be embedded in Start Pages.

Although gadgets can be placed in any HTML page, it is common to aggregate them in personalized home pages or "Start Pages". Popular examples of start pages include Netvibes and iGoogle. Start Pages aggregate news feeds, calendars and other gadgets built by the community. We examine this process in the next section.

6.1. Creating Widgets and Gadgets: Example

We will now review the process for creating a very simple science portal gadget using Google. This can be integrated into a user's iGoogle personalized home page. We base this on our Open Grid Computing Environments portal software, which uses the portlet model illustrated in Figure 2. This gadget will manage a user's logon and open the portal as a separate window.

We first create a simple XML description for the gadget and place in a URL. For example, content of the gadget descriptor located at http://hostname:8080/gridsphere/ogcegadget.html is

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="OGCE Portal" />
<Content type="url" href="http://hostname:8080/gridsphere/ogce1.html" />
</Module>
```

The content of the gadget shown here is another URL pointing to the actual HTML page that we want to load. The gadget HTML itself can include JavaScript and other markups. Once created, this gadget can be added to a Google Start Page in the usual manner (click the "Add Stuff" link).

Figure 4. A personalized Google home page with the OGCE logon gadget added to the upper left corner.

Netvibes provides a powerful alternative to iGoogle. Like iGoogle, Netvibes provides clients that can consume RSS/Atom feeds and embed externally provided widgets. Developers wishing to provide their own widgets make use of the Universal Widget API. To make a widget, one must simply provide a URL pointer to an HTML fragment that follows Netvibes widget conventions. An example listing is shown below (and see inline comments marked by <!-- -->):

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <!—Start with meta data tags-->
    <meta name="author" content="Huapeng Yuan" />
    <meta name="description" content="A Netvibes Widget for OGCE" />
    <meta name="apiVersion" content="1.0" />
    <meta name="inline" content="true" />
    <meta name="debugMode" content="false" />
    <!—Additional meta tags can be used to control refresh rates-->

    <!—Use Netvibes style sheets -->
    <link rel="stylesheet" type="text/css"
      href="http://www.netvibes.com/themes/uwa/style.css" />
```

```
    <!—Import Netvibes JavaScript libraries -->
    <script type="text/javascript"
       src="http://www.netvibes.com/js/UWA/load.js.php?env=Standalone">
    </script>

    <title>OGCE Portal</title>
    <link rel="icon" type="image/png"
      href="http://www.netvibes.com/favicon.ico" />
  </head>
  <!—Begin HTML web form -->
  <body>
     <form action="http://hostname:8080/gridsphere/gridsphere?cid=login"
           method="post" id="form2" target="_blank">
        <p>
        <input name="JavaScript" value="" type="hidden" />
        </p>
        <table>
            <tr>
                <td  style="width:100">
                    <span> UserName  </span>
                </td>
                <td style="width:60">
                    <input type="text" name="username" size="20"
                           maxlength="50" value="yuanh"/>
                </td>
                <td >
             </td>
            </tr>
        </table>
        <table>
            <tr>
                <td style="width:100">
                    <span> Password </span>
                </td>
                <td style="width:60">
                    <input type="password" name="password" size="20"
                    maxlength="50" value="qwerty"/>
                </td>
                <td ></td>
            </tr>
        </table>
        <table>
            <tr>
                <td>
                    <input type="submit" name="gs_action=gs_login"
                     value="Login"/>
                </td>
            </tr>
        </table>
     </form>
  </body>
</html>
```

The example listing is standard (X)HTML for a Web form, with some additional Netvibes meta tags that provide necessary metadata about the widget. The basic steps are

1. Develop a widget as a standalone XHTML page.
2. Add Netvibes-required meta tags to the HTML as shown.
3. Use Netvibes-provided CSS and JavaScript libraries.
4. Write (if desired) control code for the widget's behavior using JavaScript libraries provided by Netvibes.

The final step is not used in our example (which invokes a login form) but in general one would use Netvibes JavaScript libraries to control newsfeed and (more generally, AJAX or JSON) data loads, parse XML responses, cast JSON code, and other JavaScript tasks as discussed previously. Note our simple HTML form-based example will actually redirect you to the page specified by the action attribute, so to keep actions within the user's start page, you must process the user's actions with JavaScript.

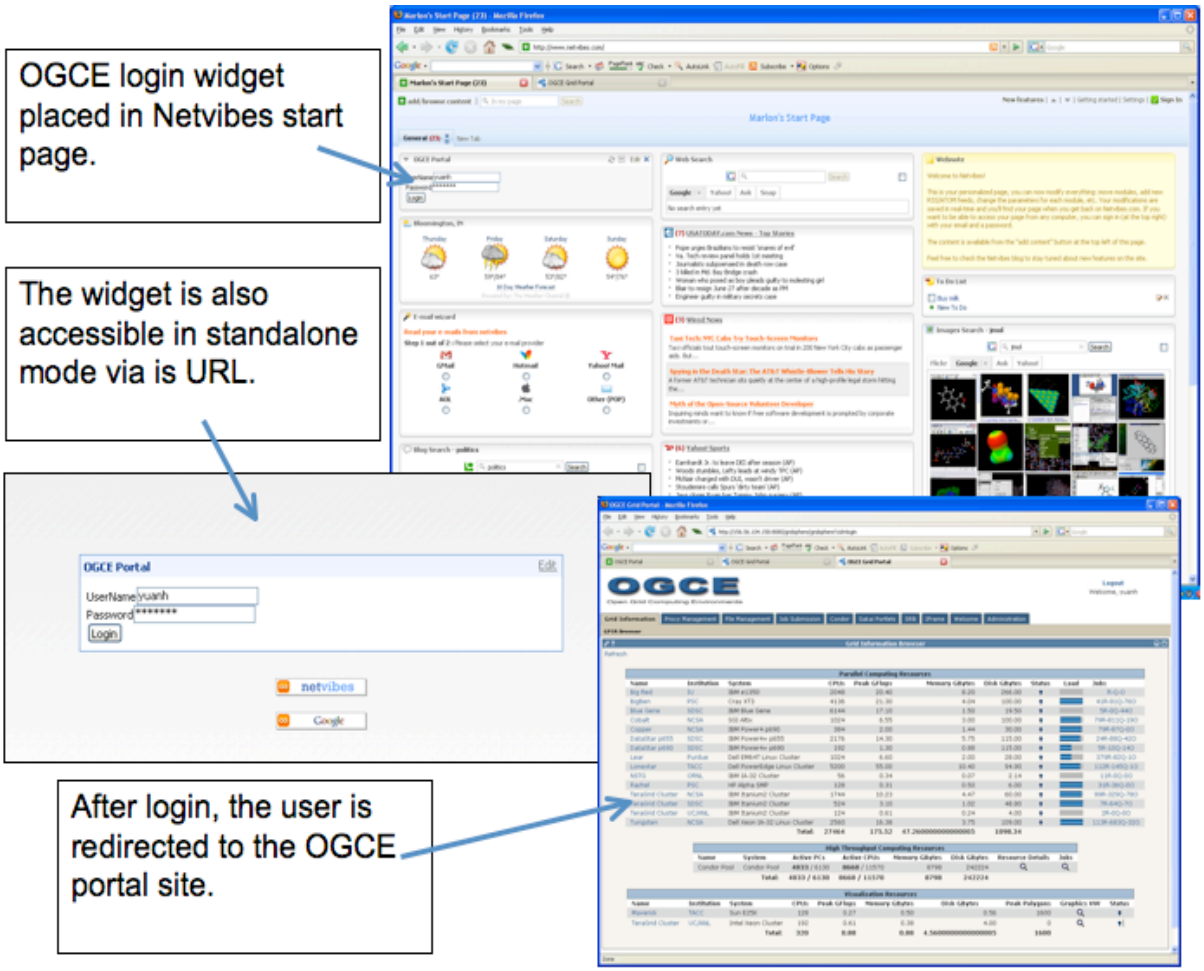

Figure 5. The Netvibes login widget API can be used to create custom portal widgets.

7. Conclusions

We have reviewed Web 2.0 technologies with an attempt to show how they can be used and combined with Cyberinfrastructure and e-Science. As the term e-Science implies, much work on cyberinfrastructure has focused on Enterprise-style applications that we surveyed in column 2 of Table 1. The Enterprise approach may be roughly characterized as requiring specialized programming and development expertise with particular tools (such as the Globus toolkit or the OGSA-DAI database services) and places an emphasis on sophistication (such as complicated XML specifications, security concerns, reliability, database transaction support, etc.).

With some reservations on coining yet another term, we propose i-Science as an alternative to e-Science that uses lessons learned from Web 2.0. i-Science is network based and should encourages "do it yourself" Web science applications. i-Science project teams should closely integrate domain scientists as co-developers and not simply treat them as customers and requirement sources for the IT development team. To accomplish this, services and components must hide complexity, rather than expose it. Interfaces and message formats must be simple, so that anyone with a reasonable amount of programming skill can learn to develop applications that add value.

8. Acknowledgements

We thank Rick McMullen and Kia Huffman of the CIMA project for help creating the Atom feeds.

References

1. D. E. Atkins, K. K. Droegemeier, S. I. Feldman, H. Garcia-Molina, M. L. Klein, D. G. Messerschmitt, P. Messina, J. P. Ostriker, and M. H. Wright, "Revolutionizing Science and Engineering Through Cyberinfrastructure." Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure, January 2003. Available from http://www.nsf.gov/cise/sci/reports/atkins.pdf.
2. A. J. G. Hey, G. Fox: Special Issue: Grids and Web Services for e-Science. *Concurrency - Practice and Experience* 17(2-4): 317-322 (2005).
3. F. Berman, G. Fox, and T. Hey, T., (eds.). *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley & Sons, Chichester, England, ISBN 0-470-85319-0 (2003). http://www.Grid2002.org.
4. I. Foster and C. Kesselman (eds.) *The Grid 2: Blueprint for a new Computing Infrastructure*, Morgan Kaufmann (2004).
5. I. Foster., "Globus Toolkit Version 4: Software for Service-Oriented Systems." *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, pp 2-13, 2006.
6. M. P. Atkinson, D. De Roure, A. N. Dunlop, G. Fox, P. Henderson, A. J. G. Hey, N. W. Paton, S. Newhouse, S. Parastatidis, A. E. Trefethen, P. Watson, and J. Webber: Web Service Grids: an evolutionary approach. *Concurrency - Practice and Experience* 17(2-4): 377-389 (2005).
7. P. Graham (November 2005). Web 2.0. Available from http://www.paulgraham.com/web20.html.
8. D. De Roure and C. Goble, "myExperiment – A Web 2.0 Virtual Research Environment". To be published in proceedings of *International Workshop on VirtualResearch Environments and Collaborative Work Environments*. Available from http://www.semanticgrid.org/myexperiment/myExptVRE31.pdf.
9. The Open Grid Forum: http://www.ogf.org/
10. R. T. Fielding, and R. N. Taylor: Principled design of the modern Web architecture. *ACM Trans. Internet Techn.* 2(2): 115-150 (2002).
11. For a list of public mash-ups, see The Programmable Web: www.programmableweb.com.
12. G, C. Fox, D. Gannon: Special Issue: Workflow in Grid Systems. *Concurrency and Computation: Practice and Experience* 18(10): 1009-1019 (2006).
13. I. T. Foster, C. Kesselman, and S. Tuecke: The Anatomy of the Grid - Enabling Scalable Virtual Organizations *CoRR* cs.AR/0103025: (2001).
14. C. E. Catlett: TeraGrid: A Foundation for US Cyberinfrastructure. *NPC 2005*: 1. See also http://www.teragrid.org/.
15. For a list of social networking Web sites and their claimed memberships, see http://en.wikipedia.org/wiki/List_of_social_networking_websites.
16. D. Winer, "RSS 2.0 Specification". Available from http://cyber.law.harvard.edu/rss/rss.html.
17. M. Nottingham and R. Sayre (Eds), "The Atom Syndication Format." *Internet Engineering Task Force RFC 4287*. Available from http://tools.ietf.org/html/rfc4287.
18. R. Bramley, K. Chiu, T. Devadithya, N. Gupta, C. A. Hart, J. C. Huffman, K. Huffman, Y. Ma, and D. F. McMullen: Instrument Monitoring, Data Sharing, and Archiving Using Common Instrument Middleware Architecture (CIMA). *Journal of Chemical Information and Modeling* 46(3): 1017-1025 (2006).
19. J. J. Garrett, "Ajax: A New Approach to Web Applications." Available from http://www.adaptivepath.com/publications/essays/archives/000385.php.
20. D. Crane, E. Pascarello, and D. James, *AJAX in Action*. Manning, Greenwich, 2006.
21. A. Cooper, "Your Program's Posture." Available from http://www.chi-sa.org.za/articles/posture.htm.
22. D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)". *Internet Engineering Task Force RFC 4627*. Available from http://tools.ietf.org/html/rfc4627.
23. Z. Liu, M. E. Pierce, and G. C. Fox Implementing a Caching and Tiling Map Server: a Web 2.0 Case Study Proceedings of the *2007 International Symposium on Collaborative Technologies and Systems* (CTS 2007).
24. P. Corbett, and P. Murray-Rust: High-Throughput Identification of Chemistry in Life Science Texts. *CompLife* 2006: 107-118.

25. The World Wide Web Consortium Semantic Web Activity: http://www.w3.org/2001/sw/.
26. N. Shadbolt, T. Berners-Lee, and W. Hall: The Semantic Web Revisited. *IEEE Intelligent Systems* 21(3): 96-101 (2006).
27. B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley and Sons, New York, 1996.
28. J. Alameda, M. Christie, G. Fox, J. Futrelle, D. Gannon, M. Hategan, G. von Laszewski, M. A. Nacar, M. Pierce, E. Roberts, C. Severance, and M Thomas. The Open Grid Computing Environments collaboration: portlets and services for science gateways. *Concurrency and Computation: Practice and Experience* Volume 19, Issue 6, Date: 25 April 2007, Pages: 921-942
29. N. Wilkins-Diehr and T. Soddemann: Science gateway - Science gateway, portal and other community interfaces to high end resources. *Proceedings of Supercomputing 2006*: 16.